Scheduling for Horizontal Systems: The VLIW Paradigm in Perspective

Franco Gasperoni
Courant Institute of Mathematical Sciences
New York University

May 1991

A dissertation in the Department of Computer Science Submitted to the Faculty of the Graduate School of Arts and Sciences in partial fulfillment of the requirements for the degree of Doctor of Philosophy at New York University.

Approved: ______________________________
Edmond Schonberg, Research Advisor

Scheduling for Horizontal Systems: The VLIW Paradigm in Perspective

Franco Gasperoni
Thesis advisor: Edmond Schonberg
1991

Abstract

This work focuses on automatic extraction of operation level parallelism from programs originally intended to be sequential. Optimality issues in the framework of very long instruction word architectures and compilers (VLIW) are investigated. Possible advantages of an idealized dynamic scheduler over a purely static one are also explored. More specifically the model and the results of scheduling theory are extended to account for cyclicity and branching capabilities present in sequential programs. The existence of inherent bottlenecks in the VLIW paradigm is substantiated and the advantage of dynamic over static scheduling is demonstrated for certain type of loops. A novel technique for efficient parallelization of straight line loops is presented. A simple scheduling heuristic for arbitrary programs is proven to perform between a constant and a logarithmic factor from appropriately defined optimality criteria. Finally we prove the existence of loops containing branches for which no parallel program can achieve time optimal performance on VLIWs with unlimited resources. The overall aim of the thesis is to identify the family of sequential programs for which the VLIW model of parallel computation is viable.

à Pierre et Nathalie

Acknowledgements

This dissertation originated at NYU and developed during my stay at IBM T.J. Watson research center. I would like to thank both institutions for their financial support.

There are many persons that helped me through the long and sometimes spooky process of writing a dissertation. First of all I would like to thank my advisors Ed Schonberg of NYU and Uwe Schwiegelshohn of IBM for their help and suggestions and for constantly being available to monitor my progress. Along with Ralph Grishman of NYU, whom I also thank, they had the uneasy task of digesting my theoretical claims. I would also like to thank Uwe for his friendship and assistance during my stay at IBM making my experience there a very enriching one. In more than one occasion Uwe helped me finalyze some of the ideas appearing in this thesis.

I would like to thank all my friends at NYU in particular John Turek with whom I spent many long evenings trying to solve the open problem mentioned before theorem 4.6.8. Part of the results in section 4.6 have emerged from the excitement of those evenings. I am grateful to Kemal Ebcioğlu of IBM for sharing with me his experience and ideas on VLIWs and to Brion Shimamoto from IBM who helped me understand the importance of managerial skills in an industrial research environment. I would also like to thank Bulent Abalı, Hu Chao and Toshio Nakatani all from IBM for helpful discussions.

Finally I am grateful to my parents and my uncle Gian-Carlo for so many things that I cannot list them here and to my wife Nathalie for her constant love and affection without which I would have never gone through the dissertation adventure.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Vast amounts of serial applications have been written to date. To exploit increases in computing performance offered by parallel processing such applications have to be rewritten or recompiled with parallelizing compilers.

The first approach provides the biggest opportunity for speedup since one can design a parallel program suited to the architecture at hand. This is not a painless and inexpensive activity and often involves a thorough knowledge of the target machine idiosyncrasies. Users resort to this solution when the running time of a program is critical.

Acknowledging a user's reluctance to recode working applications, a great deal of attention has been devoted to port automatically sequential applications on parallel machines (parallelizing compilers) [37,7,6]. Parallelizing compilers, however, are not as effective as programmers in their program transformation task. A parallelizing compiler has to rely on user assertions and/or source code modifications to improve the quality of the code it generates. When users are unable to restructure source programs, part of the code runs serially. This can potentially imply a poor overall speedup. What makes automatic program parallelization for conventional SIMD/MIMD architectures difficult is the need to extract coarse grained parallelism from existing serial applications.

In 1972 Riseman & Foster inspected traces of sequential programs to determine the amount of operation level parallelism present. They concluded that such parallelism was abundant but discarded it on the basis of exponential hardware exploitation costs [51]. In 1984 Nicolau & Fisher repeated the same experiment. Their final conclusion, however, was that fine grained parallelism could be exploited effectively by a clever combination of hardware and software, as opposed to the purely hardware approach of Riseman & Foster [46].

This combination of hardware and software is termed VLIW paradigm. Since their introduction in the early 80s, VLIWs have gained considerable interest in the academic and industrial communities.

1.2 State of the Art in VLIW

The archetype VLIW architecture comprises several independent functional units, one or several register files and data memory. All machine resources are driven by the same clock and are connected using some sort of topology. The bits which operate each unit are gathered in a *very long instruction word*, whose size can exceed 1,000 bits. A single instruction stream can therefore initiate several independent operations every cycle. A central control unit dictates the sequencing of VLIW instructions based on the conditionals scheduled in the instruction being executed. In first approximation VLIW instructions are similar to horizontal microcode except that they are much longer and more orthogonal.

VLIWs are hard to program by hand as the number of concurrent activities that one needs to keep track of is overwhelming. The scheduler is therefore a key component in every VLIW system. By statically orchestrating resource allocations and data movements, VLIW schedulers transform sequential programs written in high level languages directly into VLIW instructions.

The simplest VLIW scheduling technique one can think of is local scheduling where instructions are restricted to contain operations from the same basic block. In addition to being theoretically unsatisfying, this technique is also poor in practice as the degree of parallelism which can be extracted by local scheduling is often less than a factor two [51].

Global scheduling disregards basic block boundaries when rearranging operations. The first true global technique, trace scheduling, was introduced by Fisher in 1979 [20,22,18]. Trace scheduling employs branching probabilities to select the most likely execution path of a program (the trace). Once a trace has been selected it is compacted as if it was a basic block by using some slightly extended local scheduling algorithm. The scheduling phase may move operations above or below conditional jumps. To preserve semantic equivalence with the original program, trace scheduling has to insert bookkeeping code in the points where the trace interfaces with the rest of the program. After the most probable trace has been compacted the next most probable trace is selected and the process repeats.

As a result of his experience with trace scheduling, Nicolau introduced percolation scheduling, a new framework for the arbitrary motion of operations within a program [45,3]. Percolation scheduling relies on four semantic preserving program transformations in the spirit of [47], that can repeatedly be applied using some high level guidance rules. The repeated application of such elementary transformations allows operations to percolate to the beginning of a program without introducing any superfluous bookkeeping code. Superimposed scheduling algorithms such as compact path [3] and greedy compact [14] assume VLIWs with unbounded resources. The first bounded resource scheduling algorithm relying on percolation scheduling has been devised by Ebcioğlu and Nicolau [17].

Region scheduling is another recent global scheduling technique developed by Gupta and Soffa [29]. Region scheduling works by evenly redistributing the parallelism available throughout a program, so that machine resources are always fully utilized. As with percolation scheduling, parallelism is redistributed through repetitive application of semantic preserving program transformations. The elementary transformations are not directly defined on the flow graph of a program, but rather on the program dependence graph, a new intermediate program representation paradigm [19].

