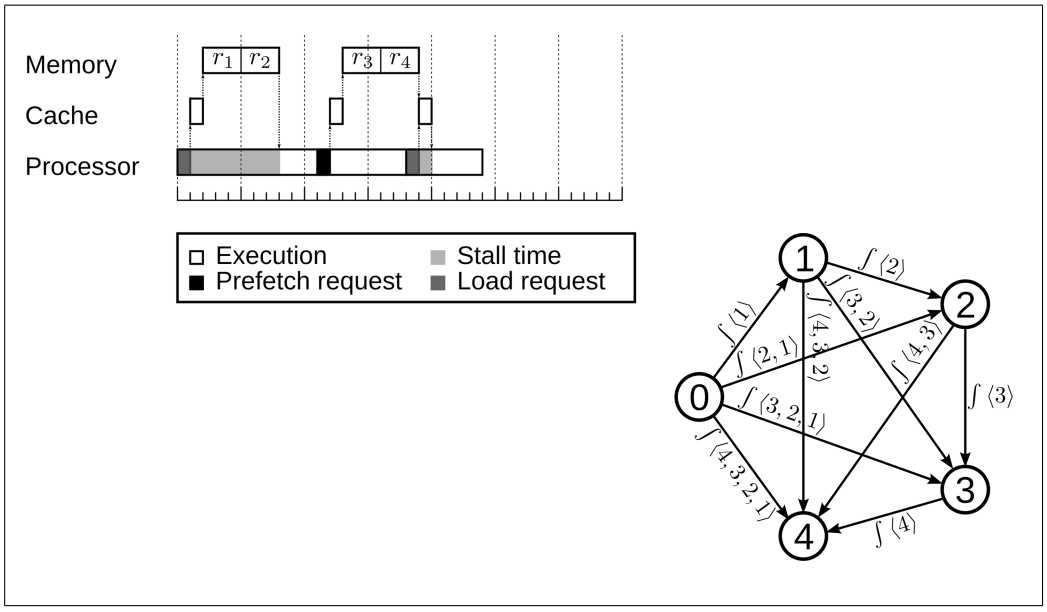
Scheduling for memory management and prefetch in embedded multi-core architectures

Sergiu Carpov

Par Sergiu CARPOV

Scheduling For Memory Management And Prefetch In Embedded Multi-Core Architectures.

Thèse présentée
pour l'obtention du grade
de Docteur de l'UTC.



Soutenue le : 14 octobre 2011
Spécialité : Technologies de l'Information et des Systèmes

UNIVERSITÉ DE TECHNOLOGIE DE COMPIÈGNE

SCHEDULING FOR MEMORY MANAGEMENT AND PREFETCH IN EMBEDDED MULTI-CORE ARCHITECTURES

THÈSE DE DOCTORAT

pour l'obtention du grade
de docteur de l'Université de Technologie de Compiègne
spécialité Technologies de l'Information et des Systèmes

présentée et soutenue publiquement
par
M. Sergiu Carpov

le 14 octobre 2011

devant le jury composé de :

Président :	M. A. Moukrim	Professeur à l'Université de Technologie de Compiègne
Rapporteurs :	Mme C. Hanen	Professeur à l'Université Paris X Nanterre
	M. E. Pinson	Professeur à l'Université Catholique de l'Ouest
Directeurs de thèse :	M. J. Carlier	Professeur à l'Université de Technologie de Compiègne
	M. D. Nace	Professeur à l'Université de Technologie de Compiègne
Encadrant :	M. R. Sirdey	Ingénieur-chercheur au Commissariat à l'Énergie Atomique

To my family

Abstract

This PhD thesis is devoted to the study of several combinatorial optimization problems which arise in the field of parallel embedded computing. Optimal memory management and related scheduling problems for dataflow applications executed on massively multi-core processors are studied. Two memory access optimization techniques are considered: data reuse and prefetch. The memory access management is instantiated into three combinatorial optimization problems.

In the first problem, a prefetching strategy for dataflow applications is investigated so as to minimize the application execution time. This problem is modeled as a hybrid flow shop under precedence constraints, an $\mathcal{NP}$-hard problem. An heuristic resolution algorithm together with two lower bounds are proposed so as to conservatively, though fairly tightly, estimate the distance to the optimality.

The second problem is concerned by optimal prefetch management strategies for branching structures (data-controlled tasks). Several objective functions, as well as prefetching techniques, are examined. In all these cases polynomial resolution algorithms are proposed.

The third studied problem consists in ordering a set of tasks so as to minimize the number of times the memory data are fetched. In this way the data reuse for a set of tasks is optimized. This problem being $\mathcal{NP}$-hard, a result we have established, we have proposed two heuristic algorithms. The optimality gap of the heuristic solutions is estimated using exact solutions. The latter ones are obtained using a branch and bound method we have proposed.

Résumé

Cette thèse est consacrée à l'étude de plusieurs problèmes d'optimisation combinatoire qui se présentent dans le domaine du calcul parallèle embarqué. En particulier, la gestion optimale de la mémoire et des problèmes d'ordonnancement pour les applications flot de données exécutées sur des processeurs massivement multicœurs sont étudiés. Deux techniques d'optimisation d'accès à la mémoire sont considérées : la réutilisation des données et le préchargement. La gestion des accès à la mémoire est déclinée en trois problèmes d'optimisation combinatoire.

Dans le premier problème, une stratégie de préchargement pour les applications flot de données est étudiée, de façon à minimiser le temps d'exécution de l'application. Ce problème est modélisé comme un flow shop hybride sous contraintes de précédence, un problème $\mathcal{NP}$-difficile. Un algorithme de résolution heuristique avec deux bornes inférieures sont proposés afin de faire une estimation conservatrice, quoique suffisamment précise, de la distance à l'optimum des solutions obtenues.

Le deuxième problème traite de l'exécution conditionnelle dépendante des données et de la gestion optimale du préchargement pour les structures de branchement. Quelques fonctions économiques, ainsi que des techniques de préchargement, sont examinées. Dans tous ces cas des algorithmes de résolution polynomiaux sont proposés.

Le troisième problème consiste à ordonner un ensemble de tâches de façon à maximiser la réutilisation des données communes. Ce problème étant $\mathcal{NP}$-difficile, ce que nous avons établi, nous avons proposé deux algorithmes heuristiques. La distance à l'optimum des solutions est estimée en utilisant des solutions exactes. Ces dernières sont obtenues à l'aide d'une méthode branch-and-bound que nous avons proposée.

Acknowledgements

First and foremost, I want to address my respectful acknowledgements to Mr. Jacques Carlier and Mr. Dritan Nace, both professors at Université de Technologie de Compiègne, for directing my research work and sharing their eminent expertise in the field of combinatorial optimization. It was during my Master 2 degree studies in 2007 that they managed to awake in me the desire to follow up with a research career.

Also, I want to address my sincere thanks to Mr. Renaud Sirdey, research engineer at Commissariat à l'Énergie Atomique, for supervising me on a daily basis from early 2008, for the confidence he had in me and for the numerous advices he gave me, professional as well as personal.

I wish to thank Mrs. Claire Hanen, professor at Université Paris X Nanterre, and Mr. Eric Pinson, professor at Université Catholique de l'Ouest, for honoring me in accepting to report on this thesis.

I express my appreciation to M. Aziz Moukrim, professor at Université de Technologie de Compiègne, for presiding my thesis jury.

My warmest affection goes to my dear Ana for encouraging me to pursue this experience of PhD thesis, for being near and for her support in the difficult moments. I express my kindest gratitudes to my mother Nina and to my father Sofronie for being an example worth following. I want to affirm my respect to my brother Costea for his unconditioned help. I will never forget what you all have done for me.

And, last but not least, I want to express many thanks to my friends, colleagues and fellow PhD students: Grigore, Radu, Oxana, Cristi, Stanca, Ion, Elena, Nicolas, Ilias, Thomas, Pablo, Oana, Valentin, Marius.

Contents

Acknowledgements **ix**

Introduction **5**

List of publications **11**

1 Research context and motivations **13**
1.1 Computation intensive embedded systems 13
1.1.1 Massively parallel processor architecture 14
1.1.2 Dataflow models of computation 16
1.1.3 Execution model 20
1.2 Memory access optimization techniques 24
1.2.1 Caching 24
1.2.2 Prefetching 26
1.2.2.1 Software prefetching 27
1.2.2.2 Hardware prefetching 29
1.2.2.3 Hybrid prefetching 32
1.2.2.4 Prefetching for parallel processors 33
1.3 Research motivation 34
1.3.1 Data prefetching memory access optimization 35
1.3.2 Data reuse memory access optimization 37

2 Prefetching for dataflow applications **39**
2.1 Problem formulation and complexity 40
2.2 Related works 41
2.3 Lower bounds 41
2.3.1 Global lower bound 1 42
2.3.2 Global lower bound 2 45
2.3.2.1 Inter-stage precedence relations 46
2.3.2.2 Jobs precedence relations 46
2.3.2.3 Cumulative previous work 47
2.3.2.4 Jackson's preemptive schedule 48
2.3.2.5 Energetic reasoning 48

2.3.2.6 $GLB2$ computation 50
2.4 List scheduling 50
2.5 Adaptive randomized list scheduling 51
2.6 Experimental results 54
2.6.1 Instance generation 54
2.6.2 Global lower bounds 54
2.6.3 List scheduling heuristics 56
2.7 Conclusions 61

3 Prefetching for branching structures **63**
3.1 Problem formulation 63
3.2 Related works 65
3.3 Expected execution time 66
3.3.1 Fractional prefetch 66
3.3.2 All-or-nothing prefetch 67
3.4 Worst-case execution time 69
3.4.1 Fractional prefetch 69
3.4.2 All-or-nothing prefetch 71
3.5 Conclusions 76

4 Task ordering and memory management **77**
4.1 Problem formulation and complexity 78
4.2 Applications 79
4.3 Related work 80
4.4 Memory management for a fixed sequence of tasks 81
4.5 Branch-and-bound 84
4.5.1 Branching rule 85
4.5.2 Lower bounds 85
4.5.3 Dominance relation 88
4.5.4 Selection rules 89
4.6 Heuristics 89
4.6.1 Two-stage TSP based heuristic (2STSP) 90
4.6.1.1 One-step history limited data reuse 90
4.6.1.2 A simple two-stage heuristic 90
4.6.2 Randomized cheapest insertion heuristic (RCI) 91
4.7 Computational results 91
4.7.1 Employed instances 91
4.7.2 Branch-and-bound evaluation 93
4.7.2.1 Random instances 93
4.7.2.2 Image convolution 94
4.7.3 Heuristics evaluation 96
4.7.3.1 Random instances 96
4.7.3.2 Image convolution 98

4.7.3.3 Two-stage heuristic evaluation 98
4.8 Conclusions 99

Conclusions and future work **101**

Bibliography **102**

Introduction

A combinatorial optimization problem consists in finding in a huge, but finite, set of elements an element which has the optimal cost. The branch of mathematics which treats such type of problems is one of its most frequently applied branches. Combinatorial optimization problems arise in many industrial fields, the computer industry is not an exception. Moreover, the combinatorial field experienced a tremendous growth from the time the first computer appeared to nowadays. New resolution methods and algorithms were possible due to computing capacities offered by the computers.

The computer industry is concerned with the development of software, the design of computer architectures and networks, the manufacture of computing systems, etc. Complex combinatorial problems routinely crop up in all these areas, a non exhaustive list includes program scheduling and optimization, network routing and dimensioning, ending with more classic production planning problems.

This thesis deals with a number of combinatorial problems arising in the field of *computation intensive embedded systems*. Such systems consist of the components needed to perform one or few specific computationally intensive tasks under soft or even hard real-time constraints. Schematically, such a computing system is made up of a microprocessor (possibly multi-core), some memory, IO interfaces and an application. Because of technological limitations, memory access speed is small when compared to the computing speed of processors. In order to reduce the performance gap between the memory subsystem and the computing subsystem special attention should be paid to the management of memory accesses. Namely, it is in this context of memory access optimization that our works are situated. The results of this thesis have been reported in a number of publications, refer to Chapter "List of publications" on page 11.

Many different ways for representing computer applications exist. One of them, which was proved to be viable for programming the currently emerging generation of massively multi-core architectures (more than a hundred of cores), is the dataflow model. In this model, an application is represented as a set of tasks, or equivalently actors, communicating through FIFO channels. The actors are sequential programs. Besides exhibiting a convenient kind of determinism, this model provides under certain conditions interesting properties: compile-time

knowledge of memory accesses, pseudo-static scheduling, compile-time communication channel dimensioning, etc. In the context of embedded applications these properties are enforced by the manner of programming and compiling embedded software. The goal of this thesis is to exploit the additional properties offered by the dataflow model for the optimization of the external memory access, via *data reuse* and *data prefetch* techniques, in the context of computation intensive embedded applications. In the following paragraph we are going to define in more details the data reuse and data prefetch optimization techniques.

The memory locations accessed by an application generally exhibit a certain degree of temporal and/or spatial locality. In other words, either the same location is accessed several times in a small interval of time or neighboring locations are accessed successively. A common optimization technique is to use a cache memory (possibly a hierarchy of caches). The cache is a fast memory, faster than the main memory, but it is smaller in size. The cache intent is to store the most frequently used main memory locations for latter reuse. This memory access optimization, which allows to reduce the number of times the same data is fetched from memory, is called *data reuse*. The mechanism which decides what memory locations to store in the cache and what locations to evict from the cache is called a cache replacement policy. Many cache replacement policies have been proposed. The lack of knowledge about future memory accesses makes the cache replacement policies to speculatively choose which locations to push out of the cache. From the quality of this "foreseeing" depends the application speedup. The accessed memory locations are gradually revealed during the execution of an application. The memory locations accessed by an instruction must be loaded into the cache before the execution of this instruction can start. Evenmore, it is possible to request the memory data several processor cycles before the instruction which uses them starts. In the meantime other instructions are executed. In this way the application is sped up by the saved cycles. The process of loading data in advance to their actual use, described above, is called *data prefetching*. The prefetch is *speculative* when there is no certainty that the fetched data will be effectively used (e.g. prefetching for branch instructions). The data prefetch can be interpreted as masking the memory access latency by processor computations. In parallel processors prefetching is further relied upon so as to keep several computing units fed with data, hence busy.

Our research focuses on clustered massively multi-core processors. They are composed of several hundreds of computational cores organized in clusters. Each cluster has its own shared memory[1] to which the computational cores have access. The memory hierarchy of these processors consists of the external (main) memory, the cluster shared memory (L2 cache) and the processor cache (L1 cache). In this hierarchy the data reuse and data prefetch optimizations have two instantiations: (i) between the external memory and the cluster memory, (ii) between the cluster

[1]Cluster shared memory can be either seen as a "normal" cache or as a scratchpad.

memory and the processor cache memory. The first instantiation corresponds to the optimization at the dataflow level of abstraction. Whilst the second one corresponds to the memory access optimization for an actor, which is simply a sequential program. The memory access optimizations aspects for sequential programs have been extensively studied in the literature. Here, we limit our discussion to the optimization of memory accesses between the external memory and the cluster shared memory.

The manuscript is organized in four chapters along with an introduction chapter and some concluding remarks. In what follows we give a brief overview of each chapter and the motivations behind the studied optimization problems.

The aim of Chapter 1 is to introduce the context, related works and motivation of our research in more details. The first section deals with the description of the components which form a computation intensive embedded system. Namely, an example architecture of a massively multi-core microprocessor, the dataflow model of computation for representing embedded applications and an execution model which allows to efficiently and correctly execute dataflow applications are described. In the following section a review of existing works on memory access optimization is introduced. More specific related works are introduced along the discussion in each chapter. In the last section of this chapter we state in more details on the motivation of our research.

Our first research problem, discussed in Chapter 2, consists in optimizing the data prefetching so that the execution time of a dataflow application is minimized. In a dataflow graph when an actor is executed, some data (possibly including code, actor proprietary or input data) may have to be loaded from the external memory to the cluster memory. Only after the data has been loaded the execution of the actor can start. Thus, an actor is executed in two steps: the loading of data and the execution itself. Only one communication channel exists between the external memory and the cluster memory, so the data loading operations have to be serialized. Whereas, the actor execution can be performed on one of the available computing cores. We model the prefetch optimization problem as a hybrid two-stage flow shop (HFS) under precedence constraints. More formally, we are given a set of n jobs. Each job has two operations, thus a job is executed in two stages. The first stage operations are executed on a single machine, whereas the second stage ones are executed on m parallel machines. The second stage operations are constrained by precedence relations. No preemption is allowed in operation execution. The objective of this problem is to minimize the makespan (the total execution time). Two HFS versions are considered, the classical version where idle time between the operations of the same job is allowed and the no-wait version where idle time is not permitted. The HFS problem is $\mathcal{NP}$-hard. An adaptive randomized list scheduling heuristic is proposed for solving the HFS problem. The distance to optimality of the obtained solutions are conservatively estimated using two global lower bounds which we

have introduced. The evaluation of the heuristic is done on a set of randomly generated instances. The solutions for the classical HFS in average are *provably* situated below 2% from the optimal ones, and on the other hand, in the case of the no-wait HFS the average deviation is below 5%.

In Chapter 3 we study the problem of optimal prefetch management for branching structures. Besides actors which have a static execution, as those considered in the previous optimization problem, the dataflow model of computation contains also data-controlled actors. The program structure which permits to conditionally execute one actor from a set of actors is called a branching structure. The branching structure selects the actor to execute in function of a conditional input. Often, it is possible to load some data for the n actors before the conditional input value is available. The interval of time during which the data prefetch is possible is called prefetching time and we suppose that it is a random variable. Our goal is to find a prefetch strategy to apply during the prefetching time such that the execution time is minimized. By prefetch strategy we mean either what ratios of data to prefetch for each actor, or, in which order to prefetch the actors. The branching structures are random processes for which we cannot employ deterministic objective functions. That is why we analyze two statistical objectives: (i) the expected execution time and (ii) the worst-case execution time. It is shown that the mathematical expectation objective leads to elementary models. In contrast to this, the case with the worst-case execution time objective requires more advanced resolution techniques. Nevertheless, for both objective functions and prefetching strategies polynomial resolution algorithms are given.

In Chapter 4 is studied a task ordering and memory management problem. As stated previously, an actor is executed in two steps: data loading and execution. The data loading durations are not necessarily constant, as we have supposed for previously discussed problems. Each actor uses a set of input data and some input data may be common for two or more actors. Executing consecutively (or near enough) two actors which are using the same data allows to cache these data into cluster memory and consequently to save some external memory accesses. Thus, the data loading durations can vary in function of the order in which the actors are executed. Given a set of n tasks, the goal of the task ordering and memory management problem is to find an optimal task execution order which minimizes the number of times the same input data is fetched. The execution of the actors is not constrained by any precedence relation, which, as we shall see further, is not an overly restrictive assumption. The task ordering and memory management problem was proved to be $\mathcal{NP}$-hard. Two heuristic algorithms are introduced for solving this problem. In order to compare the distance to the optimality of heuristic solutions either global lower bounds or optimal solutions are needed. There is little hope to be able to obtain tight global lower bounds, as this is already a very difficult task for a particular case of our problem. So, we introduce an exact resolution method. More particularly we describe the components of a

branch-and-bound algorithm. The evaluation of the heuristics and of the branch-and-bound algorithm is done using randomly generated instances having from 10 to 30 tasks. Approximatively 67% of the instances were solved to optimality by the branch and bound method. The best heuristic is situated in average below 2% from the optimum.

This thesis is concluded by a chapter which discusses several perspectives in terms of both research and application directions.

List of publications

Accepted

- Carpov S., Carlier J., Nace D., Sirdey R., "*Two-stage hybrid flow shop with precedence constraints and parallel machines at second stage*", Computers & Operations Research 39, pp. 736–745, 2012.
- Carpov S., Carlier J., Nace D., Sirdey R., "*Task ordering and memory management problem for degree of parallelism estimation*", Lecture Notes in Computer Science 6842, (Proceedings of the 17th Annual International Computing and Combinatorics Conference), pp. 592–603, 2011.
- Carpov S., Carlier J., Nace D., Sirdey R., "*Probabilistic parameters of conditional task graphs*", Proceedings of the 5th International Workshop on Advanced Distributed and Parallel Network Applications (ADPNA 2011), accepted.
- Carpov S., Carlier J., Nace D., Sirdey R., "*Speculative data prefetching for branching structures in dataflow programs*", Electronic Notes on Discrete Mathematics 36 (Proceedings of the International Symposium on Combinatorial Optimization), pp. 119–126, 2010.

Communications

- S. Carpov, "*Two-stage hybrid flow shop with precedence constraints and parallel machines at second stage*", 7th Optimeo day, PRiSM lab, Versailles, June 2011.
- Carpov S., Sirdey R., Carlier J., Nace D., "*Memory bandwidth-constrained parallelism dimensioning for embedded many-core microprocessors*", CPAIOR 2010 workshop on Combinatorial Optimization for Embedded System Design, Italy, 2010.
- S. Carpov, "*On memory bandwidth dimensioning for massively parallel processors*", 5th Optimeo day, Supélec, October 2009.

Chapter 1

Research context and motivations

1.1 Computation intensive embedded systems

According to Moore's law the number of transistors which can be placed on an integrated circuit doubles each 2 years. This rule was verified for more than 40 years and is expected to be true for at least ten more years. Despite of the exponential growth of transistor count the performance of practical computing systems does not follow the same growth rate. Several causes exist. The clock frequency cannot overpass a predefined limit because either the power consumption prohibitively increases[1] or the distance between the transistors is so small that it becomes comparable to the clock wavelength, which requires additional hardware resynchronization mechanisms between multiple parts of the integrated circuit. The only viable option, today, is to convert the additional transistors into computing power at relatively low clock frequency by designing and using parallel processing systems. Nowadays, the mainstream processors have gone multi-core and the next generation of circuits, known as massively multi-core chips, has to contain hundreds if not thousands of cores.

The majority of existing applications (excluding the programs used in computer simulations), which have appeared during the last decades, were developed with the principles of sequential programming in mind. Due to this, even if more computing power is available, under the form of parallel cores, it is not straightforward to exploit it. Such processors, which are intrinsically complex systems, require a systemic approach for their design, from new programming paradigms down to innovative compilation technologies and execution models. In the next subsections we are going to describe the elements of an innovative approach to

[1] The dynamic power dissipation of processors is proportional to the square of the operating voltage and linearly proportional to the clock value. The minimum admissible supply voltage augments with the clock frequency increase. That is why the increase of power dissipation due to the augmentation of the clock frequency cannot be compensated by reducing the supply voltage.

embedded parallel computing which will make the parallel microprocessors capacities accessible to mainstream programmers and not only to specialists in parallel programming. We start with the description of the hardware architecture. In the sequel, we describe a programming model and an execution model allowing deterministic and high performance execution of applications.

1.1.1 Massively parallel processor architecture

Roughly speaking, a parallel computing system integrates a number of processing elements, an interconnection network and a coordination mechanism that distributes the work between the processing elements. The Flynn's taxonomy [37] gives a coarse classification of computing systems, especially of parallel ones. Following this classification the computing systems can be divided in 4 groups according to the available number of parallel instructions and data streams: Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD) and Multiple Instruction Multiple Data (MIMD). Only the last three categories describe parallel execution possibility, while the first one is for usual sequential processors, where at each step one instruction is executed on a single data. Almost all of the parallel computing systems are either SIMDs or MIMDs. The SIMD are easily programmable because there is only one program flow, but the available parallelism is harder to exploit. On the other hand, in MIMD systems each processor executes its own program flow which makes them more flexible.

The existing parallel computing MIMD systems can be grouped, according to the memory organization, into *distributed memory machines (DMM)* and *shared memory machines (SMM)*. The distributed memory machines consist of a number of processing elements which are interconnected using a network. Each processing element has its own local memory which it can access directly. In order to retrieve data from other processor's local memory a message-passing operation is performed via the interconnection network. A special attention should be paid to the programming of DMMs, particularly to the data layout since local data are accessed significantly faster than non-local ones.

In a shared memory system a number of processing cores are connected to a shared memory space. The communications between the cores is done using common variables in the shared memory, thus by reading and writing to a shared address in the memory. Concurrent accesses to the same memory data by several processors should be carefully managed since unpredictable results can occur. An advantage of the SMM over DMM is that the communications via the shared memory are easy to perform and no data replication is needed. However, the increase of the number of processors in a SMM is limited because the interconnection fabric must provide a higher bandwidth. The use of a larger number of processors leads to an increase in effective memory bus access times since collisions

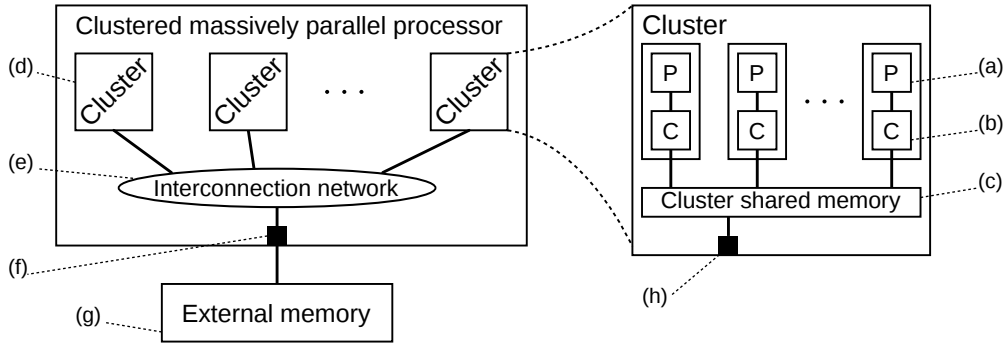


Figure 1.1: Clustered massively multi-core architecture (a - processor, b - processor cache, c - cluster shared memory, d - cluster, e - cluster interconnection network (NoC), f - external memory access controller, g - external memory, h - network adapter).

become more frequent. In real architectures the maximum number of processors rarely exceeds 32 on a single bus [97].

A *massively multi-core processor* is an integrated circuit containing hundreds, if not thousands, of processing cores, some memory hierarchy and the supporting infrastructure making possible their joint work. In order to be efficient such a processor needs to be scalable and provide a high level of performance. We have seen that the SMM systems are efficient in terms of computing power but hardly scale to more than 32 processors. At the same time, DMMs are easily scaled but the communication between the processors becomes slower, so for DMMs the performance increase is also upper-bounded. A *clustered massively multi-core processor* is a compromise solution between the drawbacks of SMMs in terms of scalability and of DMMs in terms of performance. Roughly speaking, this architecture is a DMM in which the nodes are SMMs, but not single processors.

Such a processor architecture is illustrated in Figure 1.1. This type of architecture is built up of many processing elements organized in clusters. A processing element contains a processing core and a private cache memory. Each cluster, besides its processing elements, has also a shared memory space. The architecture includes a network-on-chip (NoC), which interconnects the clusters. The computing system in which this processor is used, has an additional external memory for storing application data and instructions (which do not necessary fit the internal shared memory). External memory locations are accessed without direct core involvement, i.e. the latter only initiates and finishes the data transfers. The external memory access controller is responsible for fetching data from and to the external memory.

The processors of the same cluster can exchange data efficiently using the shared memory space. The same cannot be said about the processors belonging to different clusters which must use the chip interconnection network (NoC) for

data transfers. In order to assure a high degree of performance the NoC must provide enough bandwidth needed for the inter-cluster communication. That is why the NoC plays a primary role in the design of massively parallel processors. Basically a NoC is built up from routing nodes (routers) and links which are connected according to different *network topologies*. The topology describes the geometrical structure used to arrange the routers and links. Each cluster is connected to a routing node via a network adapter. In this way the clusters have access to the NoC. The communication on the NoC is packet-based and different *routing techniques* can be used. A routing technique determines along which path the packets are transfered from the sender to the receiver. The network communication times have practically the same value and are not proportional to the distance between the communicating clusters. For more information on NoC design issues please refer to [10].

1.1.2 Dataflow models of computation

Programming parallel systems is far from a straightforward job. Processor architectures with more than a hundred processing cores, with complex memory hierarchies and communication modules (NoC) are full scaled parallel machines. When an application is developed for such an intrinsically complex system one must handle at the same time the following issues: efficiently exploit system parallel architecture, allocate limited resources (memory, NoC), efficiently and correctly execute large parallel programs. The main obstacle in writing parallel programs is the synchronization between program tasks. One can express the synchronization structures explicitly, but this job is difficult and error-prone. In order to lighten the developer work and make the developed programs more reliable one should get rid of the synchronization constructs or at least mask their explicit use.

The dataflow programming model matches quite well the requirements exposed earlier. It allows to express parallel programs easily without worrying about explicit synchronization. The instructions in a dataflow program are executed when its operands (data) are available. This is different from the control-flow execution[2] in which the instructions are sequentially ordered using a program counter. Pioneer works in the field of dataflow computing, which date from the early 1970s, relied on fine-grained expression of the inherent program parallelism. The overhead incurred in the execution of fine-grained dataflow program is higher when compared to the control-flow counterpart. That was the main reason why the interest in dataflow computing models fell down. A renewed interest in dataflow programming has appeared due to the scarcity of advancements in the conventional parallel processing field. Instead of using pure,

[2]The control-flow execution is the conventional execution model used in the majority of nowadays computers.

fine-grain dataflow programs, coarser-grained dataflow models are studied. A hybridization between the dataflow computing and the control-flow one is produced. The smallest dataflow program bricks are full scaled tasks, whose instructions are sequenced using the conventional control-flow paradigm.

A leading work on dataflow programming models was introduced by Lee and Parks [75]. In the sequel we base our discussion on this work. Alike the Kahn process networks (KPN) [64] the dataflow process networks exhibits some sort of determinism. More particularly, for a given input data the computation results produced by the network do not depend on the execution order of processes, on nondeterministic execution times or on other kind of execution hazards. This property is primordial in parallel computing.

A dataflow model of computation is simply a set of *actors*, or equivalently *tasks*. The actors are communicating through unbounded unidirectional first-in-first-out (FIFO) channels, *and exclusively through these channels*. A dataflow network can be represented as a directed graph which nodes represent the tasks and which edges represent the communication channels. Traditional concepts such as "variable" or "memory"are missing[3]. The data that passes through the FIFO channels is quantized into *tokens*. A data token is indivisible and represents the smallest quantum of data that traverses the channels. The number of tokens in a channel is potentially infinite. A *stream* denotes the flow of tokens on a channel. We shall note that the dataflow model of computation is only an abstract model that does not constrain at all the implementation on a real platform. For example in a static execution model, the communication channels must be actually dimensioned if the execution of the dataflow network is envisaged.

An actor is executed, or *fired*, when all its input channels contain the required quantity of data tokens. The data tokens on the input channels are consumed by the actor and result tokens are produced. The result tokens are then written to the output channels and subsequently are yielded to next actors. The synchronization between the tasks is done implicitly via the data which traverse the FIFO channels. The number of produced and consumed data tokens together with their types is called the *signature* of an actor. An actor can have more than one signature. Two types of actors are distinguished: *static* actors and *dynamic* actors. A static actor has one or more signatures. A dynamic actor has at least two signatures, the choice of signature is data-dependent. The difference between static and dynamic actors is that the next signature to use for a static actor is predictable whilst for a dynamic actor it is not. As we will see in the next paragraphs according to the restrictions that are imposed to the signatures of the actors a multitude of dataflow computing models are defined. We orient interested readers to [60, 43, 86, 73] for an in-depth description of dataflow models of

[3]The actors of a dataflow application are simple control-flow programs for which the notions of variable and memory are defined. The absence of these concepts at global level in dataflow applications enforces its deterministic behavior.

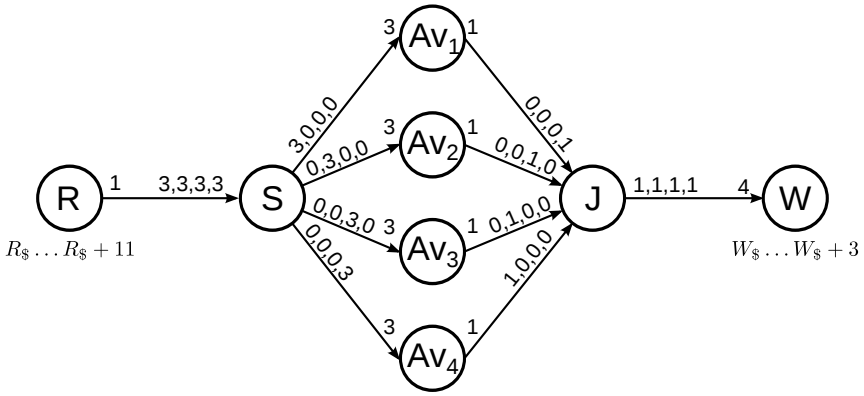


Figure 1.2: A CSDF application example. The application calculates the average of elements situated on a line for a 4×3 matrix (stored in the memory as a contiguous range). The resulting 4-element vector is returned reversed.

computation.

The most representative dataflow models are: *synchronous dataflow* (SDF) [74], *cyclo-static dataflow* (CSDF) [9] and *boolean dataflow* (BDF) [15]. In the SDF model all the tasks are static. Their signatures are unique and do not change during the execution. The quantity of produced and consumed data on the output and input FIFO channels is the same between two consecutive task firings. The CSDF model authorizes static actors with a single or a finite list of signatures. For actors having more than one signature, at each actor firing the next signature from the list is employed. After the last signature was used this process restarts and the first signature is reused. The number of produced and consumed data tokens changes cyclically. For an example of CSDF application refer to Figure 1.2. The values situated at the ends of each arc represent the actor's signatures. Actor **J** has 4 signatures: in the first execution it consumes one token on input channel 4 (the lowest one) and forwards it to the output channel, the same happens for the next 3 input channels and then the process restarts.

The BDF model is a generalization of the SDF in which data control execution is envisaged. Static and dynamic actors are used in the BDF model. Dynamic actors have two signatures. The choice between the two signatures is made in function of a boolean token consumed on an input channel. The BDF model can be easily generalized so that dynamic actors with more than two signatures are defined. The choice of the signature being made by an integer data token[4].

The first two models described above (SDF and CSDF) exclude any form of nondeterminism. That is why several fundamental properties, such as deadlock

[4]The integer-valued control dataflow model (IDF) was introduced in [14]. Besides the generalization of boolean controlled actors it includes several definitions which permit to express iterative constructs easier.

freeness or whether the memory required by the FIFO channels is bounded, are decidable for them. In contrast, these properties of dataflow networks are not decidable for BDF models. Nevertheless, approximate analysis of BDF networks is possible [98]. The big advantage of BDF model over SDF or CSDF is its Turing completeness[5]. The last fact implies an increased expressive power of the BDF networks over SDF or CSDF.

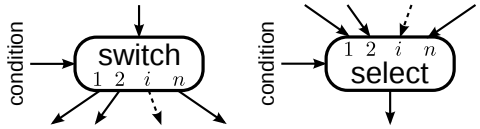
In what follows we introduce more formal definitions for the actors of a BDF programming language. Our further discussion will be grounded in this programming model.

In the context of real parallel processors several *system* actors, mainly for data access, data reorganization and conditional execution, should be defined. For more detailed information refer to [49, 50, 65]. Two system agents for reading and writing data are introduced. The actors take a range of memory addresses as parameter. At each actor firing, a memory location from the range is read/written in a round-robin fashion. The next class of actors are the data reorganization ones: SPLIT, JOIN and DUP. They permit to partition and assemble streams of data. Without going into too much detail, SPLIT actor partition the data stream, JOIN actor assembles several data streams, DUP duplicates input data stream into several output streams. In the application from Figure 1.2 the following actors are used: **R, W** - data read and write, **S, J** - split and join, **Av$_i$** - computes the average of 3 data tokens. Underneath the data access actors are given the memory ranges which they point to ($R_\$$ and $W_\$$ are fictitious). The SPLIT actor **S** partitions the matrix stream into lines while the JOIN **J** assembles 4 separate values into a stream.

The data dependent execution is introduced by a pair of actors: SWITCH and SELECT. In function of the used control value type SWITCH and SELECT actors can be divided into boolean-value (as in BDF) and integer-value control actors (as in IDF). The integer-value control actor is a generalization of the boolean-value control one. In what follows, we are not going to make a difference between these two types of control actors. Figure 1.3 illustrates the SWITCH actor and the SELECT actor. A SWITCH control actor is a dynamic actor with one data input, one control input and n, $n \geq 2$, output channels. A token from the data input channel is forwarded to one of the output channels in function of the control input token value. A SELECT actor has n data inputs, one control and one data output channels. Equivalently to the SWITCH actor, the SELECT actor forwards the data from one of its input channel to the output channel in function of the control actor token. In what follows we are going to denote by *branch* an output/input channel of respectively SWITCH/SELECT actor.

A *branching structure* is a traditional conditional schema made up of a SWITCH actor and a SELECT actor. Refer to Figure 1.4 for an illustration of a branching

[5]The Turing completeness means that a BDF network can implement an universal Turing machine.



(a) SWITCH actor. (b) SELECT actor.

Figure 1.3: Data controlled actors.

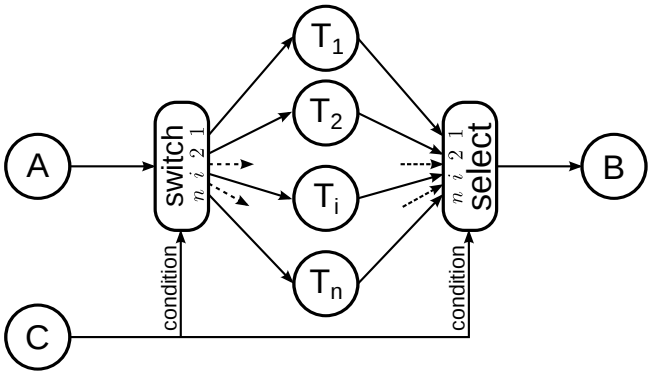


Figure 1.4: A branching structure with n branches. The actor to execute is chosen in function of the control token generated by actor **C**.

structure with n branches, also called an n-way branching structure. This schema allows data dependent execution of an actor from a set of n actors $\mathbf{T_1}$, $\mathbf{T_2}$,..., $\mathbf{T_n}$. The branch to execute is chosen in function of the value of the control token generated by **C**. To be more precise, suppose that the task **C** generates a data token with value i. This token is send to SWITCH and SELECT actors. The SWITCH actor forwards the data generated by actor **A** to the task $\mathbf{T_i}$, which at its turn consumes this data and sends the result to the SELECT actor. The SELECT actor forwards the data present on input i to its output channel.

1.1.3 Execution model

The dataflow graphs, as defined in the previous section, allow to represent parallel applications. This model of computation specifies only application constraints but does not define how to execute the application on a embedded hardware platform. An *execution model* is a formal specification for a dataflow graph execution technique on hardware platforms. It includes several steps: dataflow graph optimization, consideration of repetitive execution and hardware constraints, scheduling.

The optimization of the dataflow graph is needed in order to reduce the complexity of the graph caused by programmer-oriented (abstract) constructs (SPLITs, JOINs, etc.). In Figure 1.5 is depicted an example of a possible optimiza-

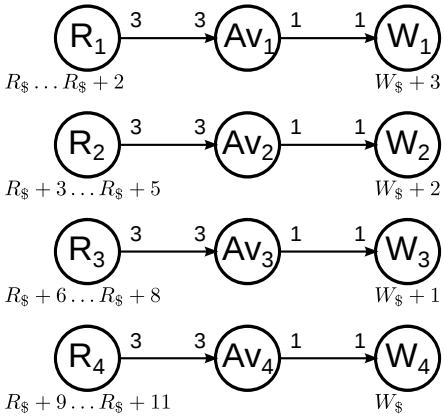


Figure 1.5: Optimized version of the dataflow program presented in Figure 1.2.

tion for the CSDF graph example from the previous section. We can see that the data distribution actors, **S** and **J**, together with data read/write ones, **R** and **W**, in the initial graph are replaced by direct read/write actors $\mathbf{R_i}$ and $\mathbf{W_i}$. This optimization avoids an useless copy of the data streams in the SPLIT and JOIN actors.

Usually in the field of embedded computing, the dataflow graphs are cyclically executed on unbounded streams of data. In a cyclic execution of the original dataflow graph the actors are executed periodically. Each execution of an actor is denoted as *instance* of the actor. The execution of actors is not preemptive, thus once started the execution must continue without interruption until it is finished. Following the terminology from the dataflow literature [74], an acyclic precedence graph (APG) is a graph (potentially infinite) composed of the actor instances. The edges of APG correspond to precedence relations from the initial dataflow graph and to precedence relations between the instances of actors (instances of an actor must be ordered). A simple dataflow graph together with its APG representation are depicted on Figure 1.6. An *iteration* of a dataflow graph corresponds to the number of times each actor should execute such that the system comes back to its initial state (in terms of token number on each edge). In the example, an iteration consists of two executions of actor **A** and three executions of actor **B**. We can observe that the number of actor instances does not correspond to the number of dataflow graph executions. The last is caused by the non-homogeneity in production/consumption rates of actors.

Another example is illustrated in Figure 1.7. The APG corresponds to the upper pipeline of actors from the example in Figure 1.5. As in previous example the numbers in superscript after the actor name differentiate the instances. The precedence relations between the instances of the same task imposes an order over the instance execution. If in the initial pipeline there was no parallelism, in the APG data parallelism is present. For example up to 3 instances of actor $\mathbf{Av_1}$ can be executed in parallel on different streams of data.

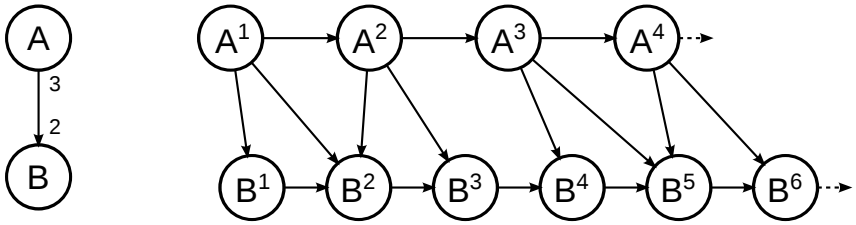


Figure 1.6: A simple dataflow graph (on the left-hand side) and the corresponding APG (on the right-hand side). In superscript actor instances are numbered. Unit production/consumption rates are omitted.

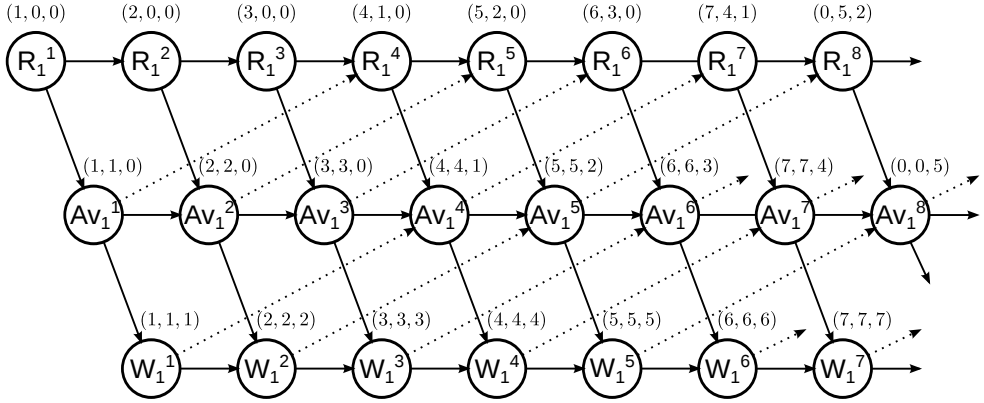


Figure 1.7: The APG of one pipeline of 3 actors from the example in Figure 1.5.

In the APG model only precedence relations between the actors and actor instances are considered. When the execution of a dataflow program on a hardware target is envisaged special attention should be paid to the dimensioning of inter-actor communication buffers. The size of a communication channel can be dimensioned by restricting the number of concurrent actor instances. In the APG from Figure 1.7 the buffer dimensioning problem is considered. Suppose that the buffer size between the actors $\mathbf{R_1}$ and $\mathbf{Av_1}$ have to be limited to 9 tokens. Knowing that $\mathbf{R_1}$ produces 3 tokens per firing, the number of concurrent instances of $\mathbf{R_1}$ should be limited to 3. This is accomplished by adding additional precedence relations between the instance i of $\mathbf{Av_1}$ and the instance $i+3$ of $\mathbf{R_1}$ (dotted lines in the example). By introducing additional precedence relations the buffer size is bounded (no more than 3 productions will occur before a consumption will take place) but the data parallelism is reduced.

Parallel processor scheduling [97, 55] consists in *allocating* tasks to processors and *ordering* their execution on each processor in order to minimize an objective function (e.g. schedule length or makespan, mean flow time, throughput etc.). The two steps can be done either at run time - *dynamic* scheduling, at compile time - *static* scheduling or both. The execution of a dynamic schedule is more flexible, but a larger overhead is introduced as scheduling decisions (allocation

and ordering) must be done at run time. Whereas in a fully static schedule data dependent execution or variability in the execution times cannot be easily considered. Compromise solutions exist where the task allocation is done at compile time and the ordering is done at run time.

In case of hierarchical parallel processors the scheduling problem becomes hierarchical too and is defined for each level of hierarchy. In the clustered massively multi-core processor introduced in Subsection 1.1.1 the scheduling problem is defined at system level and at cluster level. At system level the tasks are allocated to clusters. The set of tasks allocated to a cluster together with the precedence relations between these tasks is an independent scheduling problem at cluster level. In order to assure an acceptable level of flexibility and to provide a high-performance execution the task are statically allocated to clusters and the cluster scheduling is entrusted to a run time scheduler.

The dynamic scheduling at cluster level does not necessary mean bad, or, far from optimal schedules. Additional precedence relations can be added in order to constrain the possible schedules a dynamic scheduler can obtain. Thus, near optimal or optimal schedules can be favored. Such a method of over-constraining the tasks precedence relations can also be used to assure other interesting properties of program execution. A first example will be the dimensioning of communication buffers described earlier, which is achieved by adding precedence relations. Other examples would be data reuse maximization and hiding memory accesses with task executions (data prefetching).

In the dataflow model of computation the communication between the tasks is done through channels (modeled as FIFO buffers). When two communicating tasks are allocated to the same cluster a direct implementation of the communication channel is possible, e.g. in the cluster shared memory a space is reserved for each communication channel. More attention is required for the case when two communicating tasks are allocated to different clusters. The communication is implemented as packet-based data transfers via the NoC. For the sake of genericity, these types of communication links are replaced by couples of dummy data read and data write tasks. Distinct interpretation of communication links is excluded, thus the same mechanism of communication is preserved regardless the task allocation. The dataflow graph obtained after the addition of dummy read/write tasks has at least one connected component for each cluster. The APG representation also contains at least one connected component per cluster.

Direct implementation on a hardware platform of the APG representation of a dataflow graph is not possible[6]. One should define a mechanism that encodes the APG in a convenient manner for the run time task scheduler. The scheduler will then track the completion of tasks, detect enabled ones and schedule them for execution. A methodology for partially ordering tasks on a distributed system are *vector clocks* [81, 36]. This mechanism allows to isomorphically encode task

[6]A direct implementation will need to memorize and work with a potentially huge graph.

precedence relations from the APG. More information about the implementation of vector clocks we use is given in the technical report [99] and the patent [100].

An example of vector clock valuation is given in Figure 1.7 (see the vectors above the instances). The fourth instance of actor $\mathbf{R_1}$ is preceded by the first instance of $\mathbf{Av_1}$ and the third instance of $\mathbf{R_1}$ which is well reflected by the corresponding vector clock values $(1,1,0) < (4,1,0)$ and $(3,0,0) < (4,1,0)$. Whilst the second instance of actor $\mathbf{R_1}$ can be executed in parallel with the first instance of $\mathbf{Av_1}$ so we have $(2,0,0) \parallel (1,1,0)$.

1.2 Memory access optimization techniques

1.2.1 Caching

In the last decades, the performance of processors in computing systems has drastically increased. However, this cannot be said about the performance of the memory sub-system which has increased at lower rates. A performance gap between the memory sub-system and the computing one appeared. The processors were not able to deliver their full computing power because of memory stalls. A way for reducing memory stalls consists in using data *caching* techniques. The utility of caching resides in an empirically observed fact which states that for an application only 10% of the data is accessed 90% of times [57]. That is to say, it exists a *locality* in accessing memory data by an application. The data access locality can be divided in two types: *temporal locality* when a referenced object will be referenced again soon and *spatial locality* when neighboring objects of a referenced object will be accessed soon.

The memory sub-system of modern processors includes several levels of cache. In the memory hierarchy the size of cache memories is inversely proportional to their access speeds. For example in the clustered processor architecture illustrated in Figure 1.1, the processor cache has a smaller size than the cluster memory, but the data can be retrieved faster from the cache memory than from the cluster memory. As stated earlier, the aim of employing cache memories is for saving a copy of the most frequently used memory data for later reuse, thus potential memory stalls are avoided.

Prior to the existence of caching techniques the execution flow in a processor could be schematically represented as in Figure 1.8a. Initially, the application executed on the processor establishes a request for memory data with references r_1 and r_2. Then, the memory sub-system loads the data from the external memory to the processor (memory line in the illustration). In the meantime the processor is stalled and no useful work is done. After some time, the processor requests again memory data with references r_1, r_2, and other new memory references r_3, r_4. It can be easily seen that if the memory references r_1 and r_2 would have been stored "somewhere" for future use, the next time the processor would have

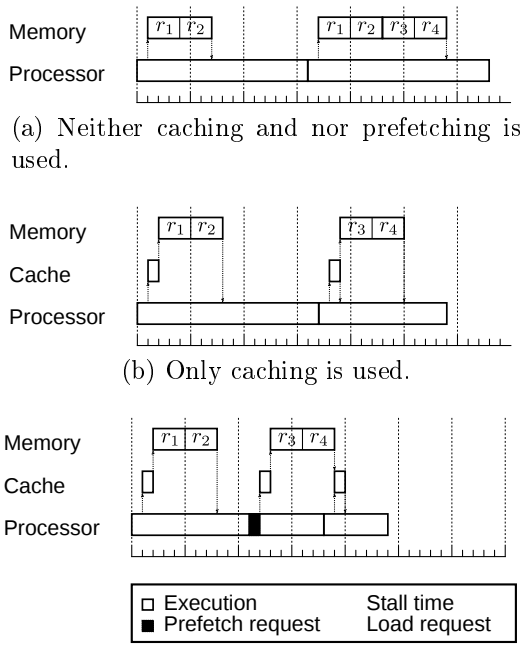


Figure 1.8: Gain in performance due to caching and prefetching techniques.

requested them the data loading process would have been faster. That is precisely the goal of the cache memory to be the place where to store memory data for future reuses. The study of hierarchical memory systems together with the benefit of caching schemes date from the early sixties [68, 108].

During the execution of an application when the processor needs data from the memory it establishes a *load request*, see Figure 1.8b for an illustration. When a load request is generated by the processor the memory sub-system verifies whether the requested data is present in the cache (for multi-level caches this process is repeated for each level of cache). If the data is present, then it is directly forwarded to the processor, if not, a *cache miss* is generated. At each cache miss the memory sub-system fetches (loads) data from the main memory into the cache and subsequently forwards it to the processor. In case there is not enough space in the cache for the new data, the memory sub-system has to evict some entries. What particular entry to evict is decided using a *cache replacement policy*. Some well know replacement heuristics are: least recently used (LRU) when the most early used entry is chosen, first in first out (FIFO) when the entry that has entered the first into the cache is chosen and randomly choose what entry to evict.

In some conditions the caching technique always generates cache misses. The *compulsory misses* happen when the requested data is accessed for the first time, since only the previously accessed data are stored in the cache. *Capacity misses* happen when the requested data was previously accessed but it was evicted from

the cache because of the application of the cache replacement policy. The data must be brought again from the main memory to the cache involving full memory access latency.

1.2.2 Prefetching

A further evolution of memory access optimization is done via the utilization of data *prefetching* technique, which is aimed for minimizing the number of compulsory and capacity misses. Data prefetching is the process of loading data from the main memory to the cache memory (for multi level caches, from an upper level of the memory hierarchy to a lower one) before these data are actually needed. Subsequently, when the processor will need these data it will use them directly from the cache. The main memory accesses are overlapped with computations thus the overall execution time is minimized. Early studies on cache memories [4] revealed the benefit of prefetching memory data before their explicit reference.

The data is prefetched upon a *prefetch request*, refer to Figure 1.8c for an illustration. In fact, the only difference between a memory load request and a prefetch request is that in the latter case a memory load request is issued in advance of the actual reference to the memory by the processor. The prefetch request is treated by the memory subsystem in the same way as a load request, thus by bringing into the cache memory the requested data but without direct processor involvement. In the best case, the prefetched data will arrive just in time for the processor to access it. Other, less favorable situations are if the data arrives too late the processor is stalled for some time, or, if the data arrives too soon it can be evicted from the cache by the replacement policy. A drawback of the prefetching is that unnecessary cache misses could be created by the prefetched data which replace cache entries that should be used before the prefetched data itself is used. This process is called *cache pollution*.

A more aggressive approach to prefetching is the *speculative prefetching*. The speculative prefetching is the process of loading data from the main memory to the cache memory (equivalently, from higher levels of memory hierarchy to lower ones) before it is even known that this data will be needed.

The prefetch request is established either by the application itself or by a dedicated hardware module in the processing system which has the function of generating prefetch requests. The prefetch is called *software prefetching* if it is generated by the application in execution or *hardware prefetching* if it is generated by the dedicated hardware module. Furthermore, *hybrid prefetching* techniques exist where the prefetch requests are established by the executed application in conjunction with dedicated hardware modules.

In software prefetch schemes an overhead is created because of required implicit memory load instructions. Contrary to software prefetch there is no overhead when using hardware prefetch techniques because no implicit instructions

are needed. Although no overhead is created in hardware prefetching schemes, usually they generate more *unnecessary prefetches* because hardware techniques infer future memory accesses without having any compile-time information. If the speculations on what to prefetch are incorrect, useless data are loaded into the cache memory. This does not create any problem for the execution of the application except the fact that the memory bandwidth is unnecessarily augmented and, potentially, the cache is polluted.

The use of a prefetching scheme in computing systems increases the bandwidth to the external memory, which in certain cases cannot be neglected. Sure that by minimizing processor stall time, the frequency rate at which the external memory locations are accessed by the application increases. The same amount of data must be loaded into the cache memory in a smaller time frame. The augmentation of external memory bandwidth must be supported by the computing system, i.e. the memory sub-system must be designed to cope with this higher bandwidth to avoid becoming saturated and make the prefetch worthless. A great attention must be taken in parallel computing systems where the memory bus utilization is typically higher than in sequential ones.

The concept of prefetching is not limited to processor memory system, it can be found also in the client-server paradigm [104, 54]. The server which is replying to requests from the client can anticipate future requests and perform a pre-computation of the replies to these requests.

Apart from prefetch optimization, other approaches that increase the performance of a computing system exist: increase the width of memory access bus (fetch 4 bytes per cycle instead of one), interlaced memories, etc. The increase of the cache memory block size will reduce the number of cache misses in programs with a high spatial data locality. The disadvantages of this technique, compared to prefetching mechanisms based on data locality, will be discussed in the next section. An interesting approach is to modify the memory placement of program data in order to augment the spatial and/or temporal data access locality [32, 109, 82]. We can also use alternative memory schemes [80] or augment the number of cache levels in the memory hierarchy. In multiprocessor systems memory bandwidth can be saved by reducing the level of consistency between different processors.

In what follows an overview of existing prefetch strategies from literature are described and it drawsmainly from the works [106, 16]. We mostly restrict our discussion to data prefetching schemes.

1.2.2.1 Software prefetching

In the majority of modern processors some kind of *fetch* instruction is supported. The fetch is a non-blocking memory load instruction, thus the processor is not stalled during the time the memory sub-system brings the data into the cache. The hardware required to implement the fetch instruction is modest when com-

pared to other prefetch approaches. The main difficulty consists in the smart insertion of fetch instructions within the program code. The last problem is known as *prefetch scheduling*.

The insertion of fetch instructions can be done either by the programmer or by the compiler during an optimization phase. In comparison to other optimizations that occur too frequently or that are too annoying to be done by hand the prefetch optimization can be done directly by the programmer. In practice it was observed that very few fetch instructions placed at strategic positions can have a huge impact on program performance.

Usually, software prefetching is applied on program loops for large array computations because a large amount of memory data is accessed with regular memory access patterns. This kind of computations are often used in scientific programs. The low data locality in such loops makes useless the caching techniques. In contrast, the memory access pattern can be easily found in these types of codes making the prefetch an useful optimization. Schematically, fetch instructions are placed inside loop bodies in order to prefetch data that will be accessed in future iterations. In this way the memory access latency for future iteration data is covered by the current iteration code execution. A further optimization can be done by using a *prologue* code block, in which, prior to the loop execution, the first data from the arrays are prefetched in order to avoid cache misses at the beginning of the loop execution.

Another problem that should be considered is whether executing a fetch instruction one iteration ahead of the actual data use is sufficient for hiding memory access latency. Early studies considered that one iteration is sufficient [18]. However, this is not always true when considering loops with small computational bodies. It may be necessary to prefetch the future data several iterations before the data is referenced. This number is known as *prefetch distance* and is expressed in number of loop iterations. It can be approximated as the ratio between the memory access latency to the execution time of the loop body. By using smallest or largest execution time the prefetch can be done more or less conservative.

The loop prefetching technique described above can be easily integrated in a compiler chain. In [85] such kind of automatic prefetch technique, which inserts fetch instructions, is described. Different works [8, 95] measured the speed up brought by such an optimization for different processors (PowerPC 601 and HP PA-8000 based computing systems). In the majority of benchmarks the prefetching improved the run-times, although for some benchmarks the applications were slowed down. The loss in performance was due to prefetch instruction overhead.

The main drawback of automatic loop prefetching is that the memory access pattern must be reliably predicted, thus array access indices must be linear functions of array indices. This is generally true for scientific codes as said earlier, but far less for general programs. Several attempts to generalize this prefetch optimization for general programs were made [29, 76, 79], but the results were

not good enough. The main cause being the irregular memory access pattern. In the case of algorithms working on data structures as graphs or linked lists there is a little chance that successive memory blocks will be accessed. Even more, often general applications have a high degree of temporal data access locality resulting in a high cache utilization, thus the potential benefit of prefetching is reduced.

The main issues to consider in a software prefetching scheme is the overhead introduced by the fetch instructions, the significant code expansion which negatively affects the instruction cache performance and the last but not least the fact that software prefetching is done statically, thus it is unable to detect if the prefetched block is still in the cache memory (and eventually re-fetching it if it was evicted from the memory by the cache replacement algorithm).