The previous global scheduling techniques are concerned with compacting acyclic code. An inner loop can be parallelized by unrolling its body, to provide sufficient parallelism, and subsequently invoking one of the global scheduling algorithms on the unrolled body. Depending on the amount of unrolling this approach can be space inefficient. Final code size is an important concern in the case of loops because it can interfere with good instruction cache performance. To overcome this potential problem a second class of scheduling algorithms based on the software pipelining concept, has emerged. Loop pipelining is the software version of hardware pipelining: one tries to start a new iteration before the preceding has completed. Loop pipelining was originally introduced by

Kogge to handcode microprogrammed pipelined machines [35], while Cytron and Munshi & Simons have explored its application to multiprocessors [13,42].

Initial attempts in the VLIW community have focused on pipelining inner loops without conditional jumps (straight line inner loops). Several such algorithms have been developed [9,49,58,55]. The most recent results are due to Lam [38], Aiken & Nicolau [4] and Zaky & Sadayappan [61]. Among these Lam's is the only to have purposely been designed for finite resource VLIWs. Lam's algorithm tries to determine some minimal delay λ that needs to be respected between the execution of consecutive iterations. Operations in the loop are rearranged so that resource and data dependence constraints are satisfied when iterations are started every λ cycles. If this is not possible, λ is increased by one cycle and the above process is repeated. Aiken and Nicolau's algorithm unrolls an inner loop a predetermined polynomial amount of times and then greedily compacts it under the assumption that unlimited resources are available. Such greedy scheduling ensures the existence of a repetitive pattern that can be used as the basis for a software pipelined loop. The algorithm finds time optimal loops for VLIWs with unlimited resources. The size of the output loop may however be exponential in the size of the input loop. By formulating the straight line loop pipelining problem as a linear algebraic problem on some path algebra, Zaky and Sadayappan achieve the same optimality result as Aiken and Nicolau but the size of the output loop is guaranteed to be polynomial in the input.

Researchers have also focused their attention on loops containing conditional jumps. Algorithms such as pipeline scheduling and its offspring [14,16] and perfect pipelining [5] are recent algorithms that have explicitly been designed for loops with branches. They all rely on percolation scheduling to carry out the motion of operations. None of these algorithms has received wide pragmatic acceptance nor have theoretical performance bounds been established.

1.3 Problems of Current VLIW Techniques

Despite its interesting results for numerical programs, trace scheduling has some downfalls. For one thing if branching probabilities are insufficiently skewed performance can be meager; furthermore Ellis showed that exponential increase in code size is possible

[18]. Finally because only one trace at a time is considered some bookkeeping code is superfluous and motion of certain operations can be limited [2].

This last problem is solved in percolation scheduling. Unfortunately algorithms relying on percolation scheduling tend to be slow. This is due to the granularity of its transformations which force, during code generation, continuous traversals of the source program control flow graph. Superimposed scheduling algorithms for unlimited resource VLIWs meet final resource constraints by partitioning and recombining instructions initially created. Part of the scheduling work is thus undone to accommodate resource limitations. Ignoring machine capabilities during scheduling not only increases compilation time but may actually yield a VLIW program slower than its sequential counterpart. Finally extensive code rearrangement is needed, thus requiring unacceptable amounts of compensation code.

To decrease compilation time, Ebcioğlu and Nicolau have explicitly worked with bounded resources [17]. The draw-back of their method is the excessive duplication of computational paths in the presence of mutually blocking operations. This can slow compilation and cause significant code size increases.

Region scheduling transformations are also elementary. Because they are based on the program dependence graph, motion of operations in region scheduling should be faster than in percolation scheduling. Algorithms based on them have however reported only limited speedups [28].

The trace scheduling effort has shown that VLIWs can yield good speedups for numerical programs [18]. However only limited speedups have been reported for programs which are non-numerical in nature [21,43].

1.4 Contributions

The problems of current VLIW techniques can be summarized as large increase in object code size, lengthy compilation time and restricted speedup for non-numerical type of programs. Because of these shortcomings the dissertation investigates whether some of these limitations are inherent. More in particular the dissertation extends the model and results of scheduling theory to account for cyclicity and branching capabilities present in sequential programs [10,25].

The work is theoretical in nature in that results put forward are substantiated by formal proofs rather than benchmarking evidence. For one thing, negative results that pertain to the computational nature of a model cannot be disproved by an implement & benchmark approach. Furthermore as accurate benchmark selection is difficult and a continuously evolving endeavor, positive results substantiated by implement & benchmark may be incorrect. A purely theoretical approach can also be fallacious. Because reality is often complex and peculiar, formal models are either incomplete or intractable. Inevitably simplifying assumptions have to be made and there is risk of oversimplification. Furthermore theoretical results may hide expensive implementation costs. Ideally theory and practice should complement one another as it is the case in the physical sciences: experiments, in this case implement & benchmarking, should be used to validate the theoretical model. As such this thesis is only a first step in this direction.

The machine model considered comprises m identical processors or functional units operating synchronously and in parallel. More specifically every execution cycle r operations are dispatched to r different processors ($1 \leq r \leq m$). No preemption is allowed: once started, an operation has to be executed till its completion. The set of operations that are being processed in a same cycle is termed an instruction. Within an instruction conditional branches are arranged to form a decision tree that specifies what set of operations should be executed next. This branching model is more general than the one needed for trace scheduling and adopted in the TRACE machines [12]. It is directly inspired from the branching paradigms of Karplus & Nicolau and Ebcioğlu [33,15]. The machine is assumed to be combinatorial that is, an operation can be executed by a processor only if the processor is idle. Results very similar to those stated in this work can be obtained in the case of pipelined processors. Because operations are seen as atomic entities with no mention of variables, no algebraic manipulations of the input program are considered. Furthermore constraints in the number of available registers are ignored. Certainly if the machine is well balanced register availability should not hinder parallel performance.

To extract fine grained parallelism the VLIW paradigm advocates the use of static schedules, i.e. schedules generated at compile time. With the appearance of superscalar processors such as the Intel i860, Intel 80960CA or the IBM RIOS chip set, dynamic extraction of fine grained parallelism is regaining momentum. This concept dates back to the 1960s where machines like the IBM 360/91 or the CDC 6600 [8,57] provided hardware mechanisms to exploit operation level parallelism automatically at run time (dynamic schedules).

Another contribution of this dissertation is the time performance comparison of static and dynamic schedules. Because infinite operation lookahead is allowed, the dynamic schedules introduced are more of a theoretical gadget and are not likely to be generated by realistic hardware designs. The purpose of this comparison, however, is to show the gap between what is statically feasible and what is theoretically possible. Whether this gap can actually be exploited in practice by a dynamic scheduler is beyond the scope of this work. It is important to remark that the operation ordering which must be preserved at compile time can be significantly more constraining than at run time. In fact static schedulers may have to make overwhelming dependence assumptions if excessive aliasing is present in the sequential program. Indeed only dynamic operation-level parallelism was measured in program traces by Riseman & Foster and Fisher & Nicolau. To make a somewhat fair comparison between static and dynamic schedulers it is assumed that the set of operation dependences is the same. Because dynamic schedules are allowed infinite lookahead and static schedules are given the same run-time power as dynamic ones, static and dynamic schedules differ only in the presence of loops. In fact loop schedules generated statically must possess a certain regularity that dynamic ones need not have. Given this distinction it is shown in chapter 4, section 4.6 that in the presence of arbitrary dependences dynamic schedules can severely outperform static ones. When dependences are periodic, however, it is shown in chapter 4, section 4.6 for straight line loops and chapter 6, section 6.5 for loops with branches, that static schedules are as good as dynamic ones.

Two shortcomings of current VLIW scheduling techniques are code explosion and limited speedups achieved for non-numerical type of programs. One of our contributions is investigating whether pragmatic research efforts in VLIWs are undermined by inherent bottlenecks. Regarding code explosion none of the algorithms or implementations presented so far guarantees the absence of awesome increase in code size. Examples of exponential increases have been demonstrated for each of the individual algorithms but none is inherent; that is, increases in code size pertain to the nature of the algorithm used rather than the essence of VLIWs. In chapter 5, section 5.6 we prove the existence of sequential programs for which no algorithm can guarantee good speedups unless the schedule generated has exponential code size. Thus space and time performance of VLIW programs seem, in the worst case, to be antagonistic.

All VLIW scheduling algorithms implemented have reported only limited speedups for programs which are non-numerical in nature [21,43]. In chapter 5, section 5.6 it is shown that in the absence of profiling information or when branching probabilities are only slightly skewed, conditional jumps inhibit exploitable parallelism, regardless of whether the extraction process is static or dynamic. More specifically it is shown that if the input program has a high fraction of conditionals on every path and the target machine can execute m conditional jumps every cycle then only a speedup of roughly $\log_2 m$ can be achieved. If branching probabilities are skewed then better speedups are possible. However as the skewing increases the general branching model degenerates into the trace scheduling model.

In chapter 4, section 4.7 the problem of pipelining straight line loops for finite resource VLIWs is tackled. A novel strategy that deletes certain critical edges from the loop dependence graph is devised. Such edge deletion renders the dependence graph acyclic and can be executed in $O(|V|\cdot|E|)$ time where $|V|$ and $|E|$ respectively denote the number of vertices and edges in the loop dependence graph. The transformed dependence graph can be used to generate a loop schedule that is within a small constant factor of the optimum.

In chapter 4, sections 4.5 and 4.6, some mathematical properties of periodic and optimal schedules for straight line loops are explored.

When generating a VLIW instruction, statically or dynamically, it is often the case that operations from different computational paths may be available for execution. In the case where available operations cannot all be executed together a selection criterion must be employed. A simple heuristic is produced in chapter 5, section 5.7 and is proven to perform between a constant and a logarithmic factor from the optimum, depending on the skewing of branch probabilities.

Several researchers have posed the problem of optimal parallelization of arbitrary loops for VLIWs with unlimited resources [4,59]. Chapter 6, section 6.4 shows the existence of simple loops containing conditional jumps for which there cannot exist static schedules with time optimal performance on VLIWs with unlimited resources.

1.5 Dissertation Overview

The subdivision of the thesis does not correspond to the above partitioning of research contributions. We have tried to give a coherent organization by adding increasingly powerful capabilities to the original model of scheduling theory. After presenting simple mathematical notions and standard graph theoretical concepts and terminology, chapter 3 introduces task systems, the original model of scheduling theory and our starting point. In chapter 4 cyclicity is added to the basic task system model (cyclic task systems). Chapter 5 is independent of chapter 4 and adds conditionals to task systems (branching task systems). The last chapter depends on chapters 4 and 5. It adds cyclicity to the model of chapter 5 (cyclic branching task systems).

Chapter 2

Preliminaries

This chapter establishes common terminology for simple mathematical and finite/infinite graph theoretical concepts.

2.1 Mathematical Notions

Let A, B be two sets and f a function mapping A into B. f is said to be injective if for all $a_1, a_2 \in A$, $f(a_1) = f(a_2)$ implies $a_1 = a_2$. f is said to be surjective if for any $b \in B$ there exists some $a \in A$ such that $f(a) = b$. If f is both surjective and injective then f is said to be bijective.

Let S be some set. S is said to be countable if and only if there exists an injective function mapping S into the integers. If S is finite then $|S|$ denotes its cardinality. An equivalence relation $\equiv$ defined on S is a reflexive ($\forall\, x \in S \quad x \equiv x$), symmetric ($\forall\, x, y \in S \quad x \equiv y$ implies $y \equiv x$) and transitive relation ($\forall\, x, y, z \in S \quad x \equiv y$ and $y \equiv z$ implies $x \equiv z$). A partial order $\sqsubseteq$ defined on S is an irreflexive, anti-symmetric ($\forall\, x, y \in S \;\; x \sqsubseteq y$ implies $y \not\sqsubseteq x$) and transitive relation .

The symbol "$\log_2$" denotes the base 2 logarithm and "ln" the base e logarithm. The symbol "$\exists!$" means "there exists one and only one". The symbol $\emptyset$ denotes the empty set. The term positive number denotes a number strictly greater than 0. For a real number x, $\lfloor x \rfloor$ denotes the floor of x, that is the greatest integer smaller than x and $\lceil x \rceil$ the ceiling of x, that is the smallest integer greater than x.

2.2 Graph Theoretical Concepts and Terminology

A directed graph or di-graph G, is a pair $G = (V, E)$ where V is a set whose elements are called vertices and E a subset of $V \times V$ whose elements are called edges. If V is finite then G is said to be a finite graph, otherwise G is said to be infinite.

For $e = (u, v) \in E$ one says that e goes from u to v. Vertices u and v are respectively referred to as the tail and head of e. The out-degree (respectively the in-degree) of a vertex $v \in V$ is the number of edges in E that have v as their tail (respectively the head). Two edges are said to be consecutive if the head of the first coincides with the tail of the second.

A path P of G is a not necessarily finite sequence of consecutive edges. A sub-sequence of edges in P is called a sub-path of P. Note that every path is a sub-path of itself. The length of a finite path is the number of edges it contains. If a vertex $v \in V$ is the head or tail of some edge in P, P is said to traverse v. A path P is said to be simple if no vertex is the head of two edges in P. Given two vertices u and v of G, a path which first traverses u and then v is said to go from u to v. One also says that v is reachable from u. Vertex u is said to precede or be a predecessor of v (alternatively v is said to succeed or be a successor of u) if and only if there exists a path from u to v and no path from v to u. If this path is of length one then u is said to be an immediate predecessor of v (v is an immediate successor of u). A path going from a vertex to itself is called a cycle.

If no path in G is a cycle, G is said to be acyclic, otherwise G is cyclic. If there is a path from any vertex to any other vertex of G, G is said to be strongly connected.

A subgraph H of G is a pair $H = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq V' \times V' \cap E$. A subgraph H of G is said to be traversed by a path P if and only if P traverses some of H vertices. The subgraph induced by a set $V' \subseteq V$ of vertices is defined as $(V', V' \times V' \cap E)$.