1.2.2.2 Hardware prefetching

Hardware prefetching schemes introduce a prefetch possibility without programmer or compiler involvement. The existing program codes and executables do not need any change because hardware prefetching is done without explicit fetch instructions (the overhead due to fetch instructions is entirely eliminated). In contrast to software approaches which are normally applied to prefetch data, hardware schemes can be used to prefetch data as well as program instructions. Also hardware prefetching schemes can take advantage of program run-time information, thus a hardware prefetching is potentially more effective.

Sequential prefetching A sequential prefetching scheme prefetches memory blocks which are next to the currently referenced block. In a caching scheme multiple memory words are grouped into a single block in order to take advantage of spatial data access locality. Thus data caching can be seen as an implicit prefetch of data that are likely to be accessed in the near future. One can use larger memory block sizes in order to increase the implicit data prefetch, which is not always beneficial. With the increase of cache block size, the amount of potentially useful data evicted from the cache (by the cache replacement technique) increases too. This is why in certain cases the further increase of cache block size results in performance loss. The limitations of the implicit prefetching (data caching) are prevented by using smaller cache block sizes along with a sequential prefetching method.

The simplest sequential prefetching scheme is the *one block lookahead (OBL)* approach. It acts as follows: when a memory block with reference r is accessed the next block, with reference $r+1$, is prefetched. This sequential prefetching differs from simply doubling memory block sizes, because the prefetched blocks are treated separately by the cache replacement policy. Potentially, more frequently referenced data can be stored in two smaller cache blocks than in a single larger one.

Several OBL implementations exist [101]. They are different on what type of access to block r initiates a prefetch of block $r+1$. The most interesting approach is the *tagged prefetch*. In this method a tag bit is associated with each block. This bit is used to track the blocks that were demand fetched or the prefetched blocks which were referenced for the first time. In both of these cases the next block is prefetched. We shall note that in some OBL implementations a direction parameter is used in order to perform either a forward $(r+1)$ or a backward $(r-1)$ prefetch.

One drawback of the OBL prefetch appears in program loops with small bodies. The prefetch is not established early enough from the actual use in order to avoid processor stall times. To cope with this issue one can prefetch K sequential blocks instead of a single one, where K is the *degree of prefetching*. When a prefetched block r is accessed for the first time, the next $r+1, \ldots, r+K$ memory blocks which are not present in the cache are prefetched. It could be tempting to use large values for K, however the overhead (in terms of additional memory bus traffic and cache pollution) introduced by prefetching K blocks in program phases that have little spatial locality tends to make the prefetch gain negative for the whole program. In [89], it is stated that the overall gain in performance for values of K larger than one is nullified by the induced overhead on the memory bandwidth and because of cache pollution. We shall note that the prefetching schemes described above, one and K block lookahead, can be directly used to prefetch program instructions. Even more, their efficiency for code prefetching will be higher than for data prefetching as usually the subsequent program instructions are very often close to each other in memory.

A straightforward solution for the above performance issue is an *adaptive sequential prefetching* scheme introduced in [30]. In this approach, the value of the parameter K is changed in function of the spatial locality exhibited by the program at a particular instant of time. A prefetch *efficiency metric*, expressed as the ratio between the number of successful prefetches (i.e. number of prefetches resulting in a cache hit) to the total number of prefetched blocks, is periodically calculated during the execution of the program. Initially the value of parameter K is set to one. During the execution, if the efficiency metric goes over an upper threshold, the value of K is incremented and respectively if the efficiency metric drops below a lower threshold the value of K is decremented. If the value of K becomes zero, thus prefetch is disabled, the efficiency metric based on number of good prefetches is changed to a metric based on the frequency of cache misses to block $r+1$ when block r is in the cache. Experimental evaluations of the adaptive sequential prefetching method revealed some improvements over the K block lookahead method, although the overall performance gain was not as one would expect.

Another improvement which minimizes the cache pollution of K block lookahead prefetch scheme was introduced in [63]. It consists in adding a FIFO *stream*

buffer into which the K prefetched blocks are brought from the memory before being transfered to the cache. Whether the head position block of the stream buffer is accessed, it is brought into the cache memory. Afterwards, a new memory block is prefetched into the tail position. If the block accessed by the processor is not situated in the head position of the stream buffer, the buffer is flushed. Thus, in order to be effective the prefetched blocks must be referenced in the order they have been brought into the buffer. A further improvement suggested by the authors is to use several stream buffers that are acting in parallel.

Arbitrary stride prefetching The sequential prefetching schemes described above can be easily implemented with relatively simple hardware and provide a reasonable improvement of the execution time. The sequential prefetching for programs which use scalar references, array accesses with large strides or other non-sequential memory access patterns results in a lot of useless prefetches because these types of accesses do not expose enough spatial locality.

Several prefetching methods have been proposed to cope with strided memory accesses resulting from looping structures [5, 39, 28]. These methods are based on the comparison of successive referenced addresses of memory access instructions. The prefetch scheme from [28] computes at each iteration the difference between the addresses referenced by a memory access instruction in the current r_2 and in the previous r_1 iterations. If the difference, $\Theta = r_2 - r_1$, is not zero then Θ is assumed to be the memory stride of this memory instruction. During the loop execution, the prefetch scheme predicts that the data with address $r_2 + \Theta$ will be accessed during the next iteration and prefetches it. The prefetching continues in the same way until the prediction is no longer valid, that is the prefetched data is no longer the real referenced location.

In this approach, the previously referenced address and the last detected stride must be recorded for each memory access instruction. Storing this information for all program memory instructions is practically impossible. Instead, only the reference histories of the most recently used memory instructions is stored in a dedicated cache called *reference prediction table (RPT)*. For each entry of the RPT a state machine is used to track whether the predicted stride is correct or not, and also to initiate prefetches of the predicted data.

The prefetching scheme described above behaves better in the case of strided memory accesses when compared to sequential approaches. However it cannot manage loops with small bodies as the prefetch distance is limited to one iteration. A solution to this issue is the introduction of a *distance* field to RPT which specifies explicitly the prefetch distance. The next address to prefetch will be situated at $\Theta \cdot \mathit{distance}$ from the currently referenced address. A more sophisticated mechanism is needed to manage the update (increment/decrement) of the distance field. We shall note that these techniques can be adapted to multi-dimensionally strided memory access patterns.

A large part of existing prefetching schemes require regular memory access patterns in order to be efficient. Nevertheless some works discuss an extension of the RPT prefetching scheme for data structures linked by pointers (i.e. linked lists, graphs). A special mechanism detects indirect memory reference strides using a modified RTP table. An approach for prefetching data in programs that are using linked list data structures is discussed in [93]. But instead of extracting reference patterns for a single memory instruction this method detects dependencies between load instructions. In particular, the memory loaded values that serve as base addresses for subsequent loads are recorded and used for prefetching linked data structures.

It was observed that a cache miss to a given address is followed by another miss to a set of predictable addresses. Several studies proposed prefetching schemes that associate to the current cache miss address a set of likely subsequent miss addresses. The set of potential cache miss addresses is built and updated either using previous observations [2] or Markov predictors [62].

1.2.2.3 Hybrid prefetching

As we have seen earlier, the software prefetching schemes take advantage of compile time information in order to guide the prefetching. Whereas the hardware prefetching methods adapt their prefetch strategy at run-time in function of the executed code. Each method has its strengths and its weaknesses. The software approach introduces additional fetch instructions which increase the total execution time. In contrast, the hardware approach do not introduce any overhead as prefetch decisions are taken by a dedicated hardware at execution. The main disadvantage of hardware prefetching is that many useless prefetches are generated because the speculation on the next accesses is less informed. Many researchers came to the conclusion that a hybrid prefetching scheme, an approach which combines software and hardware techniques, will overcome the weakness of each method.

In the adaptive sequential prefetching, the degree of prefetching K is adapted at run-time in function of the observed prefetch efficiency. Much time and memory bandwidth is lost until the value of K becomes adapted to the currently executed code. The authors of work [47] propose to compute the value of K at compile time and to pass it to the prefetch hardware. A special fetch instruction that manages such behavior is introduced. Prefetching for non-sequential referencing is managed using normal fetch instructions. A similar approach was proposed in [27] with the exception that compile time prefetch instructions are given to a hardware strided prefetch engine, namely the RTP prefetching scheme. Entries (tag, address and stride) are inserted into the RTP table before program loops which can benefit from strided prefetch.

In [112] a hybrid prefetching scheme for irregular memory accesses is proposed. Memory locations are tagged in such a way that the reference to an object element

either initiates prefetches for other elements of this object or the object pointed by it. The compiler has the responsibility to tag memory objects and a hardware part (situated within the memory system) handles the prefetching.

1.2.2.4 Prefetching for parallel processors

The memory access sub-system is already a bottleneck which opposes the performance increase of sequential processors. The data access problem is more distinguishable in parallel processing systems because of many processors which compete for using the shared memory bus. Many of the prefetching schemes described in the previous sections can be directly applied in many-core processors. Although, a special attention should be paid to balancing the computing cores competition for memory bandwidth, assuring the coherence of the data between the multiple copies on different processors, balancing processors usage between prefetching (prediction, pre-execution) and computing. A multi-core processing systems has several points that differentiate it from a single-core processor:

- Typically, parallel programs are written with different programming paradigms, which offer additional information on the program parallelism and memory access patterns.
- The memory hierarchy of a multi-core processor is usually built up of more levels (layers) than a memory hierarchy in a single-core computing system. This provides different possibilities for choosing prefetch source and destination.
- The prefetching scheme in a multi-core processor can have a higher importance (when compared to single-core prefetching) in increasing the overall system performance, because a well managed prefetching tends to provide better memory bandwidth utilization.

In paper [38] the authors studied the performance improvement due to data prefetching mechanisms in vectorized parallel processor applications. In vectorized applications the stride information of memory accesses is explicitly available (as all the operations are done on vectors) and no special analysis (software or hardware) is needed to extract it. The information about memory access strides is encoded in vector references and is directly available for the prefetch hardware. Two prefetching schemes were used. They are variations of the K block lookahead approach with some modifications that consider the available information about memory access strides. Simulation results revealed that both prefetching strategies increase the performance of the computing system when compared to executions with disabled prefetching.

A prefetching approach that significantly differs from previous ones is described in [48]. In this paper, prefetch possibilities in a distributed memory

multiprocessor with global and local memories, interconnected by a network, are studied. Instead of prefetching one word (cache block) at a time, the data is prefetched in large blocks (i.e. many cache blocks per memory access) from the global memory to the local one. In this way a better bandwidth utilization over the interconnection network is obtained in comparison to having a single word transfer per memory access. Like in compiler driven software prefetching schemes, in this approach, the compiler inserts prefetch instructions in the program code around looping constructions. But rather than inserting fetch instructions for each word referenced within the loop body, larger blocks of data are prefetched before the loop is executed. In this way the overhead introduced to the program code is significantly smaller. The limitations of this prefetching scheme are: the referenced data must remain read-only between the prefetch and its use because no coherence mechanism is provided, also, the prefetching of conditional accesses is not done in order to avoid prefetching potentially a large amount of useless data or worse prefetching nonexistent one. Due to these limitations approximatively 42% of memory references could not be prefetched in the six benchmark programs the authors have used.

In the context of a shared-memory multiprocessor architecture the authors of [84] study two different prefetch approaches. In the first one, a *remote access cache (RAC)*, placed between the interconnection network and the shared memory, is used to store prefetched memory data blocks. In the second approach, the prefetched memory block is directly stored in processor cache. The use of a dedicated RAC cache for prefetched data was supposed to increase the overall system performance by separating prefetched data from demand-fetched data. Simulation experiments proved that the use of RAC did not offer the expected result, the speedup of prefetching directly into processor cache being higher.

With a large amount of computing power available in parallel processing systems, more complex software prefetching algorithms can be used. Several works [41, 78] proposed to pre-execute the application on a free processor (or during idle processor cycles in a single-core system) in order to predict future cache misses. Usually in these methods, a restricted version of the program is pre-executed, i.e. the conditional code is not executed, a prediction of code executed in future is done.

1.3 Research motivation

The management of memory accesses plays a crucial role in high-performance exploitation of the computing power offered by a parallel processor. Two memory access optimization techniques augment the execution performance of applications. Data reuse allows to reduce the number of times the same data is accessed, whereas data prefetching reduces the memory access time lag.

We place ourselves in the context of the execution of dataflow applications

on clustered massively multi-core processors (as that depicted in Figure 1.1). To each cluster an APG (as defined in 1.1.3) graph is associated. This graph is cyclically executed. The execution of an actor can start only after some of its data (including code, actor proprietary or input data) has been loaded from the external memory to the processor cache. The actor code is not mutable. Whereas the content of actor data can change in consecutive dataflow graph iterations. In what follows, we are not going to differentiate between the data and the code loading operations. By abuse of language we denote both operations as data loading.

All the memory communications must transit the cluster memory, i.e. the processors cannot directly access the external memory. The data is loaded either from the external memory to the cluster memory or from the cluster memory to the processor cache memory. Depending on the place where the memory access optimizations take place we differentiate *high-level* and *low-level* optimizations, thus we have high-level/low-level data reuse and data prefetching. The high-level optimization aims to optimize the data reuse/prefetch strategy at the level of the dataflow graph executed on a cluster. Whereas the low-level optimization is responsible for the data reuse/prefetch strategy at actor level. We suppose that the cluster memory is a kind of software-controlled scratchpad memory. The executed application decides what data to prefetch, store and evict from the cluster memory. When the cluster memory is sufficiently large to store the code and data of all actors it is less necessary to worry about memory access optimization. The data access operations are executed once and then the data is reused at each cyclic firing. However, if only a part of actors code and data fits in the cluster memory the access optimization becomes a crucial issue.

In the following subsections we describe several problems which aim to provide optimal (or nearly optimal) data access management strategies. In order to be certain that the data prefetch and the data reuse strategies found off-line are respected in an on-line execution environment (cluster run time scheduler) one should add new precedence constraints to the APG graph. An over-constrained APG, which follows the prefetch strategies, is obtained. In this way, the prefetch strategies would not interfere with existent compilation technique and execution model. Another solution will the the addition of a helper actor which will apply the given optimization strategy.

1.3.1 Data prefetching memory access optimization

The data prefetching strategies can be divided into two groups, *generic* prefetching is a general prefetching technique which is applied to any application and *specific* prefetching is a prefetching strategy which is tailored to each application. Specific prefetching is furthermore differentiated into *adaptive* and *adapted*. An adaptive prefetching updates its prefetching technique on the fly, i.e. at run

time (e.g. parameter estimation of a Markovian model). Whereas in an adapted prefetching only informations available at compile time are used. We can further divide the specific adapted prefetching into *a priori* when only intrinsic or statically obtained information about the application are used and *a posteriori* when additionally are included information (i.e. actor execution times, branch probabilities) captured during application profiling.

Traditional prefetch methods, found in the literature, can be used to load in advance actor data and code from the cluster memory to the processor cache. As the application of our work specifically targets prefetching at the dataflow level of abstraction, we do not consider low-level prefetching models. Furthermore, the latter models have been extensively addressed in the literature. Literature on prefetch techniques for dataflow graphs is scarce and limited to a few particular cases [107, 17, 87]. This is the main motivation of our research. In a compilation environment for dataflow applications, initially we build a prefetching strategy from application characteristics known at compile time (specific prefetching adapted *a priori*), and in the sequel, this prefetching strategy is refined with run time feedback information (i.e. actor execution times, branch probabilities) obtained after profiling the application execution (specific prefetching adapted *a posteriori*). Thus, our first research motivation is to investigate *the high-level, adapted prefetching management specific for each dataflow application.* This motivation is instantiated into two problems: (i) prefetching for dataflow applications and (ii) speculative prefetching for branching structures. In what follows we are going to briefly introduce these problems.

As stated earlier, the execution of an actor cannot start until all its data have been loaded from the external memory to the cluster memory. We consider that the actors are executed in two steps, operations: a data loading operation and an execution operation. The operations are not preemptive, thus once started they must finish without interruption. The data loading operations of a dataflow application must be executed sequentially as only one memory access channel is available between the external memory and the cluster shared scratchpad. Whilst the execution operations are performed in parallel on one of the available cluster computing cores. The execution operations are constrained by the precedence relations of the dataflow graph. The objective will be to find a schedule of data loading and execution operations which minimizes the total execution time. The problem defined in this way can be modeled as a *hybrid flow shop under precedence constraints*. This problem is discussed in details in Chapter 2.

The main limitation of the previous model is that the data-controlled actors are not considered. In the next optimization problem we are going to examine optimal prefetching strategies for branching structures separately. We recall that a branching structure is the programmatic construction that permits to execute one actor from a set of actors in function of a control input data. We suppose that the available prefetching time is a random variable, so that probabilistic execution

times could be taken into account. The goal of the speculative prefetching for branching structure problem, examined in Chapter 3, is to find optimal prefetch strategies for branching structures such that statistical objective functions are minimized. By prefetch strategy we mean either ratios of branch data (fractional strategy) to load or ordering of data loading operations (all-or-nothing strategy). Two objective are examined: expected execution time and worst-case execution time. As long as the branch to execute is unknown some data is loaded to the cluster memory following the prefetch strategy. When the branching structure will choose the actor to execute, some or all of the actor data will be already present in the memory of the cluster.

1.3.2 Data reuse memory access optimization

In a dataflow application the actors are using external memory data for their execution. Some of this data is common for several actors[7]. Loading several times the same data objects is redundant and engenders higher memory bandwidth, larger NoC traffic, ineffective utilization of the available memory, etc. For performance issues one should avoid as much as possible to load repeatedly the same data object.

The traditional data reuse optimization consists in the addition of a hierarchy of cache memories. The data objects, once fetched from the main memory, are stored in a special memory, or cache, for later reuse. If the same data object is accessed for a second time, soon enough, then potentially the execution time gains one memory access. In the clustered parallel architecture we study, data reuse can be done either at high-level or at low-level. The low-level data reuse refers to an actor execution, whilst the high-level data reuse refers to an entire dataflow graph. Many works which treat the data caching for sequential programs exist in the literature. The execution of an actor being equivalent to the execution of a sequential program we do not study the low-level aspect of the data reuse optimization. Few works on caching techniques for dataflow applications exists. One representative paper is [70] in which the authors describe cache aware scheduling for SDF applications. Our second research motivation is to propose *optimal high-level data reuse strategies for dataflow applications*. In the following paragraph we describe a problem intended to provide such data reuse strategies.

The task ordering and memory management problem is studied in Chapter 4. Like previously, the actors are executed in two steps: data loading and execution. In contrast, we suppose that the duration of data loading operations change in function of their order. The common data between the actors is stored in the cluster memory for later use. So the data loading durations of successive actors

[7] As soon as a dataflow application involves data parallelism some actors can share the same code and potentially some shared constants, dataflow implementations of the FFT algorithm are archetypal of this.

using the same data is potentially smaller. The objective of the task ordering and memory management problem is to find an execution order of data loading operations such that the data reuse is maximized. Because of a single memory access channel the data loading operations have to be serialized. We suppose that the number of available cores for actors executions is unlimited. Under this hypothesis only the data reuse aspect of the problem is considered. This is not necessarily a drawback, for example the found order of data loadings can be used to prioritize an on-line cluster scheduler.

Chapter 2

Prefetching for dataflow applications

A dataflow application executed on a massively multi-core processor is represented by a graph of task instances (in what follows, we omit the term task instances and use simply tasks) or APG (Acyclic Precedence Graph). The tasks are using external memory data. In the context of clustered architectures, before the execution of a task can start all the data on which it depends must be loaded from the external memory to the cluster memory. This representation of dataflow application scheduling allows to naturally model data prefetching, because the memory access operations are separated from task executions and can start in advance.

The work of this chapter will appear in *Computers & Operations Research* [26].

Hereafter, we introduce a formal definition of a dataflow application scheduling problem which optimizes the data prefetching. We model it as a two-stage hybrid flow shop (HFS) problem with precedence constraints and parallel machines at second stage. Two versions are examined, the classical HFS where idle time between the operations of the same job is allowed and the no-wait HFS where such idle time is not permitted. Each APG task can be viewed as two consecutive operations, the first one is the loading of the data used by the task from the external to the cluster memory and the second one is the task execution itself. Usually in a parallel computer the memory accesses are done sequentially, so only one data loading can be done at a time, whereas the execution of the tasks can be done concurrently on the available processors. Hence data loading corresponds to the first stage operation in the HFS problem, and task execution corresponds to the second stage operation. Second stage precedence relations between the operations are equivalent to the partial order of APG and reflects the internal data dependencies (amongst other dependencies). In order to limit data buffering, the execution of a task has to start when its data loading is finished, this corresponds

to the no-wait case of the HFS, whereas the classical HFS corresponds to the case when no space limit is imposed on data buffering.

The chapter is organized as follows. A formal definition and complexity result of the hybrid flow shop problem are given in Section 2.1, afterwards in Section 2.2 a brief description of flow shop scheduling related works is introduced. Two global lower bounds are proposed in Section 2.3. Section 2.4 presents a list scheduling heuristic, and, in Section 2.5 we describe a randomized version of this algorithm. In Section 2.6 the lower bounds and heuristics performances are compared using randomly generated instances and Section 2.7 concludes the chapter.

2.1 Problem formulation and complexity

This work considers the hybrid flow shop problem under precedence constraints. More precisely the two-stage hybrid flow shop $HF\,(1, P_m)$ with precedence constraints on the second stage is studied, by abuse of notation we denote it HFS in what follows. Assume a set of n jobs have to be processed in two stages. There is only one machine for the first stage and m identical parallel machines for the second stage. Each job $i \in \{1, \ldots, n\}$ consists of two operations: the first operation of duration $a_i > 0$ is executed on the first stage, and afterwards the second operation of duration $b_i > 0$ is executed on the second stage. No preemption is allowed in operation execution. The precedence constraints of the operations on the second stage are given by a directed acyclic graph $G = (V, E)$, where V represents the set of jobs and E gives the dependence relations between those jobs. There are no precedence constraints between the operations on the first stage.

The objective is to minimize the *maximum completion time* or *makespan*. Two different cases of HFS can be distinguished: the *no-wait* HFS when once a job has started it is executed on all the stages without being interrupted (the end time of first stage operation coincides with the start time of second stage operation) and the *classical* HFS when no such constraint is imposed. In the $\alpha \mid \beta \mid \gamma$ notation the flow shop problems we examine are $HF\,(1, P_m) \mid G_1 = \emptyset, G_2 = G \mid C_{max}$ and $HF\,(1, P_m) \mid G_1 = \emptyset, G_2 = G, no - wait \mid C_{max}$.

Despite that no precedence relations are defined for the first stage operations, the second stage constraints can be extended over the first stage because they are dominating the order in which the first stage operations are executed. This fact is obvious in the case of no-wait HFS. On the other hand in a classical HFS several orders of first stage operations can be defined for the same second stage schedule such that the solution value does not change. In what follows we consider that if a second stage operation must be executed after another second stage operation then the corresponding first stage operations must follow the same order.

The HFS problem described above is clearly $\mathcal{NP}$-hard. The proof is simple and inspired from [52]. Actually, if all the first stage operations have zero du-

rations, then we obtain a problem of minimizing the makespan on m parallel machines subject to precedence constraints. As the latter is known to be $\mathcal{NP}$-hard in the ordinary sense [105] we conclude that the HFS problem is $\mathcal{NP}$-hard too. We can furthermore prove that the HFS problem is strongly $\mathcal{NP}$-hard because its particular case, the hybrid flow shop problem with at least two machines at one stage, is strongly $\mathcal{NP}$-hard [58].

2.2 Related works

The literature on the hybrid flow shop problem under precedence constraints is quite scarce, even though lot of works exist on the hybrid flow shop and on the flow shop with precedence relations. For a review of the plentiful work on the hybrid flow shop problem we refer to [94, 91]. We shall note that most of the work is done for the general m-stage hybrid flow shop, nevertheless many authors tried to adapt the Johnson algorithm for the two-stage flow shop. A model close to ours, the two-stage hybrid flow shop with parallel machines at first stage only is studied in [53]. The authors determine the optimal ordering on the second stage given a scheduling of jobs on first stage and introduce some interesting lower bound concepts.

Although less represented in the literature, the flow shop problem under precedence constraints is quite well studied. In [44] the authors provide a classification of two and three machine flow shop problems under machine-dependent precedence constraints. Different models of shop scheduling problems with precedence constraints are considered in [103]. In their study the authors introduce two types of precedence constraints and provide complexity results and some polynomial time algorithms for shop scheduling models. The authors of [51] propose to reduce the job shop problem to a flow shop problem under precedence constraints, and introduce several modified flow shop heuristics for solving the flow shop problem constrained by precedence relations.

The hybrid flow shop problem under precedence constraints is studied in a few papers [34, 11, 12, 94], from an applicative point of view. In the studies mentioned above some heuristics are proposed. The authors are using stage-independent precedence relations between the jobs and different optimization criteria.

2.3 Lower bounds

Without loss of generality we suppose, in what follows, that the digraph $G = (V, E)$ describing the precedence relations between the operations on the second stage contains one source vertex, denoted 0, and one sink vertex, denoted $*$, with zero processing times. Also we suppose that the number of jobs is greater than the number of available second stage machines, $n > m$.

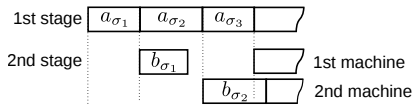


Figure 2.1: Second stage idle time needed to execute first stage operations (in this example the total idle time equals to $(a_{\sigma_1} + a_{\sigma_2} + a_{\sigma_3} - b_{\sigma_1}) + (a_{\sigma_1} + a_{\sigma_2})$).

2.3.1 Global lower bound 1

Some concepts of the following lower bound were introduced in [52] for the hybrid flow shop problem. We have adapted it in order to take advantage of second stage precedence relations.

$$GLB1 = \max\left(GLB1^1, GLB1^2\right)$$

In the first part $GLB1^1$ of the bound we take into account that there is inevitably an idle time on the second stage machines during the execution of the first $m+1$ jobs. During this idle time the first stage operations of the respective jobs are executed (see Figure 2.1 for an illustration).

Let $\sigma_1, \ldots, \sigma_{m+1}$ be the ordering of the first executed $m+1$ jobs on the first stage, σ_i represents the job on position i. For any precedence constraint between two jobs i, j, thus any edge $(i,j) \in E$ of graph G, if both jobs i, j belong to the ordering then relation $\sigma_i^{-1} < \sigma_j^{-1}$ must be verified (σ_i^{-1} is the position of job i). The precedence relations can be rephrased as: operation σ_1 has to be a successor of the source node 0 such that σ_1 has only one predecessor (which is the source node itself), operation σ_k must verify $\text{pred}\,(\sigma_k) \subseteq \{0, \sigma_1, \ldots, \sigma_{k-1}\}$, and so on. Hereafter, $\text{succ}\,(i_1, \ldots, i_k)$, $\text{pred}\,(i_1, \ldots, i_k)$, represents the union of successors, respectively predecessors, of vertices $i_1, \ldots, i_k$ in the graph G.

The idle time on the second stage machine where job σ_k is executed, is at least $\sum_{i=1}^{k} a_{\sigma_i} + \max\left(\sum_{i=k+1}^{m+1} a_{\sigma_i} - b_{\sigma_k}, 0\right)$. For the ordering $\sigma_1, \ldots, \sigma_{m+1}$ the total second stage idle time is:

$$Z_1 = \sum_{k=1}^{m} \left(\sum_{i=1}^{k} a_{\sigma_i} + \max\left(\sum_{i=k+1}^{m+1} a_{\sigma_i} - b_{\sigma_k}, 0 \right) \right)$$

The sum between the minimum possible idle time Z_1 and the total amount of second stage jobs duration divided by the number of available second stage machines gives a lower bound on the execution time. As all processing times are integers the lower bound should have also an integer value, a ceiling operator $\lceil\ \rceil$ is used for this purpose:

$$GLB1^1 = \left\lceil \frac{1}{m} \left(Z_1 + \sum_{i=1}^{n} b_i \right) \right\rceil$$

In order to find the sequence $\sigma_1, \ldots, \sigma_{m+1}$ which satisfies the precedence constraints and minimizes Z_1, the following combinatorial problem must be solved:

$$\begin{aligned} Z_1 = \text{Minimize} \quad & \sum_{k=1}^{m} \left(\sum_{i=1}^{k} a_{\sigma_i} + \max \left(\sum_{i=k+1}^{m+1} a_{\sigma_i} - b_{\sigma_k}, 0 \right) \right) \\ \text{s.t.} \quad & \text{pred}\,(\sigma_k) \subseteq \{0, \sigma_1, \ldots, \sigma_{k-1}\} \end{aligned}$$

The following relaxation makes this problem solvable in polynomial time (here relation $\text{anc}\,(l)$ gives the ancestor vertices of vertex l):

$$\begin{aligned} Z_1' = & \underset{\sigma_k^1}{\text{Minimize}} \sum_{k=1}^{m} a_{\sigma_k^1} (m - k + 1) \\ & + \underset{\sigma_k^2}{\text{Minimize}} \sum_{k=1}^{m} \max \left(\sum_{i=k+1}^{m+1} a_{\sigma_k^1} - b_{\sigma_k^2}, 0 \right) \\ \text{s.t.} \quad & \left| \text{anc}\left(\sigma_k^l\right) \right| \leq k,\ l = 1, 2 \end{aligned}$$

The relaxation consists in minimizing the two parts of the objective function separately. Firstly, an ordering σ^1 that minimizes the left side of Z_1' and afterwards a new ordering σ^2 which minimizes the right side of objective, should be found. The solution of the relaxed problem can be used for lower bound calculation in place of the initial problem solution because $Z_1' \leq Z_1$. Algorithm 2.1 finds the solution Z_1' of the relaxed problem. We shall note that in our experiments, we have obtained a deviation between the optimal global lower bound (calculated using Z_1) and the relaxed version (calculated using Z_1') less than 0.2%. This fact indicates that there is no much benefit from using the optimal calculation for Z_1 when compared to relaxed computation Z_1', especially that in the majority of cases (>75%) the same solution is found.