If there exists two subgraphs of G such that the head and tail of every edge in E are always in the same subgraph then G is said to be disconnected, otherwise G is said to be connected.

G is single entry if it has exactly one root that is a vertex with in-degree zero. G is single exit if it has exactly one sink that is a vertex with out-degree zero.

A di-graph G is said to be weighted if each edge e of G is associated with some number, called the weight of e. Given a path P of G the sum of the weights of the edges in P is termed the weight of P.

A di-graph T is said to be a rooted tree if T is single entry and every vertex, except for the root, has in-degree one. The sinks of a tree are also called leaves. A vertex which is not a leaf is said to be internal. The immediate successors of a non sink vertex v of T are called the children of v. The height of a finite tree is the length of the longest path from the root to a leaf. If T has height h, T is said to have $h+1$ levels. The root of T is at level 0, the vertices which are l edges away from the root are at level l. A subtree T' of T is a connected subgraph of T such that all the vertices which are reachable from the root of T' in T belong to T'. A tree where every vertex, except for the leaves, has out-degree two is called a binary tree. A binary tree is said to be complete or completely balanced if all the paths from the root to a leaf have the same length.

For concepts and terminology on depth first search please refer to [1].

Chapter 3

Task Systems

This chapter presents task systems, the original model of scheduling theory [10,25]. Task systems are the starting point of our work. One point deserves notice. Task systems are usually defined as finite entities. In this chapter task systems are allowed to be infinite. This will be useful when introducing cyclicity in chapter 4.

3.1 Preliminaries

The existence of some infinite countable set $\mathcal{O}$ whose elements are called operations is assumed. An operation models some atomic action or task that has to be accomplished.

In this work time is considered as a discrete rather than continuous entity. Time instants will therefore be modeled as positive integers, which are considered to be multiples of some predefined time unit called an execution cycle or simply cycle.

3.2 Definition

Informally a task system is a collection of several inter-dependent operations all of which must be executed in order to complete the task system.

Definition 3.2.1 *A task system T, or simply task, is a triple $T = (O, \tau, \prec)$ where:*

1. *O is the operation set of T, a non empty subset of $\mathcal{O}$. Note that O could be infinite.*
2. *τ is the duration function of T, a function mapping O into the strictly positive integers. For $op \in O$, $\tau(op)$ is the number of cycles required to execute op.*

3. $\prec$ is the dependence relation of T, a partial order on O. For $op, op' \in O$, $op \prec op'$ means that op must finish executing before op' can start: op' is said to depend on op.

The task system T is said to be finite if and only if its operation set O is finite. T is said to be infinite otherwise.

The partial order $\prec$ can be represented by a di-graph D with vertex set O. Given two operations $op, op' \in O$, op is an immediate predecessor of op' in D if and only if $op \prec op'$ and there is no $op'' \in O$ such that $op \prec op'' \prec op'$. D is called the dependence graph of T. The graph D is infinite if and only if T is infinite.

3.3 Schedules

The machine model $\mathcal{M}(m)$ considered, $1 \leq m$, comprises m identical processors operating synchronously and in parallel. The machine $\mathcal{M}(\infty)$ contains an unbounded number of processors. There is no preemption: once started, an operation has to be executed without interruption. Furthermore no pipelining is allowed, processors can execute a single operation at a time: $\mathcal{M}(m)$ executes at most m operations every cycle. The set of operations that are being processed in a same cycle is termed an instruction. More formally:

Definition 3.3.1 *A m-processor instruction I, or more briefly m-instruction, is a finite, possibly empty, subset of operations ($I \subset \mathcal{O}$) such that $|I| \leq m$. The operations contained in I are said to be executing in I. If $m = \infty$ then I can contain an unbounded but finite number of operations.*

The notion of machine schedule can now be presented. Informally a schedule for $\mathcal{M}(m)$ is a lay out of the operations as they have to be executed by $\mathcal{M}(m)$.

Definition 3.3.2 *A m-processor schedule $C = (I_1, \cdots, I_i, \cdots)$, or more briefly m-schedule, is a countable sequence of m-instructions. The i-th instruction of C $(1 \leq i)$ is assimilated to the i-th execution cycle of $\mathcal{M}(m)$. The starting cycle in C of an operation op is defined as $t(op, C) = \min\{k \mid \forall\, 1 \leq i < k \;\; op \notin I_i \text{ and } op \in I_k\}$. If C contains infinitely* *many instruction the length of C is said to be infinite, otherwise the number of instructions in C is called its length and is denoted $|C|$. When m is understood or unimportant C is just said to be a schedule.*

If C can be executed on $\mathcal{M}(m)$ then it can be executed on $\mathcal{M}(m')$ for $m \leq m'$. A m-schedule C is graphically represented by a table with m columns, each representing a processor, and $|C|$ rows, each representing an instruction. Because all processors are identical the actual assignment of operations to processors is unimportant. See figure 3.1(b) for an example.

In the sequel m denotes the number of available processors. Such number is assumed to be finite, unless otherwise stated.

3.4 Admissibility

Given some task system T and the target machine $\mathcal{M}(m)$, one would like to formalize the notion of admissibility, that is the conditions under which a schedule made up of operations in T and executing on $\mathcal{M}(m)$ respects T's dependences.

Definition 3.4.1 *Let $T = (O, \tau, \prec)$ be some task system. A m-schedule $C = (I_1, \cdots, I_i, \cdots)$ is said to be m-admissible for T, denoted $C \overset{m}{\Leftrightarrow} T$, if and only if:*

1. *The instructions of C solely contain operations in O:*

$$\forall\, i \geq 1 \quad I_i \subseteq O$$

2. *Every $op \in O$ executes only from cycle $t(op, C)$ to $t(op, C) + \tau(op) \Leftrightarrow 1$ included:*

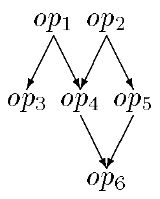
$$\forall op \in O \quad \forall 0 \leq j < \tau(op) \quad op \in I_{t(op,C)+j} \quad \textit{and} \quad \forall i \geq t(op, C)+\tau(op, C) \quad op \notin I_i$$

3. *No $op \in O$ starts executing until all operations on which it depends have completed:*

$$\forall\, op, op' \in O \quad op \prec op' \;\Rightarrow\; t(op, C) + \tau(op) \leq t(op', C)$$

If T is finite and C has smallest possible length, C is said to be m-optimum for T.

Let $T = (O, \tau, \prec)$ be the task system with operations set $O = \{op_1, op_2, op_3, op_4, op_5, op_6\}$, duration function $\tau(op) = 1$ for every $op \in O$ and dependence relation $\prec$ portrayed in figure 3.1(a). Figure 3.1(b) gives the only 3-optimum schedule for T.



(a)

op_1	op_2	
op_3	op_4	op_5
op_6		

(b)

Figure 3.1: (a) Dependence graph for T. (b) The 3-optimum schedule for T.

3.5 Approximating Optimality for Finite Task Systems

For any finite m, the problem of generating an m-optimum schedule for a finite task system is NP-complete [41]. If operations are restricted to have the same duration a polynomial time optimal algorithm for the case $m = 2$ was presented by Coffman and Graham [11]. The problem remains open for any fixed $m \geq 3$ that is, NP-hardness has not been proved or disproved [23]. If however m is considered to be a parameter the problem becomes NP-hard [41].

The algorithm of Coffman and Graham builds on the list scheduling framework. List scheduling algorithms work as follows. The operations in the task system are implicitly ordered in a priority list. Instructions are generated by repeatedly scanning and deleting operations from the priority list. More specifically for each free slot in the instruction that is being built, list scheduling selects the first operation in the priority list that is ready and schedules it in that slot. An operation is said to be ready if all its predecessors in the dependence graph have finished executing. Note that for $m = \infty$ any list scheduling algorithm yields optimum schedules. The Coffman-Graham algorithm builds the priority list according to the structure of the dependence graph. This guarantees a final running time of at most $(2 \Leftrightarrow 2/m)$ times the optimum for task systems in which operations all have the same duration [39]. If the priority list is built at random this factor increases to $(2 \Leftrightarrow 1/m)$ [10]. The NP-hardness proof given by Lenstra and Rinnooy Kan [41] implies that, unless P = NP, no polynomial time algorithm can approximate m-optimality for arbitrary m, by less than a factor 4/3.

Chapter 4

Cyclic Task Systems

This chapter models the behavior of a system which must continuously execute a fixed set of operations. A cyclic task system formalizes the intuitive notion of a straight line loop. Theoretical work in cyclic scheduling is not novel. The studies of Karp and Miller, Reiter, Romanovskii are some examples [32,50,52,27]. All previous works, however, assume periodicity of dependences which this chapter does not.

4.1 Preliminaries

Let $\mathbf{Z}$ denote the set of integers. A new value denoted ∞ is introduced. This value is by definition greater than any integer: $x < \infty$ for any $x \in \mathbf{Z}$. Addition and multiplication are extended to account for ∞. For $x \in \mathbf{Z}$, $x + \infty = x \cdot \infty = \infty$. For $a, b \in \mathbf{Z} \cup \{\infty\}$, $[a, b]$ denotes the set of integers between a and b: $[a, b] = \{i \in \mathbf{Z} \mid a \leq i \leq b\}$. Thus $[1, \infty]$ denotes the set of positive integers.

Let S be some set and $[a, b]$ the set previously defined, the set product of S and $[a, b]$ is denoted $S[a, b]$. An element of such set is designated $s[i]$: $S[a, b] = \{s[i] \mid s \in S \text{ and } i \in [a, b]\}$.

4.2 Definition

Informally a cyclic system $\mathcal{L}$ can be seen as an infinite sequence of tasks, called iterations. Iterations share the same operation set O and duration function τ but may have different dependence relations. Operations in one iteration may depend on operations in preceding iterations.

Definition 4.2.1 *A cyclic task system $\mathcal{L}$, or more briefly cyclic system, is an infinite task system $\mathcal{L} = (O[1,\infty], \tau, \prec)$ such that:*

1. *O is a finite set called the core operation set of $\mathcal{L}$. The operation set of $\mathcal{L}$ is $O[1,\infty]$. For $op \in O$, $op[i]$ denotes operation op executed during iteration i. The integer i is termed the iteration index of $op[i]$.*
2. *For all $op \in O$ and $i, j \in [1,\infty]$, $\tau(op[i]) = \tau(op[j])$. In the sequel such unique number is denoted $\tau(op)$.*
3. *For all $op, op' \in O$ and $i, j \in [1,\infty]$, $op[i] \prec op'[j]$ implies $i \leq j$.*

For any $a, b \geq 1$ the function τ and the partial order $\prec$ can trivially be restricted to $O[a,b]$. For $i \geq 1$ the task system $(O[i,i], \tau, \prec)$ is called the i-th iteration of $\mathcal{L}$ and the task system $\mathcal{L}(n) = (O[1,n], \tau, \prec)$ is termed the n-instance of $\mathcal{L}$.

The task system $\mathcal{L}(n)$ formalizes the case where $\mathcal{L}$ is required to iterate n times in order to complete.

Consider the cyclic system $\mathcal{L}'$ where each iteration is formed by x consecutive iterations of $\mathcal{L}$, for some $x > 1$. Intuitively $\mathcal{L}'$ is the same cyclic system as $\mathcal{L}$ except that its granularity has increased by a factor x. Informally one says that $\mathcal{L}'$ is obtained by unrolling $\mathcal{L}$, x times. The following definition makes this notion more precise.

Definition 4.2.2 *Let $\mathcal{L} = (O[1,\infty], \tau, \prec)$ be some cyclic system. For $x \geq 1$ the x-unrolling of $\mathcal{L}$ is defined to be the cyclic system $u^x(\mathcal{L}) = (O^x[1,\infty], \tau, \prec^x)$ where:*

1. *$O^x = O[1,x]$ and $O^x[1,\infty] = \{op[i][k] \mid i \in [1,x]$ and $k \in [1,\infty]\}$.*
2. *For $op[i], op'[j] \in O^x$ and $k_1, k_2 \in [1,\infty]$, $op[i][k_1] \prec^x op'[j][k_2]$ if and only if $op[i + x \cdot (k_1 \Leftrightarrow 1)] \prec op'[j + x \cdot (k_2 \Leftrightarrow 1)]$.*

Note that $u^1(\mathcal{L}) = \mathcal{L}$ and $u^y(u^x(\mathcal{L})) = u^{y \cdot x}(\mathcal{L})$. Because of point 2 in the previous definition $op[i][k]$ and $op[i + x \cdot (k \Leftrightarrow 1)]$ denote the same operation.

4.3 Dependence Distances and Graphs

Let $\mathcal{L}$ be some cyclic system with core operation set O. Given any two operations $op, op' \in O$ it will become important, for the purpose of generating regular schedules, to identify the most stringent dependence between op and op'. To this purpose the notion of dependence distance between op and op' is introduced.

Definition 4.3.1 *Let $\mathcal{L}$ be some cyclic system with core operation set O. For all $op, op' \in O$ the dependence distance from op to op', denoted $d(op, op')$, is defined as:*

$$d(op, op') = \min\{d \mid \forall\, i \geq 1 \;\; \exists\, k \geq i \quad op[k] \prec op'[k+d]\}$$

where the minimum of the empty set is equal to ∞ by definition. Furthermore if

$$\forall\, op, op' \in O \quad \forall\, i \geq 1 \qquad op[i] \prec op'[i + d(op, op')]$$

one says that $\mathcal{L}$ is recurrent or has recurrent dependences.