The second part $GLB1^2$ of the bound, takes into consideration the fact that the execution cannot finish before all the operations on the first stage are processed. Additionally, in the best case, the last operations executed on the second stage are those that are predecessors of the sink node and have minimal processing times. Refer to Figure 2.2 for an illustration of such configuration.

Let $\sigma_1, \ldots, \sigma_m$ be the last m jobs executed on the second stage in reverse order, that is σ_1 is the last job, σ_2 the penultimate one, etc. Like in the previous case, job precedence relations must be verified. Thus, the job on position k, $k = 1 \ldots m$, must verify $\text{succ}\,(\sigma_k) \subseteq \{*, \sigma_1, \ldots, \sigma_{k-1}\}$.

Job σ_k can start on the second stage, only after all the first stage operations which are executed before are finished. In this case, the completion time of job σ_k is at least $\sum_i a_i + Z_2^k$, where $Z_2^k = b_{\sigma_k} - \sum_{i=1}^{k-1} a_i$ represents the exceedance of job k over the total first stage workload.

A lower bound for the HFS problem is given by (2.1), where Z_2 represents the least possible exceedance for any sequence of final jobs. In order to find the

Algorithm 2.1 Algorithm for finding the optimal solution of the relaxed problem used in $GLB1^1$ calculation.

1: $B^1, B^2 = \emptyset$
2: **for** $k = 1$ **to** $m + 1$ **do**
3: $\quad \sigma_k^1 = \arg\min a_i$, such that $|\mathrm{anc}\,(i)| \leq k$ and $i \notin B^1$
4: $\quad B^1 = B^1 \cup \{\sigma_k^1\}$
5: $\quad \sigma_k^2 = \arg\max b_i$, such that $|\mathrm{anc}\,(i)| \leq k$ and $i \notin B^2$
6: $\quad B^2 = B^2 \cup \{\sigma_k^2\}$
7: **end for**
8: $Z_1' = 0$
9: $S = a_{\sigma_{m+1}^1}$
10: **for** $k = m$ **to** 1 **do**
11: $\quad Z_1' = Z_1' + a_{\sigma_k^1} \cdot (m - k + 1) + \max\left(S - b_{\sigma_k^2}, 0\right)$
12: $\quad S = S + a_{\sigma_k^1}$
13: **end for**

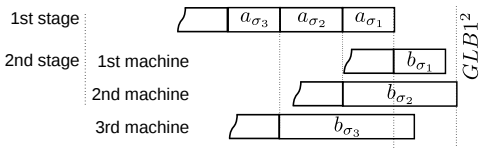


Figure 2.2: Final moments of a HFS with 3 machines on the second stage.

ordering of last m operations for which Z_2 is minimal the combinatorial problem (2.2) must be solved.

$$GLB1^2 = \sum_{i=1}^{n} a_i + Z_2 \tag{2.1}$$

$$\begin{aligned} Z_2 = \text{Minimize} \quad & \max_{k=1\ldots m} \left(b_{\sigma_k} - \sum_{i=1}^{k-1} a_{\sigma_i} \right) \\ \text{s.t.} \quad & \mathrm{succ}\,(\sigma_k) \subseteq \{*, \sigma_1, \ldots, \sigma_{k-1}\} \end{aligned} \tag{2.2}$$

Proposition 2.1. *Optimal solution of optimization problem* (2.2) *is given by the recurrent relation*

$$\sigma_k = \underset{\substack{i \notin \{*,\sigma_1,\ldots,\sigma_{k-1}\} \\ \mathrm{succ}(i) \subseteq \{*,\sigma_1,\ldots,\sigma_{k-1}\}}}{\arg\min} b_i$$

for all $i = 1 \ldots m$.

Proof. Suppose that $\sigma_1, \ldots, \sigma_m$ is the optimal solution of the problem having value Z_2, and also, suppose that there exists an operation p, $\mathrm{succ}\,(p) = *$, such that $b_p < b_{\sigma_1}$:

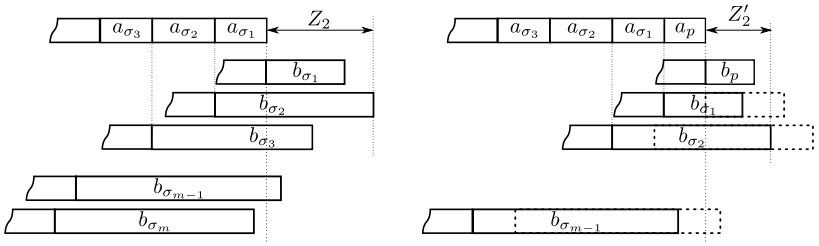


Figure 2.3: Solution Z'_2 compared to the initial one Z_2. On the right-hand side in dashed line are presented execution intervals of operations in the initial solution.

1. If $p \notin \{\sigma_1, \ldots, \sigma_m\}$ a new solution $p, \sigma_1, \ldots, \sigma_{m-1}$ (see Figure 2.3 for an illustration) will have the following value:

$$Z'_2 = \max\left(b_p, b_{\sigma_1} - a_p, \ldots, b_{\sigma_{m-1}} - \sum_{i=1}^{m-2} a_{\sigma_i} - a_p\right) = \max\left(b_p, \max_{k=1}^{m-1}\left(b_{\sigma_k} - \sum_{i=1}^{k-1} a_{\sigma_i}\right) - a_p\right) \leq Z_2$$

   The last result, $Z'_2 \leq Z_2$, contradicts the fact that Z_2 is the optimal solution.

2. If $p \in \{\sigma_1, \ldots, \sigma_m\}$ a new solution can be obtained by moving operation p before σ_1 (thus, p will be the last executed job). In an analogous way, we prove that the new solution is better.

We deduce that in an optimal solution $\sigma_1 = \arg\min_{\forall i, \text{succ}(i)=*} b_i$. Applying the same methodology the proposition is proved by induction. □

Algorithm 2.2 finds the optimal solution for the minimization problem in polynomial time using the previous result.

2.3.2 Global lower bound 2

In this subsection we introduce a global lower bound based on release times (heads) and delivery times (tails) adjustments. Let us assume that operation i cannot start earlier than its release date r_i, it is processed for either a_i or b_i time in function of the stage and must remain in the system for at least q_i time, which is the tail of operation i. In order to differentiate the first stage heads and tails from the second stage ones they are superscripted, so r_i^I, q_i^I are the heads and tails on the first stage and respectively r_i^{II}, q_i^{II} on the second stage. We use heads and tails instead of release dates and deadlines because many constraint concepts can be symmetrically expressed for heads and tails.

Algorithm 2.2 Algorithm for finding the optimal solution Z_2 of the problem used in $GLB1^2$ calculation.

$A = \{i \mid i \in \text{pred}(*), \text{succ}(i) = *\}$
$B = \emptyset$
for $k = 1$ **to** m **do**
  $\sigma_k = \arg\min b_i$, such that $i \in A$ and $i \notin B$
  $B = B \cup \{\sigma_i\}$
  $A = A \cup \{i \mid i \in \text{pred}(\sigma_k), \text{succ}(i) \subseteq \{*, \sigma_1, \ldots, \sigma_{k-1}\}\}$
end for
$Z_2 = 0$
$S = 0$
for $k = 1$ **to** m **do**
  $Z_2 = \max(Z_2, b_{\sigma_k} - S)$
  $S = S + a_k$
end for

A straightforward lower bound is (2.3), the first stage release dates and tails are not taken into account because they are dominated by the second stage heads and tails.

$$GLB2 = \max_i \left(r_i^{II} + b_i + q_i^{II}\right) \tag{2.3}$$

In what follows we introduce several constraints that the heads and tails must verify. Using constraint propagation techniques the heads and tails are iteratively adjusted until no modification is observed, the obtained $GLB2$ is a lower bound to the HFS problem.

2.3.2.1 Inter-stage precedence relations

In a two-stage flow shop the first stage operation of a job i must finish before its second stage operation starts: $r_i^{II} \geq r_i^I + a_i$. In the case of a no-wait flow shop this relation is more constrained by the fact that no idle time is permitted between the stages, so for the no-wait HFS we have $r_i^{II} = r_i^I + a_i$.

The same type of relations can be deduced for job tails: $q_i^I \geq q_i^{II} + b_j$ for the classic HFS and $q_i^I = q_i^{II} + b_j$ for the no-wait case.

2.3.2.2 Jobs precedence relations

The precedence relation graph $G = (V, E)$ for second stage operation is translated into the following constraint: $r_i^{II} \geq r_j^{II} + b_j$ for all $j \in \text{pred}(i)$. Symmetrically for job tails: $q_i^{II} \geq q_j^{II} + b_j$ for all $j \in \text{succ}(i)$. For the no-wait HFS the second stage precedence relations directly influence the partial order of first stage operations because of relations introduced in the previous section. In the case of classic HFS

the things are a little bit different, but it can be easily proved that the second stage precedence relations are dominating over the first stage ones.

2.3.2.3 Cumulative previous work

As said above, the second stage precedence constraints define a partial ordering over the first stage operations, thus before the execution of first stage operation i can start, all its first stage ancestors, defined by the second stage precedence constraints, must be completed. The release date r_i^I must be larger than the minimum makespan of a one-machine scheduling problem with release dates composed of ancestors of operation i. Let $C_{max}^{r^I}(i)$ be the optimal makespan of problem $1 \mid r_j \mid C_{max}$ for operations $j \in \text{anc}(i)$ with release dates r_j and processing times a_j, then the head of first stage operation i must verify $r_i^I \geq C_{max}^{r^I}(i)$.

The one-machine scheduling with release dates for jobs $j_1, \ldots, j_p$ is solved in polynomial time using the recurrent relation $c_{j_k} = \max\left(r_{j_k}, c_{j_{k-1}}\right) + a_{j_k}$ (Jackson's rule) with initial conditions $c_{j_1} = r_{j_1} + a_{j_1}$ and $r_{j_1} \leq r_{j_2} \leq \ldots \leq r_{j_p}$. Completion time c_{j_p} of the last job is the solution of the problem.

A straightforward relaxation of this constraint is $r_i^I \geq \sum_{j \in \text{anc}(i)} a_j$ which has a linear time computation, but it produces weaker release date bounds.

A constraint for the tails of the first stage operation is obtained in a similar way. The tail of operation i must verify $q_i^I \geq C_{max}^{q^I}(i)$ where $C_{max}^{q^I}(i)$ is the solution of the one-machine scheduling problem for descendants $j \in \text{desc}(i)$ with release date q_j and processing times a_j. A direct relaxation is obtained equivalently $q_i^I \geq \sum_{j \in \text{desc}(i)} a_j + \min_{j \in \text{desc}(i)} q_j^I$. In the above expression, the "min" term is added because the tails are not necessary zero as in the case with release dates.

In order to deduce equivalent relations for the heads and tails of the second stage operations, the parallel processor scheduling problem $Pm \mid r_i \mid C_{max}$ should be solved. The later problem is $\mathcal{NP}$-hard [42], thus a polynomial algorithm for solving it does not exist (unless $\mathcal{P} = \mathcal{NP}$). The parallel processor scheduling problem can be relaxed to a one-machine scheduling problem by dividing the processing times of the jobs by the number of processors. That is to say, we consider that a job can be executed simultaneously on all of the available processors.

For the second stage operation i, we consider the one-machine scheduling problem $1 \mid r_j \mid C_{max}$ for ancestor jobs of i with processing times $b_j' = b_j/m$ and release dates r_j^{II} for any $j \in \text{anc}(i)$. Let $C_{max}^{r^{II}}(i)$ be the optimal makespan of the above problem. The release date of second stage operation i must verify $r_i^{II} \geq \left\lceil C_{max}^{r^{II}}(i) \right\rceil$, a ceiling operator is used because the release date must be integer. A linear relaxation of the above constraint is:

$$r_i^{II} \geq \left\lceil \frac{\sum_{j \in \text{anc}(i)} b_j}{m} \right\rceil + \min_{j \in \text{anc}(i)} r_j^{II}$$

Symmetrically for the tails of the second stage operation i the following constraint is deduced $q_i^{II} \geq C_{max}^{q^{II}}(i)$. Where $C_{max}^{q^{II}}(i)$ is the optimal solution of the one-machine scheduling problem for the descendants $j \in \text{desc}(i)$ of operation i with processing times $b_j^{'} = b_j/m$ and release dates q_j. Equivalently the following linear relaxation is inferred, here we have $\min_{j \in \text{desc}(i)} q_i^{II} = q_*^{II} = 0$:

$$q_i^{II} \geq \left\lceil \frac{\sum_{j \in \text{desc}(i)} b_j}{m} \right\rceil$$

2.3.2.4 Jackson's preemptive schedule

Jackson's preemptive schedule (JPS) was introduced in [19]. It gives the optimal makespan for the preemptive one-machine scheduling with release dates and delivery times $1 \mid r_i, q_i, pmnt \mid C_{max}$. The obtained makespan value is a tight lower bound for the non-preemptive problem $1 \mid r_i, q_i \mid C_{max}$. JPS is the list schedule found by prioritizing the jobs with the most remaining work. The jobs are examined in increasing order of their release dates. At time instant t the job with the largest delivery time among the available jobs is scheduled, even if another job is in execution.

In $GLB2$ calculation the HFS problem is relaxed to $1 \mid r_i^I, q_i^I \mid C_{max}$ by dropping out the second stage and looking only at the first stage problem. The JPS is then used to adjust the global lower bound $GLB2$.

The JPS can also be used to adjust the heads and tails of operations. To adjust the head of operation c, one can build the JPS schedule where operation c has an infinite priority, thus operation c will start at time r_c. If the obtained schedule length is bigger than the upper bound UB of the HFS problem then the head of operation c can be increased. Let a_i^+ be the residual processing time of operation i at time r_c in the modified JPS schedule. Take the operations of $K_c^+ = \left\{ i \mid a_i^+ > 0, q_i > q_c \right\}$ in increasing order of q_i and find the first operation s for which relation $r_c + a_c + \sum_{q_i \geq q_s} a_i^+ + q_s > UB$ is verified. If such an operation exists then $r_c = \max\left(r_c, \max_{q_i \geq q_s} C_i\right)$ where C_i is the completion time of operation i in the usual JPS (where job c does not have an infinite priority). See [21] for more information and for an $O(n \log n)$ algorithm for updating the heads of all operations. Similarly the tails of operations can be adjusted by interchanging the roles of heads and tails.

2.3.2.5 Energetic reasoning

The previous constraints do not fully consider the limited number of machines on the second stage. In order to do so, we use the so called *energetic reasoning* in lower bound calculation for the multiprocessor scheduling problem [6, 35, 72].

Let $d_i = UB' - q_i^{II}$ be the deadline of second stage operation i, where UB' represents an attempt of upper bound for the HFS problem. Given a time interval

$[t_1, t_2] \subseteq [0, UB']$ we calculate for each job i the *left-work* $W_{left}(i, t_1, t_2)$ and the *right-work* $W_{right}(i, t_1, t_2)$ which represents the part of operation i that must be processed between t_1 and t_2 when the operation starts as soon as possible, thus at time r_i^{II}, and respectively as late as possible, at time $d_i - b_i$. The mandatory amount of work for operation i over the interval $[t_1, t_2]$ is the minimum between its left-work and right-work:

$$W(i, t_1, t_2) = \min\left(W_{left}(i, t_1, t_2), W_{right}(i, t_1, t_2)\right)$$

The total amount of work for interval $[t_1, t_2]$ is the sum of works $W(i, t_1, t_2)$ for all the operations:

$$W(t_1, t_2) = \sum_i W(i, t_1, t_2)$$

If the total amount of work $W(t_1, t_2)$ exceeds the amount of available "energy" $m(t_2 - t_1)$ then the problem is infeasible. This property can be used to increase the global lower bound value. Let L be the best value of $GLB2$ obtained so far. Set $UB' = L$ and do the above computations. If an interval $[t_1, t_2]$ for which the problem is infeasible is found then the current UB' value can be increased by at least:

$$\Delta(t_1, t_2) = \left\lceil \frac{W(t_1, t_2) - m(t_2 - t_1)}{m} \right\rceil$$

The UB' value is adjusted by adding to it the maximal increase calculated for each time interval, the new value of UB' becomes a lower bound to the HFS problem:

$$UB' = L + \max_{[t_1, t_2] \subseteq [0, L]} \left(\Delta(t_1, t_2), 0\right) \tag{2.4}$$

The direct calculation of maximal increase using relation (2.4) is pseudo-polynomial because the number of time intervals that must be examined is proportional to L^2. Hopefully not all the intervals are relevant, in [6] it is proved that only $O(n^2)$ increase calculations are representative. Particularly, in a simplified version, only the intervals $[t_1, t_2]$, such that $t_1 \in \{r_i\} \cup \{r_i + b_i\} \cup \{d_i - b_i\}$ and $t_2 \in \{d_i\} \cup \{r_i + b_i\} \cup \{d_i - b_i\}$, have to be examined.

The available energy can also be used to calculate time bound adjustments for operations release dates and deadlines. Let $SL(i, t_1, t_2) = m(t_2 - t_1) - W(t_1, t_2) + W(i, t_1, t_2)$ be the available energy over $[t_1, t_2]$ when operation i is not considered. If the left-work $W_{left}(i, t_1, t_2)$ of an operation i is bigger than the available energy $SL(i, t_1, t_2)$, then only a part, smaller or equal to $SL(i, t_1, t_2)$, of i can be processed during the interval $[t_1, t_2]$. The release date of operation i can be updated: $r_i = t_2 - SL(i, t_1, t_2)$. Similarly if $W_{right}(i, t_1, t_2) > SL(i, t_1, t_2)$ then the deadline is adjusted $d_i = t_1 + SL(i, t_1, t_2)$.

2.3.2.6 $GLB2$ computation

The computation of the global lower bound $GLB2$ is performed as follows. Firstly, the inter-stage precedence, jobs precedence and cumulative previous work constraints are grouped into a constraint programming model and a constraint propagation method is used to compute the heads and the tails for each operation. The heads and tails obtained in this way are used in a list scheduling heuristic (defined in the sequel) in the priority function calculation. The solution found by the list scheduling is an upper bound, UB, for the HFS problem. Which afterwards together with the JPS and energetic constraints are added to the constraint programming model defined above. Using the propagation technique new and eventually better values for operations heads and tails are obtained.

Due to the use in the JPS and energetic constraints of an upper bound, it is clear that the more this upper bound is tight the more the $GLB2$ is constrained, thus potentially better values for $GLB2$ could be obtained. The last fact motivated us to find an upper bound candidate UB', $UB' \in [GLB2, UB]$, such that for UB' the HFS problem is feasible and for $UB' - 1$ the problem becomes infeasible. A dichotomization procedure is introduced in order to explore the $[GLB2, UB]$ interval more optimally. In this way, a new global lower bound $GBL2^{dich} = UB'$ is obtained. The calculation of this bound is pseudo-polynomial and depends on initial UB' limits. In our calculation experiments the dichotomization procedure takes, in the worst case, less than 10 seconds, taking into account that the optimization of the constraint propagation code was not envisaged.

2.4 List scheduling

A reliable heuristic from the multiprocessor scheduling literature is the list scheduling (LS). Roughly speaking, in a LS algorithm the tasks are ordered (statically or dynamically) according to a priority rule and then are assigned in this order to the first available processor. Different priority rules have been proposed. Critical path based rules are known to provide the best results in the contest of multiprocessor scheduling.

Algorithm 2.3 is a modified version of the LS heuristic which is used for solving the HFS problem. The main difference from the list scheduling used in multiprocessor problems is that in this algorithm the start time of a job takes into account also the first stage processing. The following notation are used in the algorithm: T is a variable that stores the time when the first stage machine is available (initially it is available at instant zero). When a second stage machine M is chosen for the current job to be scheduled we denote by F the moment of time when it is available. The only difference between the list scheduling we propose for the classical and for the no-wait HFS consists in how the update of the first stage machine availability time T is made (algorithm line 10).

Algorithm 2.3 List scheduling (LS) algorithm for the HFS problem (s_j - second stage start time of job j).

1: $S = \{0\}$ {Jobs ready for scheduling}
2: $s_0 = 0$
3: $T = 0$
4: **while** $S \neq \emptyset$ **do**
5: Calculate priorities p_i for jobs $i \in S$
6: Choose top priority job $j = \arg\max_{i \in S} p_i$, $S = S - j$
7: Choose the earliest available second stage machine M for j
8: Determine time F when machine M is available
9: Schedule j on M at time $s_j = \max(T + a_j, F)$
10: Classical HFS: $T = T + a_j$. No-wait HFS: $T = s_j$
11: $S = S \cup \{i \in \text{succ}(j)\}$ such that all the predecessors of i are finished
12: **end while**

Two priority rules are proposed. The first priority rule P^I is critical path based, particularly the CP/MISF (critical path/most immediate successors first) rule described in [66]. Priority value (2.5) is computed for each job $i \in S$ and the job with the largest p_i^I is chosen for being scheduled next.

$$p_i^I = q_i^{II} + \frac{|\text{succ}(i)|}{n+1} \tag{2.5}$$

This priority function ensures that the next job to schedule is the one which has the largest tail. Or, when the tails of two jobs are equal, it selects the job with the largest number of successors.

A second rule P^{II} is proposed because the critical path based rule does not take into account the idle time a list scheduling algorithm potentially creates on the first stage. In this priority rule, the next job to schedule is the one that *fits the best* the first stage machine free time, i.e. the job $i \in S$ having the highest value (2.6) is chosen for scheduling (here we use the same notation as in Algorithm 2.3).

$$p_i^{II} = -|F - (T + a_i)| \tag{2.6}$$

This priority rule has similarities with the ETF (Earliest Time First) rule from the multiprocessor scheduling [59], and actually when relation $T + a_i \leq F$ is satisfied, P^{II} is the ETF priority rule.

2.5 Adaptive randomized list scheduling

A drawback of the list scheduling heuristic is that it returns a single solution by breaking any ties in the priority value of two or more jobs arbitrarily. Bad decisions in choosing the job to schedule (among the jobs having same priority),

potentially, makes the heuristic to find low quality solutions on some instances. In order to overcome this drawback, the list scheduling algorithm can be executed several times, each time breaking ties randomly.

Inspired by the work [56] on the randomization of greedy algorithms, we further generalize this method by introducing a randomization parameter α, $\alpha \in [0;1]$, which aims to control the randomness of the list scheduling. Let S be the set of ready jobs to be scheduled and let $p_{max} = \max_{i \in S} p_i$, $p_{min} = \min_{i \in S} p_i$ be the maximum, respectively the minimum priority values of these jobs. At each iteration of the list scheduling algorithm the next job to schedule is chosen uniformly from the jobs with the priorities belonging to the range $[p_{max} - \alpha\,(p_{max} - p_{min})\,; p_{max}]$. In this way, by adjusting coefficient α different behaviors of the list scheduling can be obtained, i.e. for $\alpha = 0$ we have the list scheduling with random ties breaking and for $\alpha = 1$ we obtain a list scheduling with a random priority rule.

The randomized list scheduling algorithm consists in executing the list scheduling with the random selection rule described above for a number of times and to retain the best obtained schedule as solution.

During the experimental phase a drawback of the randomized list scheduling was revealed. Actually the randomization parameter α cannot be chosen unequivocally for different problem parameters, as number of jobs, stage work loads, etc. The adaptive randomized list scheduling (ARLS) algorithm is then introduced to overcome this issue, see Algorithm 2.4. In this algorithm a preliminary phase is performed, during which the quality of solutions obtained for each randomization parameter is estimated. Thus, the randomized list scheduling is executed for each $\alpha \in \mathcal{A}$, where $\mathcal{A}$ is the set of used randomization parameters, the same number of times $SampCnt$ and the best solution S_α is saved. Afterwards, in function of the distance of S_α from the worst solution obtained so far S_{max} a proportional quota N_α from the total iteration count $IterCnt$ is assigned to parameter α. Thus, better is the solution S_α more iterations with parameter α are done in the second phase. When all the solutions are equal the total iteration count is split into equal parts for each α. Finally, the randomized list scheduling is executed for each α, N_α iterations and the best obtained solution is returned.

The performance of ARLS relies on the good choice of sampling phase number of iterations $SampCnt$, on the randomization parameters α and on the second phase iterations count $IterCnt$. The parameter $SampCnt$ must give statistically reliable estimates of P_α. In order to control the overall complexity of the ARLS algorithm the number of second iterations $IterCnt$ shall be carefully chosen.

We must note that there is practically no use of adapting on-line the randomization parameter α. An ARLS version which is updating α during the execution was tested, the differences in the obtained solutions were negligible.

Algorithm 2.4 Adaptive randomized list scheduling (ARLS).

Input: $\mathcal{A}$ - randomization parameters α to use
Input: *SampCnt* - number of sample runs for each α
Input: *IterCnt* - number of iterations for the search phase
Output: Best found solution, *best*

1: $S_\alpha = RandomizedListScheduling\left(\alpha, SampCnt\right), \forall \alpha \in \mathcal{A}$
2: $S_{max} = \max_\alpha S_\alpha$
3: $S_{min} = \min_\alpha S_\alpha$
4: **if** $S_{max} \neq S_{min}$ **then**
5: $\quad P_\alpha = {}^{(S_{max}-S_\alpha)}/_{\sum_{\alpha'}(S_{max}-S_{\alpha'})}, \forall \alpha \in \mathcal{A}$
6: **else**
7: $\quad P_\alpha = {}^{1}/_{|\mathcal{A}|}, \forall \alpha \in \mathcal{A}$
8: **end if**
9: $N_\alpha = P_\alpha \cdot IterCnt, \forall \alpha \in \mathcal{A}$
10: $best = S_{min}$
11: **for all** $\alpha \in \mathcal{A}$ **do**
12: $\quad sol = RandomizedListScheduling\left(\alpha, N_\alpha\right)$
13: $\quad$ **if** $best > sol$ **then**
14: $\quad\quad best = sol$
15: $\quad$ **end if**
16: **end for**
17: **return** *best*

2.6 Experimental results

The algorithms described earlier were implemented using the C++ language. We have used the constraint propagation framework from ILOG CP solver to implement the $GLB2$ calculation. The dichotomization procedure of $GLB2^{dich}$ calculation was implemented as a goal for CP solver. We shall note that only the constraint propagation feature of ILOG CP solver was used. The test programs were executed on an Intel Core2 Duo P8600 system without explicit parallelization[1].

2.6.1 Instance generation

For testing the performance of the proposed heuristics and global lower bounds we use a set of 360 graphs from the "standard task graph set", which can be found at [1]. One half of the graph instances contain 50 jobs, the other half has 100 jobs. The graphs have either fully random structure or are composed of layers of random sizes. Each task processing time is randomly sampled using uniform, exponential or normal distributions with either one or two modes.

A HFS instance from such a graph is generated as follows. The precedence relations between the tasks are used as precedence relations for second stage operations.

The processing time c_i of task i is split into two parts, $a_i = \rho c_i$ and $b_i = (1-\rho)\, c_i$. Values a_i and b_i are rounded to the nearest integers such that relation $c_i = a_i + b_i$ remains valid. The coefficient ρ is used to obtain different load balancing between the first stage and the second stage. Let $r = \frac{\sum a_i}{\sum b_i/m}$ denote the desired ratio between the first and second stage work load (i.e. when $r = 1$ the processing load is balanced between the stages). Then the coefficient ρ can be computed using relation:

$$\rho = \frac{r}{r+m}$$

Three ratios r are used in order to examine the performance of heuristics and of lower bounds for different load balances between the stages. For each task graph several HFS instances are generated, so 180 different HFS instances are obtained for each number of jobs, load ratio and number of second stage machines.

2.6.2 Global lower bounds

In the first experiment we examine the relative performance of the global lower bounds for each version of HFS problem, the classical and the no-wait one. The global lower bounds $GLB1$, $GLB2$ and $GLB2^{dich}$ are computed for 9720 problem

[1] As a multi-start heuristic, our algorithm can be straightforwardly parallelized.

		$GLB1$ vs. $GLB2^{dich}$		
	n	$>$	$=$	$<$
Classic	50	8.58%	23.17%	68.25%
	100	11.67%	26.98%	61.36%
No-wait	50	8.48%	22.16%	69.36%
	100	11.63%	26.11%	62.26%

Table 2.1: Relative comparison of $GBL1$ and $GLB2^{dich}$. Percentage of instances for which $GLB1 > GLB2^{dich}$, $GLB1 = GLB2^{dich}$ and $GLB1 < GLB2^{dich}$.

instances generated as described earlier, with $r \in \{2/3, 1, 3/2\}$, $m \in 2, \ldots, 10$ and $n = 50, 100$.

Firstly we want to study the improvement brought by the dichotomization procedure on the $GLB2$ quality. The instances for which the lower bound calculated by dichotomization, $GLB2^{dich}$, is strictly better than $GLB2$ are counted. For the classical HFS the dichotomization improved the lower bound of 1561 problem instances, which represents 16% from the total number. In the case of no-wait HFS the improvement was observed in 4355 (45%) cases. For instances of 100 jobs the number of improvements decreases slightly (<1%) when compared to instances of 50 jobs. In order to sample the quality of these improvements the deviations $1 - {}^{GLB2}/_{GLB2^{dich}}$ were calculated for each instance. In the case of classical HFS the average deviation is less than 0.1% and for the no-wait HFS less than 0.5%. Although the dichotomization procedure improves the $GLB2$ bound quality, its relatively high computation cost limits its use.