The idea behind $d(op, op')$ is to disregard dependences that do not repeat infinitely many times, as these do not pertain to the repetitive nature of cyclic systems. Informally this means that when building a schedule $\mathcal{C}$ for $\mathcal{L}$, these non repeating dependences can be accounted for in an irregular schedule preceding $\mathcal{C}$. Indeed one can show that:

Property 4.3.1 *For any cyclic system $\mathcal{L} = (O[1, \infty], \tau, \prec)$ there exists a positive integer n_0 such that*

$$\forall\, op, op' \in O \quad \forall\, i \geq n_0 \quad \forall\, d < d(op, op') \qquad op[i] \not\prec op'[i+d]$$

Proof: Let $op, op' \in O$. Because of the definition of $d(op, op')$ there must exist a positive integer $n(op, op')$ such that

$$\forall\, i \geq n(op, op') \;\; \forall\, d < d(op, op') \qquad op[i] \not\prec op'[i+d]$$

or else $d(op, op') < d(op, op')$. By letting n_0 be the greatest of all the $n(op, op')$ the desired result is obtained. $\square$

Throughout the remainder of this work one can assume, without loss of generality, that for every cyclic system $\mathcal{L}$ the n_0 of property 4.3.1 is 1. In fact if this is not the case one can split $\mathcal{L}$ into the finite task system $\mathcal{L}(n_0 \Leftrightarrow 1) = (O[1, n_0 \Leftrightarrow 1], \tau, \prec)$ and the cyclic system $(O[n_0, \infty], \tau, \prec)$ and treat each separately.

Because dependence distances are a worst case estimate of $\mathcal{L}$'s dependences, it could be that $\mathcal{L}$ is recurrent while for every $x > 1$, $u^x(\mathcal{L})$ is not.

Example 4.3.1 Let $\mathcal{L}$'s core operation set be $\{op, op'\}$ and define the dependence relation of $\mathcal{L}$ as:

$$\forall\, i \geq 1 \qquad op[i] \prec op'[i] \text{ and } op[i] \prec op'[i + \pi(i)]$$

where π is a some function randomly mapping i into $\{0, 1\}$. $\mathcal{L}$ is clearly recurrent while for all $x > 1$, $u^x(\mathcal{L})$ is not.

The converse is also true, namely $u^2(\mathcal{L})$ could have recurrent dependences while $\mathcal{L}$ does not. For instance

Example 4.3.2 Let $\mathcal{L}$'s core operation set be $\{op, op'\}$ and define the dependence relation of $\mathcal{L}$ as:

$$\forall\, i \geq 1 \qquad op[2 \cdot i] \prec op'[2 \cdot i]$$

Despite the fact that recurrence is not preserved by unrolling it is possible to show the following property.

Property 4.3.2 *Let $\mathcal{L}$ be a cyclic system with recurrent dependences and core operation set O. Then*

$$\forall\, op, op', op'' \in O \qquad d(op, op') \leq d(op, op'') + d(op'', op')$$

Proof: Because dependences are recurrent for any $i \geq 1$ it must be that $op[i] \prec op''[i + d(op, op'')] \prec op'[i + d(op, op'') + d(op'', op')]$ which implies the above result. $\Box$

The above result is what allows dependences of a recurrent cyclic system to be viewed as a weighted graph. A formal account of this remark will be given in theorem 4.3.3. Simple counter examples show that the above proposition does not generalize to arbitrary cyclic systems.

Example 4.3.3 Let $\mathcal{L}$'s core operation set be $\{op, op'\}$ and define the dependence relation of $\mathcal{L}$ as:

$$\forall\, i \geq 1 \quad op[2 \cdot i] \prec op'[2 \cdot i] \quad \text{and} \quad op'[2 \cdot (i \Leftrightarrow 1) + 1] \prec op[2 \cdot i + 1]$$

From the dependences it follows that $d(op, op') = 0$, $d(op', op) = 2$ and $d(op, op) = \infty$. Clearly $d(op, op) \not\leq d(op, op') + d(op', op)$.

For some $x \geq 1$, let us now consider the x-unrolling of an arbitrary cyclic system $\mathcal{L}$. There are a factor x^2 more dependence distances in $u^x(\mathcal{L})$ than in $\mathcal{L}$. In fact there is a dependence distance $d^x(op[i], op'[j])$ $(i, j \in [1, x])$ for each copy of op, op' in $u^x(\mathcal{L})$:

$$d^x(op[i], op'[j]) = \min\{d \mid \forall\, l \geq 0 \;\; \exists\, k \geq l \quad op[i + k \cdot x] \prec op'[j + (k + d) \cdot x]\}$$

Some software pipelining algorithms developed for VLIWs assume dependence distances to be 0, 1 or ∞. If the input loop does not exhibit such behavior, unrolling is invoked to decrease dependence distances. The following theorem relates dependence distances of a cyclic system $\mathcal{L}$ to those of $u^x(\mathcal{L})$. If O is $\mathcal{L}$'s core operation set, the theorem states that at least $x \cdot |O|$ of the $x^2 \cdot |O|^2$ dependence distances must decrease but not all $x^2 \cdot |O|^2$ dependence distances need to. Interestingly theorem 4.3.2 will show that this is the strongest possible result. The significance of these two theorems is that unrolling does not systematically guarantee smaller dependence distances. Consequently a loop pipelining algorithm must explicitly account for dependence distances greater than 1.

Theorem 4.3.1 *Let $\mathcal{L}$ be some cyclic system, O its core operation set and x some strictly positive integer. Then*

1. $\forall\, op, op' \in O \;\; \forall\, i, j \in [1, x] \qquad \left\lfloor \dfrac{d(op, op')}{x} \right\rfloor \leq d^x(op[i], op'[j])$
2. $\forall\, op, op' \in O \;\; \exists\, i, j \in [1, x] \qquad d^x(op[i], op[j]') \leq \left\lceil \dfrac{d(op, op')}{x} \right\rceil$

Proof: By contradiction. Let O^x be the core operation set of $u^x(\mathcal{L})$. Assume that $d^x(op[i], op'[j]) < \lfloor d(op, op')/x \rfloor$ for some $op[i], op'[j] \in O^x$. Since for any $l \geq 0$ there is some $k \geq l$ such that $op[i + k \cdot x] \prec op'[j + k \cdot x + d^x(op[i], op'[j]) \cdot x]$ it must be that

$$d(op, op') \leq d^x(op[i], op'[j]) \cdot x + j \Leftrightarrow i \leq (\lfloor d(op, op')/x \rfloor \Leftrightarrow 1) \cdot x + (x \Leftrightarrow 1) = \lfloor d(op, op')/x \rfloor \cdot x \Leftrightarrow 1$$

a contradiction.