In a second experiment we study the relative performance of $GLB1$ and $GLB2^{dich}$. The number of instances for which $GLB1$ is strictly better, both bounds give the same value and $GLB2^{dich}$ is strictly better are counted. The results in percentage from the total number of instances are presented in Table 2.1. As we can observe there is no substantial difference in the behavior of bounds for classic and no-wait HFS, probably because the same set of instances are equally difficult for $GLB2^{dich}$ in both HFS types. Another interesting fact is that the quality of $GLB1$ increases for instances with more jobs. The result changes for "layered" instances, for which the $GLB1$ is strictly better than (equal to) $GLB2^{dich}$ in 13% (27%) of the cases for instances of 50 jobs and 17% (34%) for instances of 100 jobs no matter the HFS type.

In order to compare the performance of lower bounds function of load ratio and number of second stage machines, for each pair (r, m) we count the number of times each global lower bound is strictly better than the other bound. Let $p_1(r, m)$ and $p_2(r, m)$ be the ratio the first bound is better $GLB1 > GLB2^{dich}$ and respectively the second is better $GLB1 < GLB2^{dich}$ expressed in percents from the total number of instances for each (r, m) and let $p_{1,2}(r, m)$ be the ratio the bounds are equal. In Figure 2.4 $p_1(r, m)$, $p_{1,2}(r, m)$ and $p_2(r, m)$ are plotted

for each pair (r, m). The results for classic HFS and no-wait HFS are practically the same, consequently only the no-wait case is only plotted.

We observe that for load ratios $r = 1$ and $r = 3/2$ the $GLB1$ bound is practically never greater than $GLB2^{dich}$. For small number of second stage machines m the first bound performs better than for large m, being equal to $GLB2^{dich}$ in approximatively 60% of the cases for $r = 3/2$ and 40% for $r = 1$ when $m = 2$. We suppose that this is due to the fact that for instances with large first stage workloads the one-machine based constraints perform better than the simple sum of first stage durations, used in $GLB1$.

When the second stage workload is dominating, $r = 2/3$, the first global lower bound performs better than in the previous cases. The best results of $GLB1$ are obtained for $m = 4$ the first bound being better in more than 50% of the cases. For other values of second stage machines, lesser or greater than 4, the performance of $GLB1$ decreases. The definition of $GLB1$ makes it perform better on instances where the workloads are asymmetrically distributed between the stages. This can be seen in the results, the overall performance of $GLB1$ is lower for $r = 1$ than for other two load ratios.

For all load ratios, with the increase of number of machines m the relative performance of the second global lower bound also increases, obviously due to the fact that for large values of m the second stage critical path plays a higher role in the HFS execution.

2.6.3 List scheduling heuristics

Firstly we investigate the influence of sampling phase iterations count $SampCnt$ and second phase number of iterations $IterCnt$ on the ARLS performance. The goal is to choose parameters that produce good solutions of the heuristic when compared to its complexity. The same set of instances as in previous section is used.

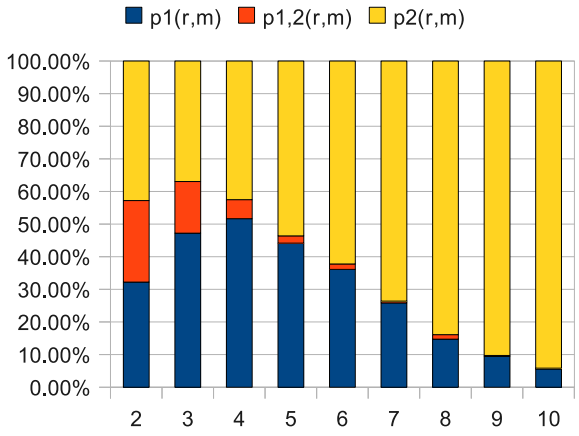
The randomization parameter α takes five values from the set $\mathcal{A}$:

$$\mathcal{A} \in \{0, 0.2, 0.4, 0.6, 0.8\}$$

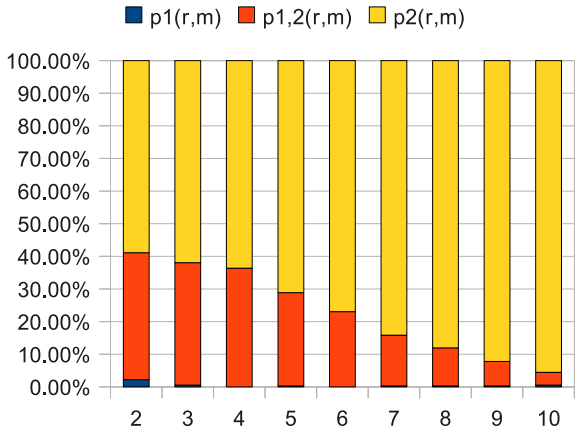
As the fully randomized list scheduling is outside the scope of this study value 1 for parameter α is not used[2]. Theoretically, when $\alpha = 1$ the used scheduling priority rule should not influence the results because the list scheduling is fully random. A finer division for α is not necessary because the performance increases insignificantly, but the total number of iterations raises.

In the sampling phase of ARLS heuristic 6 values of iterations count $SampCnt$ have been tested: 50, 100, 150, 200, 250 and 300. In order to have the same total number of iterations independently of $SampCnt$ value the number of second

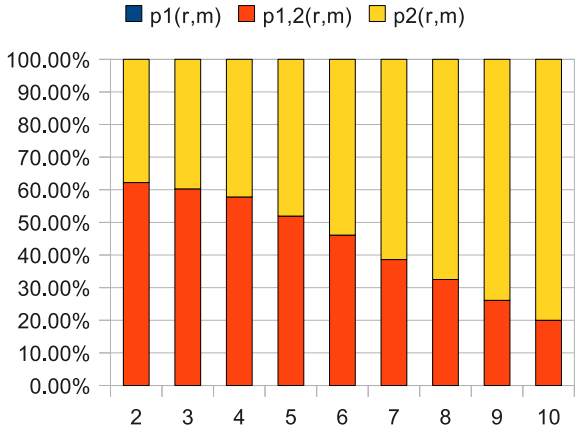
[2]We have executed the ARLS heuristic also with $\alpha = 1$ but no increase in the quality of the solutions was observed.



(a) $r = 2/3$



(b) $r = 1$



(c) $r = 3/2$

Figure 2.4: Relative comparison of $GLB1$ and $GLB2^{dich}$ for each pair (r, m) of parameters.

phase iterations count is $IterCnt = n^{\beta} + (300 - SampCnt) \cdot |\mathcal{A}|$, where $1.5 \leq \beta \leq 2$. Six values for β are experimented with such that the obtained $IterCnt$ are equidistantly situated. The execution time, in the worst observed case, is under 5 seconds and mainly depends on the graph edge density.

In order to minimize the influence of the randomness on the performance study, for each problem instance the ARLS algorithm is executed 10 times and the averaged result is retained for comparison. The deviation $S/GLB_{max} - 1$ of the averaged solution S from the maximal global lower bound $GLB_{max} = \left(GLB1, GLB2^{dich}\right)$ is calculated for each instance.

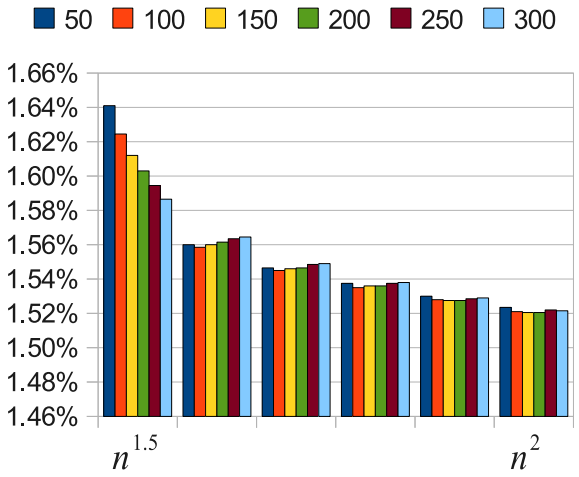
A preliminary experiment has proved that better solutions are obtained when the ARLS heuristic is executed two times, firstly with priority rule P^I and after with P^{II}, keeping the best solution of each run, even if the total number of iterations is two times smaller.

The averaged deviations for each $SampCnt$ and $IterCnt$ are illustrated in Figure 2.5. We observe that when the second phase iterations count is the lowest $IterCnt = n^{1.5}$, better results are obtained for larger sampling phase iterations count. This can be explained by the fact that the second phase iterations number is insufficient in order to explore the solution space. Another interesting fact is that for large second phase iterations count it is not always better to have larger sampling phases. We suppose this is due to the fact that parameter P_{α} (see 2.4) is reliably enough estimated for smaller $SampCnt$ and it is better to do more iterations in the second phase. For $IterCnt = n^2$ the difference in solutions obtained with different $SampCnt$ is insignificant, being under 0.01%. It can be seen that a sampling phase with $SampCnt = 100$ gives statistically reliable estimations for the parameter P_{α}. Note that, contrary to $IterCnt$, $SampCnt$ can be chosen independently of the instance size, based on statistical convergence considerations. So we use this sample phase iterations number in the next experiments, n^2 is used for second phase iterations count.

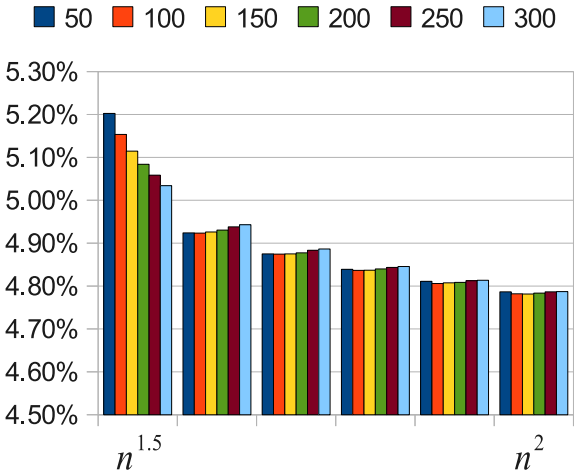
In the next experiment the priority rules are compared. It was determined that for the classical HFS the PI priority rule dominates in average PII in all the test instances, which can be explained by the dominance of the multiprocessor scheduling problem in the classical HFS, for which critical path rules are better. In the case of no-wait HFS the second priority rule PII produces better solutions for load ratios $r = 2/3$, $r = 1$ and for a second stage machines count $m \leq 4$.

In order to see the improvement of the randomization on the ordinary list scheduling in Table 2.2 the quality of solutions obtained by the ARLS heuristic and the ordinary list scheduling heuristic are compared. The randomization always improves the solutions found by the list scheduling, the deviations of solutions are decreased by ARLS with approximatively 40%.

Table 2.3 presents the average deviations of the solutions calculated by the ARLS heuristic in function of work load ratio r, second stage machines count m and number of jobs n. Also in the table are illustrated the averaged values of



(a) Classical HFS



(b) No-wait HFS

Figure 2.5: Influence of parameters *SampCnt* and *IterCnt* on the average deviation of the ARLS algorithm (the *IterCnt* parameter is on horizontal axis and different bar colors represent *SampCnt*). The deviation is computed for the minimal solution obtained by the two priority rules.

	50		100	
	LS	ARLS	LS	ARLS
Classic	2.98%	1.83%	1.95%	1.21%
No-wait	7.83%	4.62%	7.97%	4.94%

Table 2.2: Upper bound on the average deviations established by algorithms ARLS and LS together with GLB_{max}.

n	50				100			
m\r	2/3	1	3/2	$\overline{dev_m}$	2/3	1	3/2	$\overline{dev_m}$
2	0.47%	3.23%	0.24%	1.31%	0.24%	2.16%	0.07%	0.82%
3	0.82%	3.41%	0.48%	1.57%	0.47%	2.36%	0.31%	1.05%
4	1.17%	3.29%	0.74%	1.73%	0.55%	2.31%	0.78%	1.21%
5	1.50%	2.99%	1.54%	2.01%	0.67%	2.16%	1.05%	1.29%
6	1.64%	2.72%	2.37%	2.25%	0.77%	1.79%	1.33%	1.30%
7	1.63%	2.25%	3.07%	2.32%	0.82%	1.63%	1.80%	1.42%
8	1.20%	1.82%	3.17%	2.07%	0.65%	1.26%	2.08%	1.33%
9	0.78%	1.29%	3.14%	1.73%	0.53%	1.01%	2.29%	1.28%
10	0.45%	0.92%	3.03%	1.46%	0.33%	0.86%	2.50%	1.23%
$\overline{dev_r}$	1.07%	2.44%	1.98%	**1.83%**	0.56%	1.73%	1.36%	**1.21%**

(a) Classical HFS

n	50				100			
m\r	2/3	1	3/2	$\overline{dev_m}$	2/3	1	3/2	$\overline{dev_m}$
2	2.53%	10.18%	2.59%	5.10%	2.80%	11.56%	3.32%	5.89%
3	3.14%	9.67%	2.32%	5.04%	3.09%	10.64%	2.72%	5.49%
4	3.53%	8.94%	2.78%	5.08%	3.09%	10.07%	3.43%	5.53%
5	3.61%	7.75%	3.56%	4.97%	3.21%	9.35%	3.84%	5.47%
6	3.69%	7.03%	4.52%	5.08%	3.09%	8.21%	4.21%	5.17%
7	3.49%	6.03%	5.28%	4.93%	2.83%	7.13%	4.79%	4.92%
8	2.66%	4.97%	5.44%	4.36%	2.34%	5.93%	5.00%	4.42%
9	1.86%	3.81%	5.59%	3.75%	1.86%	4.82%	5.24%	3.98%
10	1.22%	2.96%	5.69%	3.29%	1.32%	3.96%	5.58%	3.62%
$\overline{dev_r}$	2.86%	6.81%	4.20%	**4.62%**	2.62%	7.96%	4.24%	**4.94%**

(b) No-wait HFS

Table 2.3: Average deviation of the minimal solution found by the ARLS heuristic using both priority rules.

deviations for each m and r. As we can see on average for the classical HFS the deviation is lower than 2% and for the no-wait case the deviation is under 5%. The deviations per number of second stage machines, $\overline{dev_m}$, tend to decrease for larger values of m. In both HFS types the hardest instances are those for which the workload is balanced between the stages, i.e. $r = 1$, the largest deviations being obtained for small m. In the case of no-wait HFS, when $m = 2$ and $r = 1$ the deviation is $<$11% for 50 jobs and $<$12% for 100 jobs. With the increase of the number of second stage machines this deviance decreases, being under 4% for $m = 10$. A closer examination revealed that the largest deviations are obtained for instances for which the processing times distributions follow an exponential law. In order to see what is their influence, the deviations were recalculated without the exponential processing time instances. It was found that in the case of no-wait HFS the worst observed deviation falls down from 12% to 8%.

2.7 Conclusions

In this chapter, two versions of the two-stage hybrid flow shop problem with second stage precedence constraints and parallel machines were investigated, the classical case and the no-wait one. An adaptive randomized list scheduling (ARLS) heuristic, together with two priority rules, were proposed for solving both problem versions. Our heuristic is made of a constructive part (ARLS) associated to a global lower bound which allows to obtain provably good solutions.

The practical application of the hybrid flow shop problem occurs in the scheduling of dataflow applications on clustered multi-core processors. Using this problem allows a fine-grain modeling of a dataflow application, below the task level.

The evaluation of the heuristic was done using randomly generated problem instances. The ARLS algorithm produced better schedules for all of the examined cases when compared to the ordinary list scheduling. The best results were obtained in the case of the classical HFS problem version, with an average deviation established by the algorithm under 2% from the optimum. For the no-wait HFS version, that deviation was smaller than 5%. The critical path based priority rule provided better solutions in average.

Chapter 3

Speculative prefetching for dataflow branching structures

In the previous chapter, prefetching strategies for dataflow applications executed on clustered multi-core processors have been studied. For keeping the processing cores busy, we have to heavily rely on prefetching data from the external memory to the cluster memory. To achieve high performances in presence of data dependent control (conditional execution), one should further speculate on data prefetching. In this chapter we examine the speculative prefetching possibilities offered by a special construction in dataflow applications: the branching structure. We focus on finding optimum prefetch strategies for a simple, n-way branching structure with respect to several objective functions and exhibit polynomial algorithms for doing so.

A reduced version of the works presented in this chapter has been published in the Proceedings of the International Symposium on Combinatorial Optimization [23].

This chapter, is organized as follows. After a detailed description in Section 3.1 of the speculative data prefetching problem for branching structures, we present some related works in Section 3.2. Further, in Section 3.3 we focus on expected execution time objective, Section 3.4 deals with the more complicated situation of worst-case execution time objective and Section 3.5 concludes.

3.1 Problem formulation

An n-way branching structure is a construction in the dataflow model which executes one task from a set of n tasks in function of the value received on its conditional input. The branching structure assures conditional execution in dataflow graphs. In Figure 3.1 we recall the illustration of an n-way branching structure. Each task $\mathbf{T_i}$ is composed of two operations: *data loading* of duration a_i and *execution* of duration b_i. The data loading operations load data from

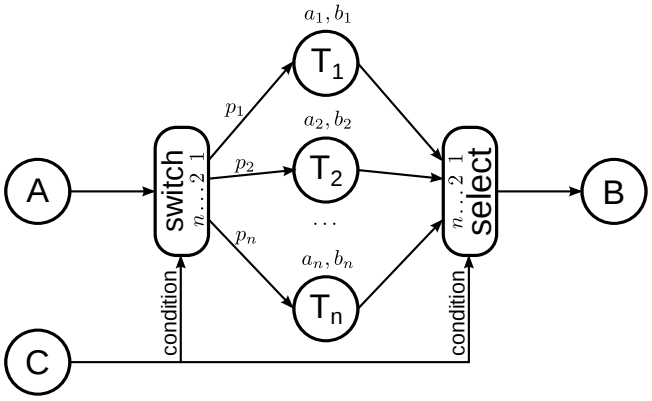


Figure 3.1: An n-way branching structure annotated with data loading duration a_i, execution duration b_i and branch execution probability p_i.

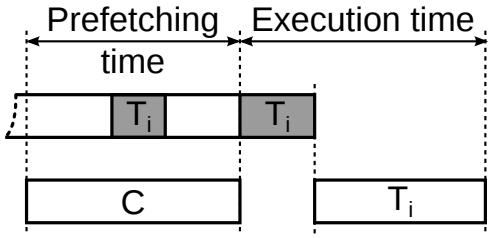


Figure 3.2: Illustration of prefetching and execution times for branching structure in Figure 3.1. The i-th branch is selected for execution by the output of task **C**.

the external memory into the cluster memory. These operations are executed sequentially because of a single memory access channel. We assume that there are no common data between the branches[1], thus the data loading durations do not depend on one another. The execution of a task can start only after all its data is loaded, i.e. the corresponding data loading operation is finished. Task execution depends also on the branch selected for execution by the branching structure. Whilst the task data loading can be executed in advance, without knowing what branch will be selected. In this way, speculative data prefetching can be performed until the branching structure's control input becomes available, in our example until task **C** finishes. In what follows, we call this period *prefetching time*. We call branching structure execution time, or simply *execution time*, the period between the start of the SWITCH actor and the end of the SELECT actor. That is to say the prefetching time is not included in the execution time. Refer to Figure 3.2 for more details (data loading operations are colored in grey).

Besides data loading and execution durations, each branch has a statistical measure associated. Let p_i denote the probability of the i-th branch to be executed, or simply *branch probability*. Clearly that relation $\sum_i p_i = 1$ is verified.

[1]Hereafter we use the notions of task and branch on an equivalent basis.

It is assumed that subsequent decisions are independent, thus the probability of branch execution is independent from previous executions of the branching structure (in the context of cyclic execution of dataflow applications). Several ways can be used to retrieve probability values for branches. The simplest way is to assume that the branches are equiprobable, i.e. $p_i = 1/n$ for any branch $i \in 1, \ldots, n$. If empirical results about application execution are available[2] then accurate statistical estimates of the probabilities can be obtained, e.g. the average execution rate of a branch. Another way is to use a priori knowledge from the program designer, thus the person who writes the dataflow program must assign probabilities to each branch.

Our goal is to define data prefetching strategies, so as to minimize several objective functions: *expected execution time* and *worst-case execution time*. The mathematical expectation objective is interesting because it allows to obtain, in average, higher execution performances. The price to pay is the induced indeterminism in the execution time of branching structures. On the other hand, the worst-case objective permits to precisely bound the execution time of branching structures. Two distinct prefetching strategies are examined: a *fractional* strategy, in which one is allowed to prefetch fractions of branch data, and an *all-or-nothing* strategy, in which this possibility is not allowed.

The available prefetching time, T, is supposed to be a continuous random variable. After this time elapses, one and only one of the branches is executed. The probability density function (pdf) of the prefetching time T is a bijection $g : \mathbb{D} \to [0, 1]$ such that $g(t) = \Pr(T = t)$ for any $t \in \mathbb{D}$ and $\mathbb{D} = [0, \sum_i a_i[$ is the domain of definition. With no loss of generality we exclude the degenerate case, $T > \sum_i a_i$, when all the data can be prefetched.

3.2 Related works

To the best of our knowledge, there are practically no works that treat speculative adapted (compile-time) prefetching problematics in branching structures. In paper [87] an adaptive (run-time) prefetching strategy for boolean-value control actors is studied. One can mention the works dealing with conditional task graphs, which are task graphs containing conditional branches (a kind of branching structures). The literature on task graphs with conditional branches is scarce and mainly consists in methods for allocating and scheduling them onto multiprocessor systems [110, 77].

[2]Empirical results about application execution can be retrieved using program profiling techniques. This method is interesting in an iterative compilation process [40] where the compiler chain "learns" from the application executions.

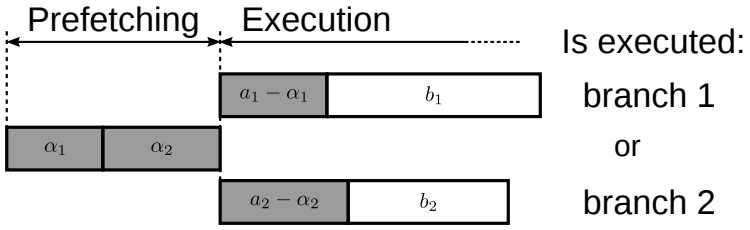


Figure 3.3: Example of a 2-way branching structure execution.

3.3 Expected execution time

In this section we examine the expected execution time minimization objective. We start by investigating the fractional prefetch strategy and then we study the all-or-nothing prefetch strategy. In both cases, we examine particular cases of the problem in terms of the available prefetching time. Afterwards we examine the more general case in which the available prefetching time is a continuous random variable.

3.3.1 Fractional prefetch

Suppose we have an n-way branching structure. Each branch i, $i \in 1, \ldots, n$, has parameters: a_i (data loading time), b_i (execution time) and p_i (branch probability). Let α_i, $0 \leq \alpha_i \leq a_i$, denote the fraction of the data loading operation i which is executed during the prefetching period, i.e. α_i is the prefetching time of branch i. We suppose that the available prefetching time is constant and is denoted by t. We look for optimal prefetching durations α_i such that the expected execution time is minimal.

Suppose that the branch i (prefetched for α_i time) is executed, then the execution time will be $a_i - \alpha_i + b_i$. An example of a 2-way branching structure execution is presented in Figure 3.3. The expected execution time is the average of branch execution times weighted by respective branch probabilities. The following linear program minimizes the expected execution time of a branching structure under a prefetching time constraint:

$$\begin{aligned} \text{Minimize} \quad & \sum_i p_i \left(a_i - \alpha_i + b_i\right) \\ \text{s.t.} \quad & \sum_i \alpha_i = t \\ & \alpha_i \in [0, a_i] \,, \; \forall i \end{aligned}$$

The last linear program is reformulated by substituting $\alpha_i = a_i \cdot x_i$ and taking the

complement of the objective function (knowing that $\sum_i p_i (a_i + b_i)$ is constant):

$$\begin{aligned} \text{Maximize} \quad & \sum_i p_i a_i x_i \\ \text{s.t.} \quad & \sum_i a_i x_i = t \\ & x_i \in [0,1], \ \forall i \end{aligned}$$

This program is nothing else but the linear programming form of the fractional knapsack problem. The fractional knapsack problem can be solved exactly in polynomial time using the well-known Dantzig algorithm [67]. In Algorithm 3.1 is illustrated an adaptation of Dantzig's algorithm for the context of our problem. This algorithm consists in prefetching the branches in decreasing order of their probabilities, as long as the prefetching time allows it.

Algorithm 3.1 Expected execution time minimization algorithm, fractional prefetch.

Input: Branch parameters a_i, b_i, p_i for any $i = 1, \ldots, n$
Input: Available prefetching time t
Output: Prefetching ratios α_i

```
1: Sort branches in decreasing order of p_i
2: α_i = 0 for any i = 1, ..., n
3: k = 0
4: while t > 0 and k < n do
5:     α_k = min(t, a_k)
6:     t = t − α_k
7: end while
```

Although elementary, this is a very interesting result: we obtain a solution whose structure does not depend on the available prefetching time t. So, the above hypothesis about fixed prefetching time can be neglected. In an optimal prefetching strategy, for any given prefetching time pdf $g(t)$, priority should be given to the branch with the highest probability.

3.3.2 All-or-nothing prefetch

In the case of an all-or-nothing prefetch strategy an order over the branches should be defined. We suppose that the available prefetching time follows an uniform distribution over the interval $\mathbb{D}$, i.e. $t \in \mathbb{D}$ with equal probabilities. Let $\sigma \in \Pi(n)$ be an ordering of data loading operations. We want to find an order σ such that for any prefetching time t the expected execution time is minimal. Let $f_{ex} : \Pi(n) \times \mathbb{D} \to \mathbb{R}^+$ be a function that associates to a branch order $\sigma \in \Pi(n)$ and an available prefetching time $t \in \mathbb{D}$ the branching structure expected execution

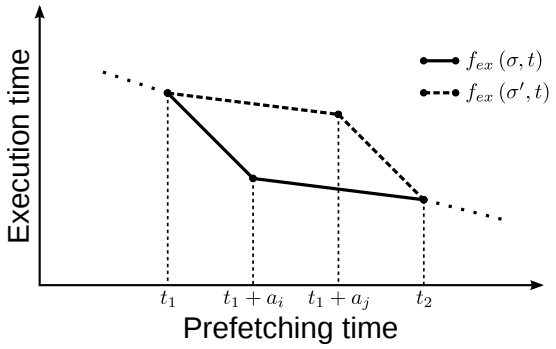


Figure 3.4: Branch prefetch ordering optimality condition proof.

time $f_{ex}(\sigma, t)$. Using this function, our problem consists in finding a permutation σ such that $f_{ex}(\sigma, t) \leq f_{ex}(\sigma', t)$ for any σ' and t.

In what follows we suppose that the branch prefetch order σ is $1, \ldots, n$. The time at which the branch at position k is fully prefetched, is denoted by $l_k = \sum_{i \leq k} a_i$. When the prefetching time is zero the execution time is independent of the branch prefetch order and is equal to $f_{ex}(., 0) = \sum_i p_i (a_i + b_i)$. The function $f_{ex}(\sigma, t)$ is a piecewise, strictly decreasing linear function. It is defined by the following relation for $k = 1, \ldots, n$ and $t \in [l_{k-1}, l_k[$:

$$f_{ex}(\sigma, t) = f_{ex}(., 0) - p_k (t - l_{k-1})$$

The next proposition describes the optimality condition of a branch ordering. It is used to build an algorithm that computes the branch ordering which minimizes the expected execution time.

Proposition 3.1. *In an optimal branch prefetch order σ we have $p_{\sigma(i)} > p_{\sigma(j)}$ for any i, j such that $i < j$.*

Proof. The proposition is proven by contradiction. Suppose that in an optimal order σ' the above condition is not satisfied. Thus, it exists two consecutive branches j and i, $j < i$, such that $p_j < p_i$. Let us inter-change the place of branches i and j and denote the new ordering σ. In Figure 3.4 are illustrated the expected execution time functions for both orderings. The derivative of $f_{ex}(\sigma, t)$ (equal to p_i) is larger then the derivative of $f_{ex}(\sigma', t)$ (equal to p_j) on interval $[t_1, t_1 + a_i]$, thus the first function decreases faster then the second one on this interval. We deduce that the expected execution time function is smaller for ordering σ then that for ordering σ', $f_{ex}(\sigma, t) \leq f_{ex}(\sigma', t)$. Our supposition that σ' is an optimal order is not valid. In consequence, the branch i in an optimal prefetching order should be ordered before the branch j.

□

Using the last proposition the optimal order of branch prefetch can be deduced. The branches are prefetched in decreasing order of their probabilities, that

is, a branch is entirely loaded before the next branch will start to be prefetched. We observe that the same solution method as in previous sub-section is obtained for both prefetching strategies: all-or-nothing and fractional.

When the available prefetching time follows a generic distribution, other then the uniform distribution we have supposed earlier, the minimization problem does not change. The prefetching time random variable is denote by T. When a random variable is used for prefetching time the objective function of the problem becomes a continuous random variable too $f_{ex}(\sigma, T)$. It is easy to verify that relation $f_{ex}(\sigma, T) \leq f_{ex}(\sigma', T)$ is equivalent to $f_{ex}(\sigma, t) \leq f_{ex}(\sigma', t)$ for any $t \in \mathbb{D}$. Thus, the obtained objective function is the same as the objective function we have studied earlier.

In the all-or-nothing prefetch strategy when the branch probabilities are equal, the order in which the branches are prefetched does not matter. So, the expected execution time minimization model is interesting when reliable estimates of branch probabilities are available. The same conclusion is true for the fractional prefetching strategy, i.e. it does not matter fractions of what branches are prefetched as long as something is prefetched.

3.4 Worst-case execution time

As in the previous section, we begin by investigating the fractional prefetching strategy, and then, we consider the all-or-nothing strategy for the worst-case execution time minimization problem. Before describing the proposed resolution methods we introduce a proposition which defines a basic particular case of the problem.

Proposition 3.2. *An n-way branching structure with parameters a_i and b_i for each $i \in 1, \ldots, n$ is given. The worst-case execution time of this branching structure is lower bounded by* $\max_i b_i$.

Proof. Suppose that the prefetching time t is sufficient to prefetch all the branches, i.e. $t \geq \sum_i a_i$. The worst-case execution time will be the maximum of branch execution times, $\max_i b_i$. Obviously that when less prefetching time is available the worst-case execution time will be at least $\max_i b_i$. □

In what follows, without loss of generality we suppose that relation $\max_i b_i < \min_i (a_i + b_i)$ is verified.

3.4.1 Fractional prefetch

As above, let us consider an n-way branching structure, and suppose that the available prefetching time is constant and equal to t. We look for optimal prefetching durations $0 \leq \alpha_i \leq a_i$, such that the worst-case execution time is minimal.

During the prefetching period (see Figure 3.3), branch i is prefetched for α_i time. If the branch i is executed, then the execution time will be equal to $a_i - \alpha_i + b_i$. Our goal is to minimize the worst-case execution time, thus the largest one of these terms. The problem can be stated as a mathematical program:

$$\begin{aligned} \text{Minimize} \quad & \max_i \left(a_i - \alpha_i + b_i\right) \\ \text{s.t.} \quad & \sum_i \alpha_i = t \\ & \alpha_i \in [0, a_i]\,,\ \forall i \end{aligned}$$

This formulation can be easily rewritten as a linear program:

$$\begin{aligned} \text{Minimize} \quad & \Gamma \\ \text{s.t.} \quad & a_i - \alpha_i + b_i \leq \Gamma,\ \forall i \\ & \sum_i \alpha_i = t \\ & \alpha_i \in [0, a_i]\,,\ \forall i \end{aligned}$$

The solution of the above linear program consists in prefetching the branches with the largest remaining execution times, as long as the prefetching time allows it. We introduce Algorithm 3.2 which computes optimal prefetch durations α_i for a fixed prefetching time t. The same procedure is applied when the available prefetching time is a random variable.

Algorithm 3.2 Worst-case execution time minimization algorithm, fractional prefetching strategy.

Input: Branches $1, \ldots, n$ with parameters a_i and b_i
Input: Available prefetching time t
Output: Prefetch durations α_i

1: Sort branches $i = 1, \ldots, n$ in decreasing order of $a_i + b_i$
2: Compute $\Delta_i = (a_i + b_i) - (a_{i+1} + b_{i+1})$ for any $i = 1, \ldots, n-1$
3: $\Delta_n = (a_n + b_n) - \max_i b_i$
4: $\alpha_i = 0$ for any $i = 1, \ldots, n$
5: $k = 0$
6: **while** $t > 0$ **and** $k \leq n$ **do**
7: $\quad c = \min\left(t, \Delta_k \cdot k\right)$
8: $\quad t = t - c$
9: $\quad \alpha_i = \alpha_i + c/k$ for any $i = 1, \ldots, k$
10: **end while**

3.4.2 All-or-nothing prefetch

In this sub-section, we aim to define an optimal branch prefetching order $\sigma \in \Pi(n)$ which minimizes the worst-case execution time. As we shall see further, it is not always possible to totally order the branches such that the worst-case execution time is minimal for any prefetching time. As in the previous section we suppose that the available prefetching time t follows an uniform distribution over the interval $\mathbb{D}$. Let $f_{wc} : \Pi(n) \times \mathbb{D} \to \mathbb{R}^+$ be a function that associates to a branch order $\sigma \in \Pi(n)$ and a prefetching time $t \in \mathbb{D}$ the branching structure's worst-case execution time $f_{wc}(\sigma, t)$.

In the sequel we suppose that the ordering σ defines a linear order, i.e. $1, \ldots, n$. We recall the definition of l_k, $l_k = \sum_{i \leq k} a_i$. For any $k = 1, \ldots, n$ and $t \in [l_{k-1}, l_k[$ the worst-case execution time function definition is:

$$f_{wc}(\sigma, t) = \max(\Lambda_k, a_k + b_k - t + l_{k-1}),$$

where $\Lambda_k = \begin{cases} \max_{i>k}(a_i + b_i) & \text{if } k < n, \\ \max_i b_i & \text{otherwise.} \end{cases}$

More formally the all-or-nothing prefetch with worst-case execution time minimization problem consists in finding a permutation of branches $\sigma \in \Pi(n)$ such that $f_{wc}(\sigma, t) \leq f_{wc}(\sigma', t)$ for any $\sigma' \in \Pi(n)$ and $t \in \mathbb{D}$. This problem can have instances for which the solution space is empty, that is to say the employed objective function does not totally order the branches. This result is proved in the next proposition.

Proposition 3.3. *For a given n-way branching structure an order $\sigma \in \Pi(n)$ that minimizes the worst-case execution time $f_{wc}(\sigma, t)$ for any $t \in \mathbb{D}$ cannot be always defined.*

Proof. To prove it, we provide a simple counterexample for which an order that minimizes $f_{wc}(\sigma, t)$ does not exist.

Suppose a 2-way branching structure, such that the relations $a_1 + b_1 > a_2 + b_2$ and $a_1 > a_2$ are verified. Refer to Figure 3.5 for an illustration. Two branch orders are possible: $\sigma_1 = \langle 1, 2 \rangle$ and $\sigma_2 = \langle 2, 1 \rangle$. It is easy to see that for $t \in [0, a_1 + b_1 - b_2[$ we have $f_{wc}(\sigma_1, t) \leq f_{wc}(\sigma_2, t)$ and for $t \in [a_1 + b_1 - b_2, a_1 + a_2]$ we have $f_{wc}(\sigma_1, t) \geq f_{wc}(\sigma_2, t)$. Thus, for this problem instance, worst-case execution time functions $f_{wc}(\sigma_1, t)$ and $f_{wc}(\sigma_2, t)$ cannot be compared. We conclude that, in the general case also, an order that minimizes the worst-case execution time is not always defined. □

Rather than attempting to compute a Pareto front, we modify the objective function as follows: we look for a branch prefetching order $\sigma \in \Pi(n)$, such that for any $\sigma' \in \Pi(n)$ we have $E[f_{wc}(\sigma, t)] \leq E[f_{wc}(\sigma', t)]$. We recall that the available prefetching time t is uniformly distributed over $\mathbb{D}$, thus the pdf is constant and

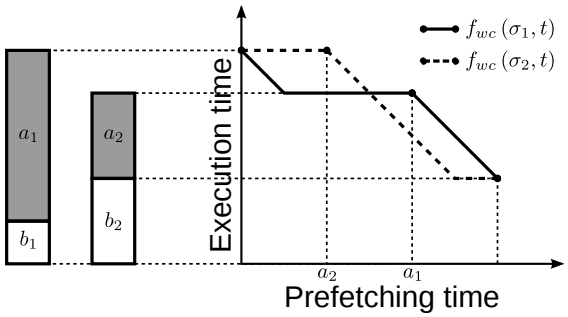


Figure 3.5: An example of branching structure for which a total order is not defined.

equal to $g(t) = 1/\sum_i a_i$ for any $t \in \mathbb{D}$. The mathematical expectation of the worst-case execution time is:

$$E[f_{wc}(\sigma, t)] = \int_{\mathbb{D}} f_{wc}(\sigma, t)\, g(t)\, dt = \frac{1}{\sum_i a_i} \int_{\mathbb{D}} f_{wc}(\sigma, t)\, dt$$

The minimization of the worst-case execution time expectation is equivalent to the minimization of the area of the region bounded by the worst-case execution time function. The integral of the worst-case execution time function over the range $[l_{k-1}, l_k[$ is equal to (for branch ordering $1, \ldots, n$):

$$\int_{l_{k-1}}^{l_k} f_{wc}(\sigma, t)\, dt = \Lambda_k a_k + \frac{1}{2} \max(0, a_k + b_k - \Lambda_k)^2$$

In what follows, we suppose that the branches are indexed in the decreasing order of $a_i + b_i$, that is $a_1 + b_1 \geq a_2 + b_2 \geq \ldots \geq a_n + b_n$.

Proposition 3.4. *Let σ be the optimal branch prefetching order. If in this order branches $p+1, p+2, \ldots, r$ are situated before the branch p, then their order does not matter.*

Proof. Since $a_p + b_p$ is greater than or equal to $a_{p+1} + b_{p+1}, \ldots, a_r + b_r$ the worst-case execution time $f_{wc}(\sigma, t)$ during the prefetch of branches $p+1, \ldots, r$ is equal to $a_p + b_p$. Refer to Figure 3.6 for an illustration. Therefore, the integral of $f_{wc}(\sigma, t)$, over the interval when the branches $p+1, \ldots, r$ are prefetched, is constant and does not depend on their order (the grey rectangle in the illustration).

□

Proposition 3.5. *If σ is an optimal branch prefetching order, then it has the following form: $\sigma = \langle r, \ldots, 1, \sigma' \rangle$, $r \geq 1$, where σ' is an optimal order over the branches $r+1 \ldots n$.*

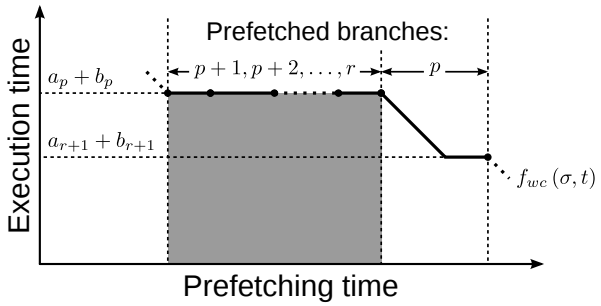


Figure 3.6: An example of branch ordering where the order of branches $p+1, p+2, \ldots, r$ does not change the worst-case execution time expectation (used in the proof of Proposition 3.4).

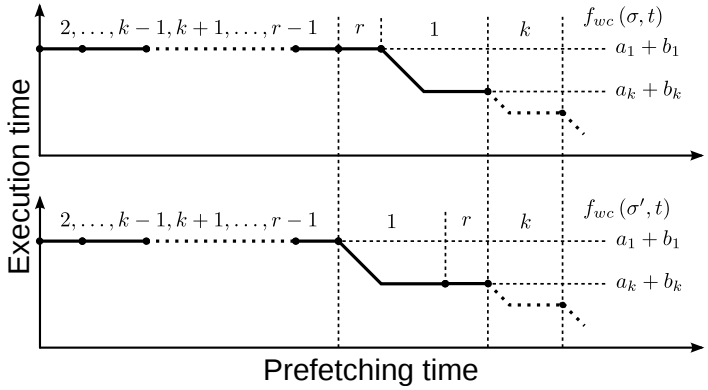


Figure 3.7: Illustration of the contradiction from Proposition 3.5.

Proof. Let r be the branch with the largest index ordered before the branch 1 in σ, that is, r is the branch with the lowest $a_i + b_i$ ordered before the branch 1. Suppose that a branch k, $k \in [2, r-1]$, is ordered after the branch 1. By interchanging branch r with 1 (see Figure 3.7) we obtain a new subset suborder that is strictly better than the initial order σ, which is in contradiction with the initial hypothesis which states that σ is an optimal order.

In the same manner, the proof is generalized to any sub-set of the branches in place of only one branch k. Also, we can state that the optimal sub-order σ' satisfies this proposition recurrently. □

For solving the all-or-nothing prefetch problem with worst-case execution time expectation minimization we introduce an optimal, polynomial time algorithm. This algorithm is based on the shortest path computation in a specific graph G that we introduce in the sequel. In the construction of the graph G we consider the result obtained in Proposition 3.4 and Proposition 3.5.

Definition 3.1. Let $G = (V, E, c)$ be a directed graph, where V is a set of nodes, E, a set of edges and $c : E \to \mathbb{R}$, a cost function that assigns a real, non-negative

number to each edge of the graph. The graph G contains $n+1$ nodes numbered from 0 to n. The meaning of the node i is that the branches $1, \ldots, i$ have been prefetched. For any i and j, the graph contains the edge (i, j) if and only if $i < j$. The value associated by the cost function c to the edge (i, j) is equal to the integral of function $f_{wc}(\sigma, t)$ over the period of time when the branch order $j, j-1, \ldots, i+1$ is prefetched, taking into account that branches $1, \ldots, i$ have been already prefetched.

An example of such a graph is presented in Figure 3.8. It corresponds to the graph built for a 4-way branching structure. In the illustration we shorten the integral notation and $\int \langle i_1, \ldots, i_m \rangle$ denotes the integral of the worst-case execution time function over the period of time when the ordering $i_1, \ldots, i_m$ is prefetched.

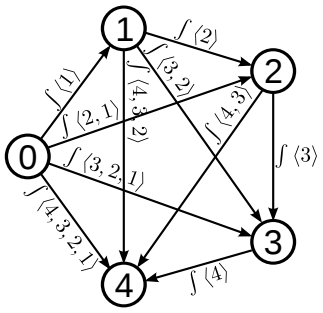


Figure 3.8: An example of graph G for a 4-way branching structure.

Let $P = \langle i_1 = 0, i_2, \ldots, i_p = n \rangle$ be a path from node 0 to node n in the graph G. The branch prefetching order that corresponds to the path P is built in the following manner: we begin by an empty order $\sigma = \emptyset$, for every $k = 2, \ldots, p$, the partial order $\langle i_k, i_k - 1, \ldots, i_{k-1} + 1 \rangle$ is appended to the end of σ, finally, σ will be the branch prefetching order that corresponds to path P.

Proposition 3.6. *Let $P = \langle i_1 = 0, i_2, \ldots, i_p = n \rangle$ be a path from node 0 to node n in the graph G and σ be the branch prefetching order that corresponds to P. Then, the cost of the path P is equal to the value of the integral of $f(\sigma, t)$ over $\mathbb{D}$, that is $\sum_{k=2}^{p} c(i_{k-1}, i_k) = \int_{\mathbb{D}} f_{wc}(\sigma, t)\, dt$.*

Proof. The proof of this proposition relies on the following transformations:

$$\sum_{k=2}^{p} c(i_{k-1}, i_k) = \sum_{k=2}^{p} \int_{l_{i_{k-1}}}^{l_{i_k}} f_{wc}(\sigma, t)\, dt = \int_{l_{i_1}}^{l_{i_p}} f_{wc}(\sigma, t)\, dt = \int_{l_0}^{l_n} f_{wc}(\sigma, t)\, dt$$

As $\mathbb{D} = [l_0, l_n]$, the last equality proves the proposition. □

The next proposition proves that the shortest path in the graph G corresponds to the optimal branch prefetching order.

Proposition 3.7. *Let $G = (V, E, c)$ be a graph built as described in Definition 3.1, and, let $P = \langle i_1 = 0, i_2, \ldots, i_p = n \rangle$ be the shortest path from node 0 to node n in this graph. Then, the branch prefetching order σ that corresponds to path P is an optimal one.*

Proof. From the definition of the graph G and the propositions 3.4, 3.5, the set of all possible paths, from node 0 to node n, covers the set of all possible branch orders $\Pi(n)$. Since the values of a path and its corresponding branch prefetching order are the same, a shortest path P corresponds to a minimal valued branch prefetching order σ. □

The procedure of finding optimal branch prefetching order described above can further be simplified. The graph G has a special structure (it contains only edges (i, j) for $i < j$) and it is possible to bypass its construction. Using this property we introduce a dynamic programming algorithm in which we make use of graph's special structure for computing the optimal prefetching order. We introduce Algorithm 3.3 for finding the optimal branch prefetching order. During the i-th iteration of the algorithm first loop the optimal prefetch order for branches $1, \ldots, i$ is computed.

Algorithm 3.3 Dynamic programming algorithm for finding the branch prefetching order which minimizes the worst-case execution time expectation.

Output: σ_n, c_n - optimal branch prefetching order and cost

1: $c_0 = 0$ and $c_i = \infty$ for any $i = 1, \ldots, n$
2: $\sigma_i = \emptyset$ for any $i = 0, \ldots, n$
3: **for** $i = 1$ **to** n **do**
4: **for** $j = 0$ **to** $i - 1$ **do**
5: $\sigma = \sigma_j + \langle i, i-1, \ldots, j+1 \rangle$ {Here the sign "+" denotes the concatenation of two orderings}
6: $c = \int_{l_j}^{l_i} f(\sigma, t)\, dt$
7: **if** $c_i > c_j + c$ **then**
8: $c_i = c$
9: $\sigma_i = \sigma$
10: **end if**
11: **end for**
12: **end for**

Now, let us suppose that the available prefetching time is a continuous random variable with pdf $g(t)$. The worst-case execution time expectation (the objective function we search to minimize) becomes:

$$E[f_{wc}(\sigma, t)] = \int_{\mathbb{D}} f_{wc}(\sigma, t)\, g(t)\, dt$$

It is easy to prove that propositions 3.4 and 3.5 remain valid when this objective function is used. An optimal branch prefetching order when the prefetching time is a random variable is found using Algorithm 3.3. A single modification of the algorithm is required: instead of $c = \int_{l_j}^{l_i} f_{wc}(\sigma, t)\, dt$ in the line 6 relation $c = \int_{l_j}^{l_i} f_{wc}(\sigma, t)\, g(t)\, dt$ shall be used.

3.5 Conclusions

In this chapter we examined the problem of speculative data prefetching in dataflow applications. Although the dataflow applications were restricted to a single n-way branching structure we obtained interesting results. When the optimization objective was the mathematical expectation of the execution time an optimum branch prefetching order was defined, which structure was furthermore independent on the available prefetching time. In the case of worst-case execution time objective, a polynomial procedure for finding optimal branch prefetching order was introduced.

Chapter 4

Task ordering and memory management

The tasks of a dataflow application, executed on a clustered multi-core processor, are often using some common data (shared code or constants for example). These data can be stored, or cached, in the cluster shared memory for later reuse. For performance issues the same data objects accessed by several tasks should be reused as much as possible. In a traditional caching scheme, a cache replacement policy decides either to keep or to discard a memory block. The cache policy is prescribed in memory system hardware and the application which runs cannot influence it. Contrary to this, the cluster memory in the processor architecture we use can be seen as a software-controlled scratchpad memory. Thus, one can decide in software what memory blocks to store for future reuse.

Suppose given an APG (Acyclic Precedence Graph) associated to a dataflow application which is executed on a cluster of the multi-core processor. As stated earlier, the tasks are executed in two steps: data loading and execution. Furthermore, we suppose that the data loading durations can change in function of their order. This variation is due to the reuse of external memory data which were previously stored in the cluster memory. That is to say, the duration of data loading operations for two successively executed tasks which are using common data is potentially smaller. Certainly, the order in which the tasks are executed influences the quantity of possible common data. In this chapter we introduce the *task ordering and memory management problem*, which tries to maximize the data reuse by optimally ordering tasks. Because of a single external memory access channel the data loading operations should be serialized. We suppose that there is an unlimited number of processing cores for task execution, in this way only the data reuse aspect is considered. As we shall see later, the practical relevance of the task ordering and memory management problem is not limited to its direct use (optimization of memory accesses) but also to the estimation of the degree of parallelism for an application.

A part of the results described in this chapter have been presented at CPAIOR 2010 workshop on Combinatorial Optimization for Embedded System Design and will appear in the Proceedings of the 17th Annual International Computing and Combinatorics Conference [25].

This chapter is organized as follows. A formal definition of the problem and a complexity result are given in Section 4.1, in Section 4.2 several applications of this optimization problem are described, followed by a survey on the existing work in Section 4.3. In Section 4.4 the issue of internal memory management for a fixed task calculation order is studied, and then, an exact branch-and-bound algorithm and two heuristics are described respectively in Section 4.5 and Section 4.6. At the end a computational study, Section 4.7, and some concluding remarks are given.

4.1 Problem formulation and complexity

In this section some useful definitions used throughout this chapter are introduced and a formal definition for the task ordering and memory management problem together with its $\mathcal{NP}$-hard complexity proof are given.

An application is a set of tasks. Each task uses external memory data which needs to be fetched into an internal memory before the task execution can start. The smallest unit of data is called an *input*, hereafter we use the terms of external memory data and input equivalently. Let $A = (S, E, \delta)$ denote an application where S are application's tasks, E represent the set of inputs used by the application and $\delta : S \to 2^E$ is a function that associates to any task s, $s \in S$, a set of inputs $\delta(s)$ needed for its calculation. When the intersection of δ function for two tasks is not empty then these tasks are using common data. We also suppose that there are no precedence relations between application tasks. Although this model seems to be limited the methods described further can be easily modified to take into account the precedence constraints. As we shall see later, an important motivation of studying this problem is to estimate the achievable degree of parallelism for an application. In this context, the ordering problem without precedence relations can be interpreted as a coarse-grained view of application task graphs where the set of tasks producing an outcome are grouped into a single task.

Let $\gamma : S \to 2^E$ be a function that associates a set of inputs $\gamma(s)$ to any task s, $s \in S$. The set of inputs $\gamma(s)$ gives the internal memory state at the beginning of task s calculation. It is evident that for any task s, $\gamma(s)$ contains at least all the inputs used by this task, $\delta(s) \subseteq \gamma(s)$. Remaining inputs that do not belong to $\delta(s)$, come from data reuse. *Data reuse* is the process of reusing inputs already present in memory, originating from previously calculated tasks. Throughout this paper we suppose that available internal memory size is equal to C, thus condition $|\delta(s)| \leq |\gamma(s)| \leq C$ must be verified for any $s \in S$. Without

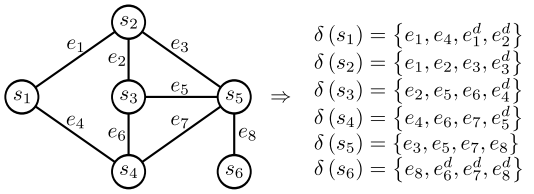


Figure 4.1: Illustration of graph transformations used to obtain an instance of our problem.

loss of generality we suppose that the total number of inputs is larger than the internal memory size, i.e. $|E| > C$.

The *task ordering and memory management problem* consists in finding a permutation π of tasks S such that the number of external memory accesses, given by cost function (4.1) is minimized.

$$\left|\gamma\left(s_{\pi(1)}\right)\right| + \sum_{i=2}^{|S|} \left|\gamma\left(s_{\pi(i)}\right) \setminus \gamma\left(s_{\pi(i-1)}\right)\right| \tag{4.1}$$

Proposition 4.1. *Task ordering and memory management problem is $\mathcal{NP}$-hard.*

Proof. Let $G = (V, A)$ be an arbitrary non-oriented graph. The problem of existence of a Hamiltonian path in graph G is $\mathcal{NP}$-complete [42, p.199]. We proceed by restriction, by showing that any instance of Hamiltonian path existence problem can be reduced to a special case of our problem. The following transformations are used.

Each vertex of graph G becomes a task in our problem $S = V$ (see Figure 4.1 for an illustration). To each edge of graph G an input is associated. The internal memory size equals to the maximum degree of graph vertices, $C = \max_{s \in V} \deg(s)$. For any task $s \in S$, $\delta(s)$ contains the set of input-edges that are adjacent to s in the graph, plus a set of $C - \deg(s)$ dummy inputs specific for each task. In this way $\gamma(s) = \delta(s)$ and $|\gamma(s) \cap \gamma(s')| = 1$ for any pair of adjacent task $s, s' \in S$.

The problem obtained in such a way is a special case of our problem. It has a solution of cost $n \cdot C - n + 1$ (n being the number of tasks) if and only if it exists a Hamiltonian path in graph G. As far as the question of existence of a Hamiltonian path in graph G is $\mathcal{NP}$-complete, our problem is $\mathcal{NP}$-hard. □

4.2 Applications

The task ordering and memory management problem can be directly employed to optimize the sequential execution of an application. An algorithm for solving this problem will try to order tasks that are using common data close to each other in order to maximize the data reuse. The lack of support for task precedence

relations does not diminish the interest in this method. Actually, many real life applications are made up of several, smaller tasks that are accomplishing different objectives and are using some common data inputs. We shall note that the task ordering and memory management problem is not limited to data load optimization but can be also applied to optimize the code load. Let's imagine that the application's tasks are using common pieces of code. It will be interesting to execute two tasks that are using the same piece of code one after another. In doing so, the common piece of code is loaded only once from the external memory and it is used two times by each task.

Another use of the task scheduling and memory management problem is to model a series of problems related to efficient memory bandwidth management in parallel processor architectures. In particular, we are interested in estimating the memory bandwidth λ required for the sequential execution of a parallel application so as to estimate the number of tasks which may execute in parallel with respect to an external memory bandwidth Λ constraint. In the best case, when the application parallelism is not an impediment, the maximum number of tasks which can be executed in parallel is limited to Λ/λ. This estimation is called the *achievable degree of parallelism*. We are going to give an example. Consider an application which in average fetches 10 units of data per second. The memory bandwidth of the application is $\lambda = 10^{units/sec.}$. Suppose that the available external memory bandwidth is $\Lambda = 100^{units/sec.}$, then in the best case 10 application tasks can be executed in parallel with respect to the hardware bandwidth constraint. By maximizing the data reuse in an application (via task ordering) we obtain less conservative estimates of the achievable degree of parallelism (λ diminishes). In a compiler chain proceeding by parallelism reduction (via task fusion [46]), such an estimation can be used to fix an appropriate target for the degree of parallelism. This estimation can also be used within an Algorithm-Architecture Adequation framework to perform an initial assessment.

4.3 Related work

Previous work related to our problem is quite rare. In the "compilers" research community similar problematics can be found. Actually, they try to minimize the execution time for an application by augmenting the reuse of data. Approximate resolution methods are proposed. Compared to these studies, our problem considers the task graph at a much larger level of granularity.

It appears that Ding and Kennedy [32] were the first to study a problem relatively close to ours. In their work they intend to reorder program instructions so as to improve program data access locality. The goal of locality minimization is to reduce the distance between instructions that are using the same data, thereby to augment data reuse. However, instead of using an optimal cache management policy for calculating the distances, a sufficient-only condition is used. The latter

requires that the *time distance*, which is the number of instructions between two accesses, is bounded by a constant in order to ensure a cache hit. This problem is modeled as a bandwidth-minimization problem. In their paper, the authors describe a tool that reorders program instructions and provide experimental results for a set of benchmark programs.

Some earlier works, [109, 82], describe methodologies for optimizing data reuse in program loops. In a series of two papers [31, 33] describe two program transformations: *locality grouping*, which reorders program data accesses in order to improve data temporal reuse, and *dynamic data packing*, which consists in reorganizing data layout so as to improve data spatial reuse. Other papers [88, 102] describe similar approaches of data locality improvement and provide benchmark analysis.

In the combinatorial optimization domain one can mention the sequence dependent setup times (SDST) scheduling problems. The SDST one machine scheduling problems have similarities with the task ordering and memory management problem as in both cases the execution time for a task depends on the history. In the SDST it depends solely on previously executed task, in contrast to our problem where the "execution time" (in terms of number of memory accesses) of a task depends on the order of all the tasks executed so far. For a survey of SDST scheduling problems refer to [3].

4.4 Memory management for a fixed sequence of tasks

We start by investigating the internal memory data management separately from the task ordering as it is an important issue of our problem. Let $A = (S, E, \delta)$ be an application. Suppose that a permutation of tasks $s_1, \ldots, s_n$ is given. We recall that for any task $s \in S$, $\delta(s)$ are the inputs that must be loaded into the internal memory before s starts. It is obvious that the duration of loading inputs $\delta(s)$ depends on the position of task s in this sequence and on the content of the internal memory. Namely, on already loaded inputs originating from data reuse process. We are interested to find the internal memory states $\gamma(s)$ for any s, $s \in S$, such that the number of external memory accesses (4.1) is minimal.

It can be easily seen that internal memory size C plays an important role in problem modeling. In the following paragraph we introduce a special case for which the optimal data management is trivial. When the internal memory size is sufficient to store all the inputs (i.e. $|E| \leq C$) then the minimal solution has a value of $|E|$. The optimal data management consists in simply loading new inputs into the internal memory without dropping them out.

For solving the memory management problem, we introduce an incremental[1]

[1]The incremental formulation is useful in the branch-and-bound procedure we introduce in

algorithm, see Algorithm 4.1, which updates the optimal memory states for a sequence of tasks $s_1, \ldots, s_{p-1}$ with the data reuse induced by a task s_p added to the end of this sequence. The complexity of this algorithm is $O(p \cdot m)$, where m is the number of inputs. The algorithm is based on the principle used in optimal cache replacement algorithm proposed by [7]. In our context[2], it may be informally stated as follows: "when a memory location is needed for a given input and the internal memory is full, free space should be obtained by dropping out the input which is to be used in the farthest future".

The next proposition describes the relation between the internal memory states of two solutions. One solution being longer by a task, which is situated at the end.