Now assume that there exist two operations $op, op' \in O$ such that for every $i, j \in [1, x]$, $d^x(op[i], op'[j]) > \lceil d(op, op')/x \rceil$. This implies that

$$\exists\, l \geq 0 \quad \forall\, k > l \quad \forall\, d \in [0, \lceil d(op, op')/x \rceil] \qquad [op[i + k \cdot x] \not\prec op'[j + k \cdot x + d \cdot x]$$

and therefore

$$d(op, op') > \lceil d(op, op')/x \rceil \cdot x \geq d(op, op')$$

a contradiction. □

As it has been previously mentioned the second claim of theorem 4.3.1 is the strongest one can state. In fact

Theorem 4.3.2 *There exists a cyclic system $\mathcal{L}$ with core operation set $\{op\}$ such that:*

$$\forall\, x \geq 1 \quad \exists\, t \in [1, x] \quad 1 < d^x(op_t, op_t) < \infty$$

Proof: Let $\mathcal{L} = (O[1, \infty], \tau, \prec)$ where $O = \{op\}$, the duration function τ is unimportant and $\prec$ is such that

$$\forall\, i \geq 0 \ \forall\, j \geq 0 \quad op[2^i + j \cdot 2^{i+1}] \prec op[2^i + (j+1) \cdot 2^{i+1}]$$

These dependences are portrayed in figure 4.1.

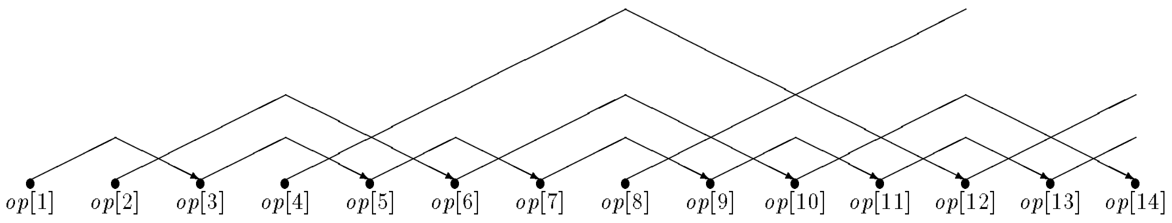


Figure 4.1: Dependences of some cyclic system.

Let $x \geq 1$ and 2^t the biggest power of two which divides x, that is $x = \lambda 2^t$ and λ is odd. It is shown that $d^x(op_{2^t}, op_{2^t}) = 2$. For obvious reasons $d^x(op_{2^t}, op_{2^t}) \geq 1$. Assume that $d^x(op_{2^t}, op_{2^t}) = 1$. Then

$$\exists\, k \geq 1 \quad op[2^t \cdot (1 + k \cdot \lambda)] \prec op[2^t \cdot (1 + (k+1) \cdot \lambda)]$$

In turn this implies that

$$\exists\, i,j \geq 0 \;\; \exists\, q \geq 1 \qquad \begin{aligned} 2^i \cdot (1+2j) &= 2^t \cdot (1+k\cdot\lambda) \\ 2^i \cdot (1+2\cdot(j+q)) &= 2^t \cdot (1+(k+1)\cdot\lambda) \end{aligned}$$

However because λ is odd the above two equations cannot be simultaneously satisfied, leading to a contradiction. Thus $d^x(op_{2^t}, op_{2^t}) \geq 2$. Finally because

$$\forall k \geq 0 \quad op[2^t+2\cdot k\cdot x] = op[2^t+k\cdot\lambda\cdot 2^{t+1}] \prec op[2^t+(k\cdot\lambda+\lambda)\cdot 2^{t+1}] = op[2^t+2\cdot k\cdot x+2\cdot x]$$

it must be that $d^x(op_{2^t}, op_{2^t}) \leq 2$. □

Dependence distances of a cyclic system $\mathcal{L}$ can be portrayed by a weighted dependence graph D whose vertex set is the core operation set of $\mathcal{L}$. D has an edge e from op to op' if and only if for all op'', $d(op, op') < d(op, op'') + d(op'', op')$. The weight of e is set to $d(op, op')$. The weight of a finite path Q in D is denoted $d(Q)$.

Because of property 4.3.2, if $\mathcal{L}$ has recurrent dependences, D provides a faithful characterization of the most stringent dependences. In fact

Theorem 4.3.3 *Let $\mathcal{L} = (O[1,\infty], \tau, \prec)$ be a recurrent cyclic system and D the dependence graph of $\mathcal{L}$. Let Q be a path in D from op to op'. Then*

$$\forall\, i \geq 1 \quad op[i] \prec op'[i + d(Q)]$$

Conversely if $op[i] \prec op'[j]$ for some $op[i], op'[j] \in O[1,\infty]$ then there exists a path Q in D which goes from op to op' and is such that $d(Q) \leq j \Leftrightarrow i$.

Proof: Let $op, op' \in O$ and $Q = (op = op_1, op_2, \cdots, op_q = op')$ a path from op to op' in D. Because $\mathcal{L}$ is recurrent, for any $i \geq 1$

$$op_1[i] \prec op_2[i + d(op_1, op_2)] \prec \cdots \prec op_q[i + d(op_1, op_2) + \cdots + d(op_{q-1}, op_q)]$$

and therefore $op[i] \prec op'[i + d(Q)]$.

Conversely assume $op[i] \prec op'[j]$ for some $op[i], op'[j] \in O[1,\infty]$. Because $\mathcal{L}$ has recurrent dependences it must be that $d(op, op') \leq j \Leftrightarrow i$. To prove the second claim it is sufficient to show the existence of a path Q in D going from op to op' and such that $d(Q) = d(op, op')$. If there exists no $op'' \in O$ such that $d(op, op'') + d(op'', op') \leq d(op, op') < \infty$ then by definition there is an edge from op to op' whose weight is $d(op, op')$. Otherwise by property 4.3.2 one must have $d(op, op'') + d(op'', op') = d(op, op')$. When the above reasoning is repeated inductively one obtains:

$$d(op_1, op_2) + d(op_2, op_3) + \cdots + d(op_{q-2}, op_{q-1}) + d(op_{q-1}, op_q) = d(op, op')$$

where $op_1 = op$, $op_q = op'$ and for $1 < j < k \leq q$ no two op_j, op_k are equal for otherwise

$$\begin{aligned} d(op_{j-1}, op_k) &= d(op_{j-1}, op_j) + \cdots + d(op_{k-1}, op_k) \\ &= d(op_{j-1}, op_j) + d(op_j, op_k) \\ &= d(op_{j-1}, op_k) + d(op_k, op_k) \quad \text{if } op_j = op_k \end{aligned}$$

Since $d(op_k, op_k)$ is by definition always strictly positive, the last equation is a contradiction. As the op_j for $1 < j \leq q$ are all distinct and there can be at most $|O|$ such operations one cannot repeat the inductive step indefinitely and a point is reached where

$$\forall\, 1 < j \leq q \quad \forall\, op'' \in O \quad d(op_{j-1}, op_j) < d(op_{j-1}, op'') + d(op'', op_j)$$

Thus there exists an edge in D from op_{j-1} to op_j with weight $d(op_{j-1}, op_j)$. By letting $Q = (op = op_1, \cdots, op_q = op')$ the desired result is obtained. □

The previous claim allows the reconstruction of the dependence function d of a recurrent cyclic system from its dependence graph. In fact let op and op' be two vertices in this graph, then $d(op, op')$ is the distance of the shortest path from op to op', or ∞ if no such path exists.

Dependence graphs of recurrent cyclic systems will play an important role in the coming sections. The strong components of these dependence graphs will be of special importance.