Proposition 4.2. *Let $s_1, \ldots, s_{p-1}$ be a task ordering and $s_1, \ldots, s_p$ an extension of this ordering with a task s_p. The optimal memory states of ordering $s_1, \ldots, s_{p-1}$ are $\gamma_1, \ldots, \gamma_{p-1}$ and of ordering $s_1, \ldots, s_p$ are $\gamma'_1, \ldots, \gamma'_p$. The following relation is valid for any $k \in 1, \ldots, p-1$:*

$$\gamma'_k = \gamma_k \cup \epsilon_k$$

$$\epsilon_k \subseteq \left(\delta_p \cap \bigcup_{i=1}^{k-1} \delta_i \right) \setminus \bigcup_{i=k}^{p-1} \delta_i$$

Proof. Let e be an input which is stored for reuse during task s_k execution in the initial ordering, $e \in \gamma_k$ and $e \notin \delta_k$. The input e must be used by a task executed earlier and later then s_k, we can write:

$$e \in A = \left(\bigcup_{i=k+1}^{p-1} \delta_i \cap \bigcup_{i=1}^{k-1} \delta_i \right) \setminus \delta_k$$

Two cases can be distinguished:

1. The internal memory permits to store in γ_k only the earliest $C - |\delta_k|$ inputs from A. In this case, when the task s_p is added to the end of the ordering the memory state γ_k remains unchanged, $\gamma'_k = \gamma_k$.

2. There is enough memory space to store all the inputs from A. When the task s_p is added to the end of the ordering then several inputs from δ_p are added to the memory state of task s_k. Thus we have:

$$\epsilon_p \subseteq \left(\bigcup_{i=k+1}^{p} \delta_i \cap \bigcup_{i=1}^{k-1} \delta_i \right) \setminus \delta_k \setminus A = \left(\delta_p \cap \bigcup_{i=1}^{k-1} \delta_i \right) \setminus \bigcup_{i=k}^{p-1} \delta_i$$

---

the next section, where it is necessary to recalculate the data reuse induced only by the last task.

[2]We do not use the algorithm described in Belady's paper, because in their model at each step only a single memory location is loaded. Contrary to our model, where at each step we can load more than one input.

□

Algorithm 4.1 Incremental internal memory management algorithm (for the sake of simplicity we consider $\gamma_i = \gamma(s_i)$ and $\delta_i = \delta(s_i)$).

Input: $s_1, \ldots, s_{p-1}$ - task ordering with already computed memory states $\gamma_1, \ldots, \gamma_{p-1}$
Input: s_p - new task to be added to the end of the ordering
Output: $\gamma'_1, \ldots, \gamma'_p$ - updated memory states

1: $k = p - 1$
2: **while** $k > 1$ **and** $|\gamma_k| < C$ **do** {Find last task with full internal memory, if it exists}
3: $k = k - 1$
4: **end while**
5: $\gamma'_i = \gamma_i$ for $i = 1, \ldots, k$
6: **for** $i = k + 1$ **to** $p - 1$ **do**
7: $L = (\delta_p \setminus \gamma_i) \cap \gamma'_{i-1}$ {Potential inputs to reuse}
8: $l = \min(|L|, C - |\gamma_i|)$
9: $\gamma'_i = \gamma_i \cup$ {First l elements of L according to a linear order}
10: **end for**
11: $\gamma'_p = \delta_p$

In order to find the optimal memory management for a sequence of tasks $s_1, \ldots, s_n$ the algorithm is executed $n - 1$ times, each time adding task s_i, $i = 2 \ldots n$, to the end of previously computed sequence. The algorithm is depicted in Algorithm 4.2. The global complexity of this procedure is $O(n^2 \cdot m)$. The minimal number of external memory accesses can be found using expression (4.1) from the calculated memory states.

From a implementation point of view the algorithm complexity can be decreased. The optimization consists in using a more efficient structure for representing sets. In the above complexity result we have considered that the set operations cost $O(m)$, which corresponds to a non optimal set representation. When the number of inputs is not too large[3] the sets can be represented as m-bit bit arrays for which the set operations are done in constant time $O(1)$. In other cases balanced binary trees can be used and instead of $O(m)$ the set operations will have a complexity of $O(\log_2 m)$.

The optimal internal memory data management based on the calculated memory states is easily found. For first task s_1 all the inputs $\gamma(s_1)$ are loaded into the internal memory. For next tasks s_i, $i \geq 2$, before the execution of task s_i starts, inputs $\gamma(s_i) \setminus \gamma(s_{i-1})$ are loaded into the internal memory in place of inputs $\gamma(s_{i-1}) \setminus \gamma(s_i)$.

[3]Less than the word width of PCs, which usually does not exceed 64 bits.

Algorithm 4.2 Internal memory management algorithm.

Input: $s_1, \ldots, s_n$ - task ordering
Output: $\gamma(s_1), \ldots, \gamma(s_n)$ - optimal internal memory states
1: $\gamma(s_1) = \delta(s_1)$
2: **for** $k = 2$ **to** n **do**
3: Compute memory states $\gamma(s_1), \ldots, \gamma(s_k)$ using Algorithm 4.1
4: **end for**

4.5 Branch-and-bound

The fact that the task ordering and memory management problem is $\mathcal{NP}$-hard legitimates the use of a branch-and-bound procedure (see [20, 69] for a detailed description of the method) for solving it. One practical utility of an exact resolution approach is for finding optimal solutions which could be used to evaluate heuristic methods, because obtaining tight global lower bounds for a particular case of this problem is already a difficult task [3].

In what follows we describe the proposed branch-and-bound algorithm as well as each of the used components. The branch-and-bound algorithm starts with an empty sequence of tasks. At each branching decision it adds to the end of this sequence a new task from the set of not yet ordered ones. A leaf is obtained when all the tasks are ordered. Lower bounds as well as a dominance relation are used in algorithm in order to reduce the search space.

Before describing the branch-and-bound procedure we introduce some useful definitions. Let us denote by (I, π) a *permutation of tasks* or *task ordering*, where $I \subseteq S$ is a set of tasks and $\pi : \{1, \ldots, |I|\} \rightarrow \{1, \ldots, |I|\}$ is a permutation of tasks of I. In the case when $I = S$ the task permutation is called a *complete permutation*, when it is not - a *partial permutation.*

A triplet $\omega = (I, \pi, \gamma)$, where (I, π) is a task ordering and $\gamma : I \rightarrow 2^E$ are optimal memory states at the beginning of each task calculation, is a *solution.* When the solution contains a partial ordering, we call it *a partial solution*, and when not *a complete solution.* Let Ω be the set of all possible partial and complete solutions.

A task ordering (I', π') *begins with* the partial task ordering (I, π) if the following relations are satisfied: $I' = I \cup K$ for $K \subseteq S \setminus I$ and $\pi'(i) = \pi(i)$ for any $s_i \in I$. Respectively, solution (I', π', γ') begins with the partial solution (I, π, γ) when (I', π') begins with (I, π).

Let $f : \Omega \rightarrow \mathbb{N}^+$ be a bijection that assigns to any solution $\omega \in \Omega$ a positive integer $f(\omega)$, where $f(\omega)$ represents the minimum number of external memory accesses of solution ω, calculated using Algorithm 4.2.

In what follows we describe each of the components used in the branch-and-bound algorithm.

4.5.1 Branching rule

A partial or a complete task ordering (I, π_I) is respectively a node or a leaf of the search tree, here I denotes the set of already ordered tasks and π_I is a permutation of tasks I. The root node of the search tree is $(\emptyset, \pi_\emptyset)$ and it corresponds to an empty task ordering. At a node (I, π_I) of the search tree, for each task $s \in S \setminus I$ the nodes $\left(I \cup \{s\}, \pi_{I \cup \{s\}}\right)$ beginning with (I, π) are created. Thus, branching from node (I, π_I) creates a number of $|S \setminus I|$ new nodes.

4.5.2 Lower bounds

A *lower bound* is a bijection $g : \Omega \to \mathbb{N}^+$ that assigns a positive integer $g(\omega)$ to a solution $\omega \in \Omega$. Let ω' be a complete solution that begins with ω. The minimum number of external memory accesses of solution ω' must verify $g(\omega) \leq f(\omega')$, thus we say that $g(\omega)$ is a lower bound to solutions that begin with ω.

Before describing the proposed lower bounds, we introduce a proposition that relates the costs of two solutions ω and ω', where ω' begins with ω. To decrease the wordiness in what follows, we suppose that task ordering is linear, i.e. for a permutation (I, π), $\pi(i) = i$ for any $s_i \in I$. Also, we make the following abbreviations: $\gamma(s_i) = \gamma_i$ and $\delta(s_i) = \delta_i$.

Proposition 4.3. *Let $\omega = (I, \pi, \gamma)$ and $\omega' = (I', \pi', \gamma')$ be two solutions such that ω' begins with ω. We have:*

$$f(\omega') = f(\omega) + \sum_{i=|I|+1}^{|I'|} \left|\gamma'_i \setminus \gamma'_{i-1}\right|$$

Proof. Suppose that $|I| = p$ and $|I'| = n$.

Using expression (4.1) the cost functions $f(\omega)$ and $f(\omega')$ can be written:

$$f(\omega) = |\gamma_1| + \sum_{i=2}^{p} |\gamma_i \setminus \gamma_{i-1}|$$

$$f(\omega') = |\gamma'_1| + \sum_{i=2}^{p} \left|\gamma'_i \setminus \gamma'_{i-1}\right| + \sum_{i=p+1}^{n} \left|\gamma'_i \setminus \gamma'_{i-1}\right|$$

Initially, when there are no data to reuse: $\gamma'_1 = \gamma_1 = \delta_1$. If we show that $\left|\gamma'_k \setminus \gamma'_{k-1}\right| = |\gamma_k \setminus \gamma_{k-1}|$ for any $k \in 2, \ldots, p$, then the proposition will be demonstrated.

The memory state γ'_k is a superset of γ_k (see Proposition 4.2) for any $k \in 2, \ldots, p$ and the excess $\epsilon_k = \gamma'_k \setminus \gamma_k$ contains inputs from γ'_{k-1} (i.e. $\gamma'_k = \gamma_k \cup \epsilon_k$ and $\epsilon_k \subseteq \gamma'_{k-1}$). Furthermore, the excess ϵ_{k-1} of memory state γ'_{k-1} contains only

inputs from the tasks $s_{p+1}, \ldots, s_n$. The last fact is a generalization of Proposition 4.2. Using the above relations we have:

$$\gamma'_k \setminus \gamma'_{k-1} = \gamma_k \setminus \gamma'_{k-1} \cup \underbrace{\epsilon_k \setminus \gamma'_{k-1}}_{\emptyset} = \underbrace{\gamma_k \setminus \epsilon_{k-1}}_{\gamma_k} \setminus \gamma_{k-1}$$

From the last relation $\left|\gamma'_k \setminus \gamma'_{k-1}\right| = |\gamma_k \setminus \gamma_{k-1}|$ for any $k \in 2, \ldots, p$, thus the proposition is proved. □

Let $\omega = (I, \pi, \gamma)$ be a partial solution associated to the node (I, π) of the search tree. Two lower bounds are proposed for computing an underestimated value of current partial solution.

The following lemma is stated without proof.

Lemma 4.1. *If $a_1, \ldots, a_n$ are sets, then $\left(\bigcup_{i=2}^{n} a_i\right) \setminus a_1 \subseteq \bigcup_{i=2}^{n} (a_i \setminus a_{i-1})$.*

In the first lower bound, for a given partial solution the internal memory capacity constraint is relaxed for the not yet ordered tasks.

Proposition 4.4. *Let $\omega = (I, \pi, \gamma)$ be a solution. Suppose that $\omega' = (I', \pi', \gamma')$ is a new solution that begins with ω, where $I' = I \cup \{s_d\}$ and s_d is a dummy task containing all the inputs of not yet ordered tasks, i.e. $\delta(s_d) = \bigcup_{s \in S \setminus I} \delta(s)$. Then $g_1(\omega) = f(\omega')$ is a lower bound of solution ω.*

Proof. Let $\bar{\omega} = (S, \bar{\pi}, \bar{\gamma})$ be the optimal solution that begins with ω. Suppose that $|I| = p$ and $|S| = n$. Using Proposition 4.3 we can write:

$$f(\bar{\omega}) = f(\omega) + \sum_{i=p+1}^{n} |\bar{\gamma}_i \setminus \bar{\gamma}_{i-1}| \tag{4.2}$$

Let us examine the rightmost term of this expression. Applying consecutively set property $|a| + |b| \geq |a \cup b|$ and Lemma 4.1 we obtain:

$$\sum_{i=p+1}^{n} |\bar{\gamma}_i \setminus \bar{\gamma}_{i-1}| \geq \left| \bigcup_{i=p+1}^{n} (\bar{\gamma}_i \setminus \bar{\gamma}_{i-1}) \right| \geq \left| \left(\bigcup_{i=p+1}^{n} \bar{\gamma}_i \right) \setminus \bar{\gamma}_p \right|$$

It is evident that $\bigcup_{i=p+1}^{n} \bar{\gamma}_i = \bigcup_{i=p+1}^{n} \delta_i$ because $\bar{\gamma}_i \subseteq \bigcup_{j=i}^{n} \delta_j$ for any $i = p + 1, \ldots, n$. Using the last relation we have:

$$\left| \left(\bigcup_{i=p+1}^{n} \bar{\gamma}_i \right) \setminus \bar{\gamma}_p \right| = \left| \left(\bigcup_{i=p+1}^{n} \delta_i \right) \setminus \bar{\gamma}_p \right|$$

If the last inequality is introduced in relation (4.2) a lower bound is obtained (the right-hand side of (4.3)):

$$f(\bar{\omega}) \geq f(\omega) + \left| \left(\bigcup_{i=p+1}^{n} \delta_i \right) \setminus \bar{\gamma}_p \right| \tag{4.3}$$

The main inconvenient is that the optimal internal memory state $\bar{\gamma}_p$ must be known. In order to avoid the computation of an optimal solution $\bar{\gamma}_p$ is replaced by γ'_p which is the internal memory state in a task ordering ω' composed of ω extended by a dummy task s_d. The inputs used by the dummy task are $\delta(s_d) = \bigcup_{i=p+1}^{n} \delta_i$. In this way, the following lower bound is obtained (and the proposition is proven):

$$g_1(\omega) = f(\omega) + \left|\delta(s_d) \setminus \gamma'_p\right| = f(\omega')$$

□

In the following proposition a computationally cheaper lower bound, but weaker than the previous one is described.

Proposition 4.5 (Lower bound 2). *Let $\omega = (I, \pi, \gamma)$ be a solution. Then $g_2(\omega)$ is a lower bound of ω:*

$$g_2(\omega) = f(\omega) + \left|\bigcup_{s \in S \setminus I} \delta(s) \setminus \delta\left(s_{|I|}\right)\right| - B$$

where

$$B = \min\left(C - \left|\delta\left(s_{|I|}\right)\right|, \left|\left(\bigcup_{s \in S \setminus I} \delta(s) \cap \bigcup_{s' \in I \setminus \left\{s_{|I|}\right\}} \delta(s')\right) \setminus \delta\left(s_{|I|}\right)\right|\right)$$

Proof. Let $\bar{\omega} = (S, \bar{\pi}, \bar{\gamma})$ be the optimal solution that begins with ω and we suppose that $|I| = p$ and $|S| = n$. Resuming the proof of Proposition 4.4 from expression (4.3) we have:

$$f(\bar{\omega}) \geq f(\omega) + \left|\left(\bigcup_{i=p+1}^{n} \delta_i\right) \setminus \bar{\gamma}_p\right|$$

For the optimal internal memory state $\bar{\gamma}_p$ we have $\bar{\gamma}_p = \delta_p \cup \epsilon_p$ where ϵ_p represents the data to reuse and $\delta_p \cap \epsilon_p = \emptyset$ is verified. Using this expression and the set property $|a \setminus (b \cup c)| = |a \setminus b| - |a \cap c|$ for $b \cap c = \emptyset$ we obtain:

$$\left|\left(\bigcup_{i=p+1}^{n} \delta_i\right) \setminus \bar{\gamma}_p\right| = \left|\left(\bigcup_{i=p+1}^{n} \delta_i\right) \setminus \delta_p\right| - \left|\left(\bigcup_{i=p+1}^{n} \delta_i\right) \cap \epsilon_p\right| \geq \left|\left(\bigcup_{i=p+1}^{n} \delta_i\right) \setminus \delta_p\right| - |\epsilon_p|$$

Subsequently, for the size of the set ϵ_p we have:

1. The data to reuse that is transfered during task s_p execution is limited by the internal memory size C:

$$|\epsilon_p| \leq C - |\delta_p|$$

2. From the definition of ϵ_p we have $\epsilon_p \subseteq \left(\bigcup_{i=1}^{p-1} \delta_i \cap \bigcup_{i=p+1}^{n} \delta_i\right) \setminus \delta_p$, thus:

$$|\epsilon_p| \leq \left| \left(\bigcup_{i=1}^{p-1} \delta_i \cap \bigcup_{i=p+1}^{n} \delta_i \right) \setminus \delta_p \right|$$

Combining together the above expressions we obtain the bound $g_2(\omega)$:

$$f(\bar{\omega}) \geq f(\omega) + \left| \left(\bigcup_{i=p+1}^{n} \delta_i \right) \setminus \delta_p \right| - B = g_2(\omega)$$

$$B = \min \left(C - |\delta_p|, \left| \left(\bigcup_{i=1}^{p-1} \delta_i \cap \bigcup_{i=p+1}^{n} \delta_i \right) \setminus \delta_p \right| \right)$$

□

4.5.3 Dominance relation

A dominance relation, described in Proposition 4.6, is proposed. It is applied before the exploration starts, so as to divide the search space into several independent search spaces.

Proposition 4.6. *Suppose that the set of tasks S can be divided into p distinctive sets of tasks $S_1, \ldots, S_p$ that are not using common inputs, thus verifying relations: $S_i \cap S_j = \emptyset$ and $\bigcup_{s \in S_i} \delta(s) \cap \bigcup_{s' \in S_j} \delta(s') = \emptyset$ for any $i \neq j$. Then any complete solution $\omega = (S, \pi, \gamma)$ is dominated by solution $\omega' = (S, \pi', \gamma')$ that is built from p independent permutations of tasks (S_k, π_k), $k = 1 \ldots p$, that is to say $(S, \pi') = [(S_1, \pi_1), (S_2, \pi_2), \ldots, (S_p, \pi_p)]$.*

Proof. This result being evident, a methodological proof is not given, only an idea of proof is presented. The sets $S_1, \ldots, S_p$ can be ordered independently because they are not containing any common input. □

In order to divide the set of tasks S into p distinctive sets $S_1, \ldots, S_p$, a graph theory algorithm for finding connected components is used [71]. This algorithm is applied on the input dependency graph $G_{ID} = (V, A, t)$, defined as follows. An *input dependency graph* is an undirected graph $G_{ID} = (V, A, t)$, where the set of nodes V represent algorithm tasks (i.e. $V = S$), the set of edges

A represents dependencies on inputs between the tasks and $t : A \to \mathbb{N}^*$ is a weighting function assigning to each graph edge $(s_i, s_j) \in A$ a positive number, $t(s_i, s_j) = |\delta(s_i) \cap \delta(s_j)|$. For any pair of tasks $s_i, s_j \in V$, edge (s_i, s_j) belongs to the graph G_{ID} if and only if $\delta(s_i) \cap \delta(s_j) \neq \emptyset$.

The fact that the sets $S_1, \ldots, S_p$ are not using common inputs allows us to apply the branch-and-bound algorithm separately on each of these sets. In this way, the number of solution search space is reduced from $|S|!$ to $\sum_{i=1}^{p} |S_i|!$. After the application of the branch-and-bound procedure onto each set of task sets $S_1, \ldots, S_p$, p partial solutions $\omega_1, \ldots, \omega_p$ are obtained. Complete solution $\bar{\omega}$ is obtained by joining together these partial solutions. The cost of complete solution $\bar{\omega}$ is $f(\bar{\omega}) = \sum_{i=1}^{p} f(\omega_i)$.

4.5.4 Selection rules

Before describing selection rules we introduce an useful definition. For a search tree node $\omega = (I, \pi)$, let K be a set that contains the tasks from the neighborhood of already ordered tasks I in the input dependency graph:

$$K = \{s_k \in S \setminus I : (s_i, s_k) \in A, s_i \in I\}.$$

The next node to be examined is selected in a greedy fashion, the immediate profit is privileged. The next three rules are applied successively on the set of feasible nodes. The first rule is applied until there are no nodes whose last two tasks are adjacent in the input dependency graph, the second rule is applied until the set K is empty, and afterwards the third one is applied to the remaining nodes. In this way the outputs that are neighbors in the input dependency graph are prioritized.

1. Select the currently active node (I, π) with the largest cost $t\left(s_{\pi(|I|-1)}, s_{\pi(|I|)}\right)$, such that $\left(s_{\pi(|I|-1)}, s_{\pi(|I|)}\right) \in A$, in the input dependency graph $G_{ID} = (V, A, t)$.

2. Select the currently active node (I, π) with the largest edge cost $t\left(s, s_{\pi(|I|)}\right)$, such that $s_{\pi(|I|)} \in K$ and $s \in I$, in the input dependency graph $G_{ID} = (V, A, t)$.

3. Select the currently active node $\omega = (I, \pi)$ with the least lower-bound cost $g(\omega)$.

4.6 Heuristics

In this section we propose two heuristics. Our motivation to introduce a heuristic method is, firstly, the $\mathcal{NP}$-hardness aspect of the task ordering and memory management problem, and secondly, the necessity in practical situations to find solutions to the problem in a reasonable time.

4.6.1 Two-stage TSP based heuristic (2STSP)

4.6.1.1 One-step history limited data reuse

Let us consider the task ordering and memory management problem for an application (S, E, δ). We suppose that tasks are calculated one by one, and we can reuse only the inputs loaded for previously calculated task, thus the data reuse is limited to one-step history. In this case the objective function to minimize becomes:

$$f'(S, \pi, \gamma) = \left|\delta\left(s_{\pi(1)}\right)\right| + \sum_{i=2}^{|S|} \left|\delta\left(s_{\pi(i)}\right) \setminus \delta\left(s_{\pi(i-1)}\right)\right|$$

This special case of the task ordering and memory management problem resembles to the one machine scheduling problem with SDST.

For a given application (S, E, δ) we build a digraph $G = (V, A, c)$, defined as follows. The set of vertices $V = S \cup \{s_d\}$ contains application tasks plus a dummy task s_d, which does not use any input $\delta(s_d) = \emptyset$. The set of edges A define a complete graph. The cost function $c : A \to \mathbb{N}$ associates to each edge (s, s') the number of external memory accesses needed to execute task s' after task s, thus $c(s, s') = |\delta(s') \setminus \delta(s)|$.

Then, it can be easily shown that the solution of the asymmetric traveling salesman (TSP) problem applied on graph G gives the optimal solution to the task ordering and memory management problem with one-step limited data reuse. Let H be the lowest cost Hamiltonian circuit in graph G, then relation $\sum_{(i,j)\in H} c(i,j) \leq \sum_{(i,j)\in H'} c(i,j)$ is true for any other circuit H'. If the cost function c is replaced with its definition we obtain the objective function of the one-step limited data reuse problem, taking into account that $\delta(s_d) = \emptyset$.

Although, the asymmetric TSP is an NP-complete problem, it allows to solve our problem special case using well studied exact or approximate applications, see [22].

4.6.1.2 A simple two-stage heuristic

In this section we describe a simple two-stage TSP based heuristic for solving the task ordering and memory management problem. The idea behind this heuristic is to divide the problem into two sub-problems and solve each of them independently. The one-step history limited data reuse special case described in the previous section together with the optimal memory management algorithm are employed. The heuristic is depicted in Algorithm 4.3.

The first sub-problem consists in finding a task ordering that minimizes the total number of external memory accesses when only an one-step history is permitted. It is solved using any approximate method for the asymmetric TSP problem. We have chosen the *randomized arbitrary insertion* (RAI) algorithm

introduced by Brest and Zerovnik [13] because it finds near optimal solutions for asymmetric TSP instances from TSPLIB library [90].

The second sub-problem is to find a memory management for a given task ordering, which is solved in polynomial time using Algorithm 4.2.

Algorithm 4.3 Two-stage TSP based heuristic.

Input: An application (S, E, δ).

1: Build the complete digraph $G = (S \cup \{s_d\}, A, c)$ as defined in Subsection 4.6.1.1.
2: Find the optimal Hamiltonian circuit starting in s_d using the TSP heuristic RAI.
3: Find the task ordering σ which corresponds to the optimal Hamiltonian circuit.
4: Remove the dummy task s_d from the sequence σ.
5: Calculate the number of external memory accesses by applying Algorithm 4.2 on task sequence σ.

4.6.2 Randomized cheapest insertion heuristic (RCI)

The work of Rosenkrantz et al. [92] on heuristic algorithms for TSP inspired us to propose an algorithm similar to the cheapest insertion heuristic. We call our algorithm *randomized cheapest insertion heuristic*. It is a pure greedy heuristic with randomization.

Suppose an application (S, E, δ) is given. The heuristic starts from an empty sequence of tasks. At each step it adds a not yet ordered task in the most favorable place, on position that gives the lowest partial solution cost. The insertions are repeated until a complete task ordering is obtained. The task to add is chosen randomly. This process is repeated a large number of times. The best solution obtained so far is stored as result. We have chosen to repeat it n^2 times. No theoretical arguments influenced our choice. This repetition count gives good results in a short time and also is proportional to the problem size. The algorithm is presented in 4.4.

4.7 Computational results

4.7.1 Employed instances

The evaluation of the proposed exact and approximate methods are done on randomly generated problem instances. Besides the randomly generated instances we have tested the proposed algorithms on an image processing application. In

Algorithm 4.4 Randomized cheapest insertion heuristic.

Input: An application (S, E, δ).
Output: A solution (I_{min}, π_{min}) of cost C_{min}

1: $(I_{min}, \pi_{min}) = (\emptyset, \emptyset \to \emptyset)$
2: $C_{min} = \infty$
3: **for** $iter = 0$ **to** n^2 **do**
4: $(I, \pi) = (\emptyset, \emptyset \to \emptyset)$
5: **while** $I \neq S$ **do**
6: Choose randomly a task $t \in S \setminus I$
7: Insert t in permutation π on the cheapest position
8: $I = I \cup \{t\}$
9: **end while**
10: **if** $f(I, \pi) < C_{min}$ **then**
11: $(I_{min}, \pi_{min}) = (I, \pi)$
12: $C_{min} = f(I, \pi)$
13: **end if**
14: **end for**

what follows we describe how the random instances are generated and in sequel how the image processing application is acquired.

A randomly generated problem instance is characterized by five parameters: the number of tasks $n \in \{10, 20, 30\}$, the number of algorithm inputs $m \in \{1/2, 1, 3/2, 2\} \cdot n$, the average number of inputs per task (IpT) $\mu \in \{1/2, 1/4, 1/8\} \cdot m$, the standard deviation $\sigma \in \{1/2, 1/4, 1/8\} \cdot \mu$ of IpT and the distribution used for generating IpT. Three random distributions are used for IpT value generation: $\mathcal{U}(\mu - \sigma, \mu + \sigma)$, $\mathcal{N}(\mu, \sigma^2)$ and $Exponential(\mu^{-1})$. An instance is generated as follows:

- a number x of inputs per task is randomly generated using one of the distribution functions with parameters μ and σ, and x is rounded to the nearest integer (eventually x is floored or ceiled in order to verify $1 \leq x \leq m$),
- for each task s the set of inputs $\delta(s)$ is uniformly drawn from the set of all inputs $e_1, \ldots, e_m$ such that the length of $\delta(s)$ is x.

For each random problem instance we define the minimal size of internal memory size as $C_{min} = \min_i |\delta(s_i)|$. In order to be feasible problem's internal memory size C must be bigger than C_{min}. In our experiments we use six values for C, $C = r \cdot C_{min}$ where $r \in \{1.0, 1.1, 1.2, 1.3, 1.4, 1.5\}$. For each combination of parameters three instances are generated, in total 5832 problem instances are obtained.

The hypothesis about intrinsically parallel applications constrains us to limit computational experiments to easily parallelizible applications. A good example of these are image processing algorithms. A highly parallel execution is possible for many of them. Image processing algorithms are working with huge amounts of data, e.g. one of the smallest image resolution being 640×480 pixels. Because of this fact, and of course, the $\mathcal{NP}$-hardness of the task ordering and memory management problem, exact resolution of practical instances is out of reach of even the most sophisticated methods. In order to be able to compare the proposed heuristics with the exact branch-and-bound method, we limit our computational results to small instances of image processing algorithms. Nevertheless, we can use the solutions of small instances so as to probe the structures of optimal solutions for real world image processing applications. Thereafter, we do so for the classical image processing primitive: image convolution (see [45] for more details).

Image processing application input and output parameters are images. An application task uses several pixels from the input image to calculate a pixel for the output one. We search to order the execution of tasks so as to minimize the number of external memory accesses. Then, guided by the obtained results we try to find patterns in task execution order and to generalize them to higher resolution images.

Image convolution algorithm calculates the convolution product of an image I with a kernel K, $\sum_i \sum_j I\left[p-i, q-j\right] \cdot K\left[i, j\right]$. It computes the value of an output image pixel in function of its neighborhood pixels in the input image, for our experiments we use a 3×3 square neighborhood. We suppose that a task calculates the convolution product for one output pixel, so each task uses 9 inputs. Different image sizes are used.

4.7.2 Branch-and-bound evaluation

In this subsection we investigate the performance of the proposed branch-and-bound method. Initially the randomly generated instances are tested followed by the image convolution ones.

4.7.2.1 Random instances

The branch-and-bound algorithm is executed on each randomly generated problem instance with a time limit of 20 minutes. We are considering as optimal solutions the solutions for which the branch-and-bound method explored the entire search tree. In reality the number of optimal solutions could be larger because in tree search methods much time is spent for optimality proof.

Table 4.1 presents the number of found optimal solutions for each parameter. From the total of 5832 problem instances 3854 were solved to optimality, which corresponds to approximatively 67%. When the internal memory is larger the number of optimal solutions increases as well, we can see that augmenting C by

1.0	1.1	1.2	1.3	1.4	1.5
56%	60%	64%	68%	72%	76%

(a) Internal memory size ratio r.

10	20	30
100%	63%	35%

(b) Number of tasks n.

$1/2n$	n	$3/2n$	$2n$
75%	66%	64%	59%

(c) Number of inputs m.

$1/2m$	$1/4m$	$1/8m$
70%	59%	70%

(d) Average number of IpT μ.

$1/2\mu$	$1/4\mu$	$1/8\mu$
71%	65%	63%

(e) Standard deviation of IpT σ.

Uniform	Normal	Exponential
50%	61%	87%

(f) IpT distribution function.

Table 4.1: Optimal number of solutions (in percents) for each parameter.

50% ($r = 1.5$) the number of optimal solutions increases from 56% to 76%. All the instances built of 10 tasks are solved to optimality, this percentage decreases for instances of 20 and 30 tasks to 63% and respectively 35%. Instances with larger input sets (number of inputs m) become more difficult, which is counterintuitive as one can think that with the increase of number of inputs the problem becomes more decoupled because the number of common inputs between tasks decreases. Another interesting fact is that for exponential number of inputs per task, 87% of instances are solved to optimality. Which is explained by relatively large internal memory sizes when compared to the average number of inputs per task, thus the data reuse is privileged.

4.7.2.2 Image convolution

As image convolution example we take a 7×7 input image instance, in this case the number of tasks will be 25 (output image pixels belonging to image boundaries are not calculated) and the number of inputs 49. We have used nine different internal memory sizes, $C \in 9, \ldots, 17$. The obtained solutions (minimal number

Memory size, C		9	10	11	12	13	14	15	16	17
Optimal solution		81	73	65	61	57	56	55	53	49
Execution time, seconds	total	3000	745	70	36	9	46	213	112	<1
	best sol.	209	33	2	2	1	3	<1	12	<1

Table 4.2: Branch-and-bound method results for 7×7 image convolution instance. The internal memory size varies from 9 to 17. Besides the optimal solutions, the time needed to find these solutions ("best sol." row) and the total run time ("total" row) are presented.

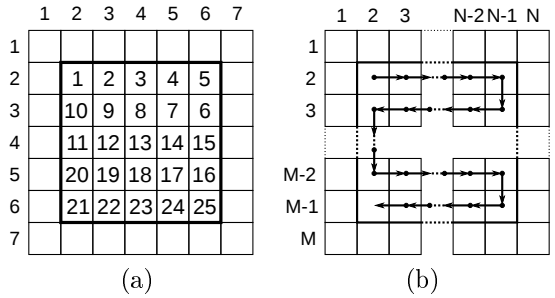


Figure 4.2: Image convolution task processing order for $C = 9$.

of external memory accesses) and execution times are depicted in Table 4.2. The minimal internal memory size is $C = 9$ (the maximum number of inputs per task). This is the most constrained instance and the search time is less than an hour. As often observed with branch-and-bound algorithms, the minimal solution is rapidly found and most of the calculation time is spent on optimality proof completion, e.g. for $C = 9$ the best solution is found in approximatively 3 minutes and more than 45 minutes are needed for optimality proof. The convolution instance with the largest memory size, $C = 17$, allows to load each input from the external memory only once. The search time is less than one second for this example.

In Figure 4.2a is illustrated the calculation order of output image pixels found by the branch-and-bound for the image convolution with the minimum possible memory size $C = 9$. The optimal number of external memory accesses for this instance is 81.

It is easy to see that two consecutively calculated output image pixels are either horizontal or vertical neighbors. Thus, we conjecture that if the calculation order of output image pixels satisfies the last rule, then this order is an optimal one. It is straightforward to deduce output image pixels calculation order for higher resolution images based on this rule, e.g. a possible order is given in Figure 4.2b. We note that for each output image pixel, 3 input image pixels must be loaded, except the first output image pixel for which 9 pixels must be loaded. Thus, the number of external memory accesses is $3 \cdot (M-2) \cdot (N-2) + 6$ for a

Image size	6×6	7×7	8×8	6×8	7×8	8×6	8×7
Memory size	15	17	19	15	17	15	17
Calculation order	No matter which			Vertical		Horizontal	

Table 4.3: internal memory sizes allowing to load input image pixels only once.

$N \times M$ input image.

Current processor technology admits bigger internal memory sizes than we have used for the above image convolution instance. Therefore, we carry out an experiment that aims to find the minimal internal memory size allowing to load only once each one of input image pixels. Several convolution instances with different image dimensions, varying from 5×5 to 8×8 pixels, were tested. The results are presented in Table 4.3. We can see that if the internal memory size equals to $2 \cdot \min(N, M) + 3$ for a $N \times M$ input image then the number of external memory accesses equals to $N \cdot M$. E.g. for a 640×480 image, if the internal memory size can store 963 $(2 \cdot 480 + 3)$ pixels then each input image pixel will be accessed exactly once, i.e. 307200 external memory accesses; contrary to a 9 pixel internal memory for which approximatively 3 times more external memory accesses are needed (914898). We shall note that two different calculation orders of output image pixels were found by the branch-and-bound algorithm. The first one is horizontal, here output image pixels are calculated line by line[4] and the second one is vertical order, output pixels being calculated column by column.

4.7.3 Heuristics evaluation

4.7.3.1 Random instances

The two-step TSP based and randomized cheapest insertion heuristics were executed on the 3854 random instances that were solved to optimality by the branch-and-bound method. In this way, we are able to compare the solutions found by the heuristics to the minimum possible ones.

The number of instances for which optimal solutions are obtained for the 2STSP heuristic is 2543 (66%) and for the RCI heuristic is 3185 (83%). Thus the RCI heuristic finds more optimal solutions than the 2STSP. The percentage of optimal solutions found by the heuristic algorithms for each random instance parameter are depicted in 4.4. Here too, the RCI heuristic gives better results.

The *average deviation from the optimum* is the average of deviations of heuristic solutions from the optimal values. Namely, the average of ratios $(S-S^*)/S^*$ where S represent heuristic solution and S^* the optimal solution value. The 2STSP heuristic is situated at 5.14% from the optimum in average, whilst the RCI is situated at less than 2% (1.85%) from the optimum in average. In terms

[4]In figure 4.2b a horizontal order is illustrated

Heuristic\r	1.0	1.1	1.2	1.3	1.4	1.5
2STSP	65%	62%	61%	63%	68%	75%
RCI	95%	81%	76%	78%	81%	87%

(a) Internal memory size ratio r.

Heuristic\n	10	20	30
2STSP	69%	51%	86%
RCI	88%	67%	94%

(b) Number of tasks n.

Heuristic\m	$1/2n$	n	$3/2n$	$2n$
2STSP	80%	62%	59%	60%
RCI	90%	83%	77%	80%

(c) Number of inputs m.

Heuristic\μ	$1/2m$	$1/4m$	$1/8m$
2STSP	76%	60%	60%
RCI	89%	82%	77%

(d) Average number of IpT μ.

Heuristic\σ	$1/2\mu$	$1/4\mu$	$1/8\mu$
2STSP	73%	66%	58%
RCI	87%	83%	78%

(e) Standard deviation of IpT σ.

Heuristic\Dist.	Uniform	Normal	Exponential
2STSP	45%	58%	84%
RCI	68%	76%	95%

(f) IpT distribution function.

Table 4.4: 2STSP and RCI heuristics number of optimal solutions (in percents) for each parameter.

Memory size, C	9	10	11	12	13	14	15	16	17	18	19	20	21
B&B	81	73	65	61	57	56	55	53	49	49	49	49	49
2STSP	81	77	73	69	65	61	57	53	49	49	49	49	49
RCI	81	73	65	61	58	57	55	54	53	52	52	50	49

Table 4.5: Comparison between the exact solution (B&B) and the approximate solutions: two-stage TSP based heuristic (2STSP) and randomized cheapest insertion heuristic (RCI).

of deviation from the optimum the RCI is also better than 2STSP.

In order to discriminate between the optimal solutions found by each heuristic apart we perform a cross comparison. The number of solutions for which the 2STSP heuristic is strictly better, equal and strictly worse than the RCI are counted. Solely in one case the 2STSP found a better solution than RCI, for 2664 (69%) instances the solutions values are the same, and, the RCI heuristic found better solutions for 1189 (31%) instances. We conjecture that the employment of the RCI heuristic is more appealing as it outperforms the 2STSP heuristic.

4.7.3.2 Image convolution

In Table 4.5 are presented the optimal and the approximate solutions found by the 2STSP and RCI heuristics. The same image convolution instance as in branch-and-bound evaluation is used. We recall that it was a 7×7 image convolution with 25 tasks and 49 inputs. The internal memory size ranges from 9 to 21. As we can observe for the minimum memory size, $C = 9$, the three solutions coincide. This can be explained by the fact that for the image convolution with a minimal internal memory size, the data reuse is limited to one-step history. As the internal memory size grows worse solutions are found by the 2STSP heuristic. This trend lasts until $C = 17$, from which on the heuristic provides optimal results. The RCI heuristic has an opposite behavior. For small values of internal memory size the heuristic gives optimal results, whereas for large values near optimal solutions. The minimum possible number of memory accesses is obtained by the branch-and-bound and 2STSP for $C = 17$ and by the RCI heuristic heuristic for $C = 21$.

4.7.3.3 Two-stage heuristic evaluation

Some others image processing algorithm instances (e.g. the Hough transformation) have been used to test the proposed heuristic, giving the same kind of insights.

4.8 Conclusions

In this chapter, the task ordering and memory management problem was examined. The aim of the last is to find a task execution order and an external memory data loading strategy so as to minimize the total number of external memory accesses for a dataflow application. The main constraint is that the cluster memory is limited in size, thus an optimal data management strategy is necessary. Besides its direct utilization for optimizing the memory accesses of an application, this problem can be also used to find the achievable degree of parallelism for a parallel application, which in a compiler chain proceeding by parallelism reduction will help to fix an appropriate target.

Initially we have supposed that we are given a task ordering, so as to study the issue of internal memory data management separately from the ordering problem. We have proposed a polynomial algorithm for its resolution. This algorithm relies on Belady's principle for virtual memory management.

We have proposed a branch-and-bound algorithm for the task ordering and memory management problem and described its building blocks (lower bound, dominance relation etc.). Because exact methods cannot be used for optimizing real applications we have proposed two heuristic methods. The first one (2SSTS) is a two-step heuristic based on Hamiltonian circuit search and the second one is a randomized cheapest insertion (RCI) algorithm.

Firstly we have performed computational experiments with randomly generated problem instances. The branch-and-bound procedure found optimal solution for more than two thirds (67%) of the cases. The heuristic methods were tested on these instances. The RCI heuristic produced optimal solutions in 83% of the instances, while the 2STSP only in 66%. Afterwards, several tests were done with an image processing application: the image convolution. The branch-and-bound algorithm was not able to solve instances of image processing applications applied on real, high resolution images, because the size of the search space is unimaginably huge for an exact algorithm. However we solved instances with low resolution input images and generalized the results for high resolution images. The heuristic methods performed quite well on these instances. With different internal memory size configurations they have found near optimal solutions.

Conclusions and future work

In this thesis, we have examined memory access management and scheduling issues for dataflow applications executed on embedded multi-core processors. These were instantiated into three combinatorial optimization problems. Two of them deal with optimal prefetch strategies and the third one studies data reutilization techniques. In the first problem the prefetch optimization aspect for a dataflow application was studied. We have modeled it as a hybrid flow shop problem under precedence constraints with makespan minimization as objective. The second problem considered prefetching for branching structures (tasks which provide data-controlled execution). The objective was to find an optimal data prefetching strategy which minimizes different execution time statistics. The last problem, the data reutilization one, consisted in sequencing data loading operations having as objective the data reuse maximization.

For each of the above problems we have proposed resolution methods, mainly heuristic algorithms, and auxiliary utilities (global lower bounds, exact resolution algorithms) needed for their performance assessment. The resolution methods quality has been experimentally established on synthetic instances and, in some cases, on practical applications.

Several possibilities exist for estimating the quality of an approximate resolution algorithm. At one extreme, one may rely only on experimental data. Alternatively, one may develop exact resolution algorithms so as to estimate the quality of the results obtained on a benchmark of small instances. It is further possible to develop computation-intensive global lower bounds (e.g. via the polyhedral approach or SDP) to push this experimental validation in the realm of larger instances. Even better, when one can obtain computationally cheap though tight enough global lower bounds, these bounds can become part of the algorithm itself providing a precise estimation of the optimality gap for each instance ever to be solved. To the exception of the case of heuristics for which an universal (and acceptable) optimality gap bound is known[5], the latter is in our opinion the best case. For the most representative work of this thesis, the flow shop problem, we have been able to fulfill the requirements of the latter approach, hence proving acceptable optimality gaps for any solved problem instance. In contrast, for

[5]However, such results are very hard to obtain and occur mainly for simple algorithms dedicated to "pure" mathematical problems.

the data reutilization problem we have analyzed the quality of the solutions on a benchmark of small instances, using an exact resolution method, because there is a little hope to obtain cheap tight global lower bounds.

A number of perspective research directions, in our opinion, represent a natural continuation of these thesis works. In particular it should be interesting to consider prefetch scheduling problems with probabilistic durations and conditional execution as an extension of the studied hybrid flow shop. A first attempt [24] was to examine probabilistic parameters (e.g. job heads and tails) of dataflow applications with conditional execution tasks. Another research direction will be to optimize the speculative prefetch management for recursive (hierarchies of) branching structures and also to consider dispersion-like objectives[6].

In terms of more practical applications of this work, it should be emphasized that the aforementioned flow shop problem closely modelizes the execution model for a real-world massively (>200 cores) multi-core embedded processor. As such, our resolution method can be considered for integration in a future release of the industry-grade dataflow compilation chain presently developed within a joint lab with a semiconductor industry partner, in particular, as already stated in this manuscript, when large memory footprint applications have to be executed on such highly-constrained platforms. On top of this, the described methods can be applied on highly-dynamic, control-oriented applications such as H.264 video encoding algorithm with early motion estimation termination [111, 96], classifier-based object tracking applications [61], cognitive radio [83], as well as next generation video coding standards such as HEVC (H.265).

As already stated, the increasing complexity of both microprocessors and their associated compilation chains reinforce the need for advanced OR techniques. By tackling a number of problems from that field, we hope to have contributed to both the advancement and the perception of combinatorial optimization as one of the key scientific discipline to make the new generation of multi-core processors a reality!

[6]In the embedded computing field repeatability guarantees (obtained by dispersion minimization) sometimes have a larger importance than other execution time statistics.

Bibliography

[1] Standard task graph set `http://www.kasahara.elec.waseda.ac.jp/schedule/index.html`. Last access July 11, 2011.

[2] T. Alexander and G. Kedem. Distributed prefetch-buffer/cache design for high performance memory systems. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, HPCA '96, pages 254–, 1996.

[3] A. Allahverdi, C. Ng, T. Cheng, and M. Kovalyov. A survey of scheduling problems with setup times or costs. *European Journal of Operational Research*, 187(3):985–1032, June 2008.

[4] W. Anacker and C. P. Wang. Performance evaluation of computing systems with memory hierarchies. *Electronic Computers, IEEE Transactions on*, EC-16(6):764 –773, 1967.

[5] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 176–186, 1991.

[6] P. Baptiste, C. Le Pape, and W. Nuijten. Satisfiability tests and time-bound adjustments for cumulative scheduling problems. *Annals of Operations Research*, 92:305–333, 1999.

[7] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[8] D. Bernstein, D. Cohen, and A. Freund. Compiler techniques for data prefetching on the powerpc. In *Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, PACT '95, pages 19–26, 1995.

[9] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397–408, 1996.

[10] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys*, 38, June 2006.

[11] V. Botta-Genoulaz. Considering bills of material in hybrid flow shop scheduling problems. In *IEEE International Symposium on Assembly and Task Planning 1997, ISATP 97.*, pages 194–199, 1997.

[12] V. Botta-Genoulaz. Hybrid flow shop scheduling with precedence constraints and time lags to minimize maximum lateness. *International Journal of Production Economics*, 64(1-3):101 – 111, 2000.

[13] J. Brest and J. Zerovnik. A heuristic for the asymmetric traveling salesman problem. In *The 6th Metaheuristics International Conference*, 2005.

[14] J. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *Signals, Systems and Computers, 1994. Conference Record of the Twenty-Eighth Asilomar Conference on*, volume 1, pages 508 –513, 1994.

[15] J. Buck and E. Lee. *Advanced Topics in Dataflow Computing and Multithreading*, chapter The Token Flow Model. IEEE Computer Society Press, 1995.

[16] S. Byna, Y. Chen, and X.-H. Sun. A taxonomy of data prefetching mechanisms. In *Proceedings of the The International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 19–24. IEEE Computer Society, 2008.

[17] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in java. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 280–291, 2001.

[18] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IV, pages 40–52. ACM, 1991.

[19] J. Carlier. The one-machine sequencing problem. *European Journal of Operational Research*, 11(1):42–47, September 1982.

[20] J. Carlier and P. Chrétienne. *Problèmes d'ordonnancement - Modélisation, Complexité, Algorithmes.* Masson, Paris, 1988.

[21] J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78(2):146–161, 1994.

[22] G. Carpaneto, M. Dell'Amico, and P. Toth. Exact solution of large-scale, asymmetric traveling salesman problems. *ACM Transactions on Mathematical Software*, 21(4):394–409, 1995.

[23] S. Carpov, R. Sirdey, J. Carlier, and D. Nace. Speculative data prefetching for branching structures in dataflow programs. *Electronic Notes in Discrete Mathematics*, 36:119–126, 2010.

[24] S. Carpov, J. Carlier, D. Nace, and R. Sirdey. Probabilistic parameters of conditional task graphs. In *Proceedings of the 5th International Workshop on Advanced Distributed and Parallel Network Applications*, 2011.

[25] S. Carpov, J. Carlier, D. Nace, and R. Sirdey. Task ordering and memory management problem for degree of parallelism estimation. In *Lecture Notes in Computer Science*, volume 6842, pages 592–603, 2011.

[26] S. Carpov, J. Carlier, D. Nace, and R. Sirdey. Two-stage hybrid flow shop with precedence constraints and parallel machines at second stage. *Computers & Operations Research*, 39(3):736 – 745, 2012.

[27] T.-F. Chen. An effective programmable prefetch engine for on-chip caches. In *Proceedings of the 28th annual international symposium on Microarchitecture*, MICRO 28, pages 237–242. IEEE Computer Society Press, 1995.

[28] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44:609–623, 1995.

[29] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W.-m. W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th annual international symposium on Microarchitecture*, MICRO 24, pages 69–73, 1991.

[30] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing - Volume 01*, ICPP '93, pages 56–63, 1993.

[31] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. *SIGPLAN Notices*, 34(5):229–241, 1999.

[32] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1):108–134, 2004.

[33] C. Ding and M. Orlovich. The potential of computation regrouping for improving locality. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 13, 2004.

[34] M. Dror and P. Mullaseril. Three stage generalized flowshop: Scheduling civil engineering projects. *Journal of Global Optimization*, 9:321–344(24), 1996.

[35] E. Fernandez and B. Bussell. Bounds on the number of processors and time for multiprocessor optimal schedules. *IEEE Transactions on Computers*, 22(8):745–751, 1973.

[36] J. Fidge. Timestamps in message passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference (ACSC'88)*, 1988.

[37] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21:948–960, 1972.

[38] J. W. C. Fu and J. H. Patel. Data prefetching in multiprocessor vector cache memories. In *Proceedings of the 18th annual international symposium on Computer architecture*, ISCA '91, pages 54–63. ACM, 1991.

[39] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, pages 102–110, 1992.

[40] G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC'02)*, pages 305–315, 2002.

[41] I. Ganusov and M. Burtscher. Future execution: A hardware prefetching technique for chip multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 350–360. IEEE Computer Society, 2005.

[42] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[43] A. Girault, B. Lee, and E. Lee. Hierarchical finite state machines with multiple concurrency models. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(6):742 –760, 1999.

[44] A. Gladky, Y. Shafransky, and V. Strusevich. Flow shop scheduling problems under machine-dependent precedence constraints. *Journal of Combinatorial Optimization*, 8(1):13–28, 2004.

[45] R. Gonzalez and R. Woods. *Digital Image Processing*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[46] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 151–162, 2006.

[47] E. H. Gornish and A. Veidenbaum. An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 02*, ICPP '94, pages 281–284. IEEE Computer Society, 1994.

[48] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of the 4th international conference on Supercomputing*, ICS '90, pages 354–368. ACM, 1990.

[49] T. Goubier, R. Sirdey, and V. David. Le Langage de Programmation ΣC v2. Technical Report LIST/DACLE/2010-0050/TG, CEA LIST, Saclay, 2010.

[50] T. Goubier, R. Sirdey, S. Louise, and V. David. ΣC: A programming Model and Language for Embedded Manycores. Submitted 2011.

[51] A. Guinet and M. Legrand. Reduction of job-shop problems to flow-shop problems with precedence constraints. *European Journal of Operational Research*, 109(1):96 – 110, 1998.

[52] J. Gupta. Two-stage, hybrid flowshop scheduling problem. *Journal of the Operational Research Society*, 39:359–364, 1988.

[53] J. Gupta, A. Hariri, and C. Potts. Scheduling a two-stage hybrid flow shop with parallel machines at the first stage. *Annals of Operations Research*, 69:171–191, 1997.

[54] W.-S. Han, Y.-S. Moon, and K.-Y. Whang. Prefetchguide: capturing navigational access patterns for prefetching in client/server object-oriented/object-relational dbmss. *Information Sciences*, 152:47–61, June 2003.

[55] C. Hanen and A. Munier. A study of the cyclic scheduling problem on parallel processors. *Discrete Appl. Math.*, 57:167–192, 1995.

[56] J. P. Hart and A. W. Shogan. Semi-greedy heuristics: An empirical study. *Operations Research Letters*, 6(3):107 – 114, 1987.

[57] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1992.

[58] J. A. Hoogeveen, J. K. Lenstra, and B. Veltman. Preemptive scheduling in a two-stage multiprocessor flow shop is np-hard. *European Journal of Operational Research*, 89(1):172 – 175, 1996.

[59] J.-J. Hwang, Y.-C. Chow, F. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, 1989.

[60] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *Computers and Digital Techniques, IEE Proceedings*, 152(2):114 – 129, 2005.

[61] O. Javed and M. Shah. Tracking and object classification for automated surveillance. In *Proceedings of the 7th European Conference on Computer Vision-Part IV*, ECCV '02, pages 343–357. Springer-Verlag, 2002.

[62] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 252–263. ACM, 1997.

[63] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 364–373, 1990.

[64] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, 1974.

[65] Kalray. Introduction to the MPPA technology. A technical overview. Technical Report KETD-58, Kalray S. A., 2011.

[66] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, 33(11): 1023–1029, 1984.

[67] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin, Germany, 2004.

[68] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *Electronic Computers, IRE Transactions on*, EC-11(2):223 –235, 1962.

[69] W. H. Kohler and K. Steiglitz. Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems. *Journal of the ACM*, 21:140–156, 1974.

[70] S. Kohli. Cache aware scheduling for synchronous dataflow programs. Technical Report UCB/ERL M04/3, EECS Department, University of California, Berkeley, 2004.

[71] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer-Verlag, 2002.

[72] T. Lang and E. Fernandez. Improving the computation of lower bounds for optimal schedules. *IBM Journal of Research and Development*, 21(3): 273–280, 1977.

[73] B. Lee and A. Hurson. Issues in dataflow computing. *Advances in Computers*, 37:285 – 333, 1993.

[74] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[75] E. A. Lee and T. M. Parks. Dataflow process networks. In *Proceedings of the IEEE*, pages 773–799, 1995.

[76] M. Lipasti, W. Schmidt, S. Kunkel, and R. Roediger. Spaid: software prefetching in pointer and call intensive environments. In *Proceedings of the 28th annual international symposium on Microarchitecture*, MICRO 28, pages 231–236, 1995.

[77] M. Lombardi and M. Milano. Allocation and scheduling of conditional task graphs. *Artificial Intelligence*, 174:500–529, 2010.

[78] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th annual international symposium on Computer architecture*, ISCA '01, pages 40–51. ACM, 2001.

[79] C.-K. Luk and T. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, pages 222–233, 1996.

[80] P. Machanick, P. Salverda, and L. Pompe. Hardware-software trade-offs in a direct rambus implementation of the rampage memory hierarchy. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VIII, pages 105–114. ACM, 1998.

[81] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Systems*, 1988.

[82] K. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, 1996.

[83] I. Mitola, J. and J. Maguire, G.Q. Cognitive radio: making software radios more personal. *Personal Communications, IEEE*, 6(4):13–18, 1999.

[84] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12:87–106, 1991.

[85] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, ASPLOS-V, pages 62–73, 1992.

[86] W. A. Najjar, E. A. Lee, and G. R. Gao. Advances in the dataflow computational model. *Parallel Computing*, 25:1907–1929, 1999.

[87] N. Petersen and M. Wojcik. Node prefetch prediction in dataflow graphs. In *Signal Processing Systems, 2004. SIPS 2004. IEEE Workshop on*, pages 310 – 315, 2004.

[88] V. Pingali, S. McKee, W. C. Hsieh, and J. Carter. Restructuring computations for temporal data cache locality. *International Journal of Parallel Programming*, 31:305–338, 2003.

[89] S. Przybylski. The performance impact of block sizes and fetch strategies. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 160–169, 1990.

[90] G. Reinelt. Tsplib - a traveling salesman problem library. *INFORMS Journal on Computing*, 3(4):376–384, 1991.

[91] I. Ribas, R. Leisten, and J. Framinan. Review and classification of hybrid flow shop scheduling problems from a production system and a solutions procedure perspective. *Computers & Operations Research*, 37(8):1439 – 1454, 2010.

[92] D. Rosenkrantz, R. Stearns, and P. L. II. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3): 563–581, 1977.

[93] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VIII, pages 115–126. ACM, 1998.

[94] R. Ruiz, F. Şerifoğlu, and T. Urlings. Modeling realistic hybrid flexible flowshop scheduling problems. *Computers & Operations Research*, 35(4): 1151–1175, 2008.

[95] V. Santhanam, E. Gornish, and W.-C. Hsu. Data prefetching on the hp pa-8000. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 264–273, 1997.

[96] M. Sarwer and Q. Wu. Adaptive variable block-size early motion estimation termination algorithm for h.264/avc video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 19(8):1196 –1201, 2009.

[97] O. Sinnen. *Task Scheduling for Parallel Systems*. Wiley-Interscience, 2007.

[98] R. Sirdey and P. Aubry. A linear programming approach to general dataflow process network verification and dimensioning. In *Proceedings ICE 2010*, pages 115–119, 2010.

[99] R. Sirdey and V. David. Procédé d'exécution cadencé par un temps logique vectoriel permettant l'exécution haute performance d'applications multi-tâches du type flot de données avec contrôle sur des architectures massivement multi-coeurs. Technical Report LIST/DACLE/2010-123/RS, CEA LIST, Saclay, 2010.

[100] R. Sirdey and V. David. Système d'ordonnancement de l'exécution de taches cadencé par un temps logique vectoriel (procédé d'exécution). Brevet, n° de dépot 1003963, 2010.

[101] A. Smith. Cache memories. *ACM Computing Surveys*, 14:473–530, 1982.

[102] M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 91–102. ACM, 2003.

[103] V. Strusevich. Shop scheduling problems under precedence constraints. *Annals of Operations Research*, 69:351–377, 1997.

[104] M. Subramanian and V. Krishnamurthy. Performance challenges in object-relational dbmss. *IEEE Data Engineering Bulletin*, 22(2):27–31, 1999.

[105] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10:384–393, June 1975.

[106] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32:174–199, 2000.

[107] Z. Wang, T. O'Neil, and E.-M. Sha. Optimal loop scheduling for hiding memory latency based on two-levelpartitioning and prefetching. *Signal Processing, IEEE Transactions on*, 49(11):2853 –2864, 2001.

[108] M. V. Wilkes. Slave memories and dynamic storage allocation. *Electronic Computers, IEEE Transactions on*, EC-14(2):270 –271, 1965.

[109] M. Wolf and M. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, 1991.

[110] Y. Xie and W. Wolf. Allocation and scheduling of conditional task graph in hardware/software co-synthesis. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '01, pages 620–625, 2001.

[111] L. Yang, K. Yu, J. Li, and S. Li. An effective variable block-size early termination algorithm for h.264 video coding. *Circuits and Systems for Video Technology, IEEE Transactions on*, 15(6):784 – 788, 2005.

[112] Z. Zhang and J. Torrellas. Speeding up irregular applications in shared-memory multiprocessors: memory binding and group prefetching. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 188–199. ACM, 1995.