4.4 Schedules for Cyclic Systems

Let $\mathcal{L}$ be some cyclic system. If one knew in advance how many iterations of $\mathcal{L}$ will need to be executed, then a finite schedule for just that number of iterations could be explicitly generated. This is the best situation one could hope for and will be the starting point.

Definition 4.4.1 *Let $\mathcal{L}$ be some cyclic system and $\mathcal{C}$ a set of finite length m-schedules. $\mathcal{C}$ is said to be m-admissible for $\mathcal{L}$, denoted $\mathcal{C} \overset{m}{\looparrowright} \mathcal{L}$, if and only if for all $n \geq 1$ there exists exactly one m-schedule $C \in \mathcal{C}$ such that C is m-admissible for $\mathcal{L}(n)$, that is:*

$$\forall\, n \geq 1 \;\; \exists!\, C \in \mathcal{C} \quad C \overset{m}{\looparrowright} \mathcal{L}(n)$$

Such unique schedule is denoted $\mathcal{C}(n)$.

If $\mathcal{L}$ has recurrent dependences any sufficiently long schedule $C \in \mathcal{C}$ must possess some interesting properties. For instance C must contain blocks of instructions where each strong component in $\mathcal{L}$'s dependence graph repeats exactly an integral number of times. The notion of integral repetition of a strong component is first formally defined and the claim is then proven.

Definition 4.4.2 *Let $\mathcal{L}$ be a cyclic system, SC a strong component in $\mathcal{L}$'s dependence graph, C a schedule and I_1, I_2 two instructions in C. SC is said to repeat an integral number of times in C between I_1 and I_2 if and only if there exists an integer $k \geq 0$ such that every op in SC executes for $k \cdot \tau(op)$ cycles between I_1 and I_2, I_1 included I_2 excluded.*

Property 4.4.1 *Let $\mathcal{L} = (O[1,\infty], \tau, \prec)$ be a cyclic system with recurrent dependences, C an m-admissible schedule for $\mathcal{L}(n)$, for some $n \geq 1$, and W a subset of C's instructions. There exists a constant c such that if $|W| > c$ then there exist $I_1, I_2 \in W$ for which every strong component in $\mathcal{L}$'s dependence graph repeats an integral number of times in C between I_1 and I_2.*

Proof: Let $s = \sum_{op \in O} \tau(op)$. For each instruction I of C define the function $\lambda(I)$ which maps I into a vector of s non-negative integers. There are $\tau(op)$ components in $\lambda(I)$ associated with an operation $op \in O$, each corresponding to a particular execution cycle of op. The component of $\lambda(I)$ associated with the i-th execution cycle of op is denoted $\lambda(I)(op, i)$. It represents the number of instructions in C occurring before I where the ith cycle of op is executing.

Let $d_{\max} = \max\{d(op, op') \mid op, op' \in O, \quad d(op, op') < \infty\}$ and $\alpha = 1 + 2 \cdot \max(m, d_{\max})$. Clearly for every $I \in C$:

$$\lambda(I)(op, 1) \geq \cdots \geq \lambda(I)(op, \tau(op))$$

Furthermore because C is an m-schedule

$$\lambda(I)(op, \tau(op)) \geq \lambda(I)(op, 1) \Leftrightarrow m$$

and therefore for any $i, j \in [1, \tau(op)]$

$$|\lambda(I)(op, i) \Leftrightarrow \lambda(I)(op, j)| \leq m < \alpha/2$$

Let SC be a strong component in $\mathcal{L}$'s dependence graph and op , op' any two distinct operations in SC. Then

$$\lambda(I)(op, \tau(op)) \geq \lambda(I)(op', 1) \Leftrightarrow d(op, op') \;\text{ and }\; \lambda(I)(op', \tau(op')) \geq \lambda(I)(op, 1) \Leftrightarrow d(op, op)$$

and consequently for any i and j such that $i \in [1, \tau(op)]$ and $j \in [1, \tau(op')]$

$$|\lambda(I)(op, i) \Leftrightarrow \lambda(I)(op', j)| < \alpha/2$$

Let $\lambda_{\text{mod}}(I)$ be the vector also of size s where each component is equal to the corresponding component in $\lambda(I)$ modulo α. There are at most α^s different vector values for $\lambda_{\text{mod}}(I)$. Thus if $|W| > \alpha^s$ two different instructions I_1 and I_2 in W must verify $\lambda_{\text{mod}}(I_1) = \lambda_{\text{mod}}(I_2)$. Without loss of generality assume that I_1 comes before I_2 in C. Because $\lambda_{\text{mod}}(I_1) = \lambda_{\text{mod}}(I_2)$ it must be that

$$\forall\, op \in O \quad \forall\, i \in [1, \tau(op)] \quad \exists\, k_i(op) \geq 0 \qquad \lambda(I_2)(op, i) \Leftrightarrow \lambda(I_1)(op, i) = k_i(op) \cdot \alpha$$

Thus for every strong component SC in $\mathcal{L}$'s dependence graph

$$\forall\, op, op' \in SC \quad \forall\, i \in [1, \tau(op)] \quad \forall\, j \in [1, \tau(op')]$$

$$\begin{array}{rcl} |k_i(op) \Leftrightarrow k_j(op')| \cdot \alpha & \leq & |\lambda(I_2)(op, i) \Leftrightarrow \lambda(I_2)(op, j)| + |\lambda(I_1)(op, i) \Leftrightarrow \lambda(I_1)(op, j)| \\ & < & \alpha \end{array}$$

and therefore $k_i(op) = k_j(op')$. Consequently

$$\exists\, k \geq 0 \quad \forall op \in SC \quad \forall\, i \in [1, \tau(op)] \qquad \lambda(I_2)(op, i) \Leftrightarrow \lambda(I_1)(op, i) = k \cdot \alpha$$

this implies that every operation in SC is executed for $k \cdot \alpha \cdot \tau(op)$ cycles between I_1 and I_2. $\square$

When generating VLIW instructions, statically or dynamically, it is often the case that several operations may be available for execution in a same instruction. In the case where available operations cannot all be executed together a selection criterion must be employed. For cyclic systems this selection is further complicated by that one may discriminate between $op[i]$ and $op[i+k]$. In the case of cyclic systems with recurrent dependences it is shown that performance is not affected if $op[i]$ is systematically preferred to $op[i+k]$. This is not always the case for cyclic systems with arbitrary dependences.

Property 4.4.2 *Let $\mathcal{L}$ be some recurrent cyclic system with core operation set O and let C be an m-admissible schedule for some n-instance, $\mathcal{L}(n)$ of $\mathcal{L}$. Then there exists a schedule $C' \overset{m}{\Leftrightarrow\!\!\rightarrow} \mathcal{L}(n)$ which is equal to C if iteration indices are disregarded and C' is such that*

$$\forall\, op \in O \quad \forall\, i \in [1, n \Leftrightarrow 1] \quad t(op[i], C') \leq t(op[i+1], C')$$

Proof: C' is constructed from C. Let $C = (I_1, \cdots, I_c)$ then $C' = (I'_1, \cdots, I'_c)$ where I'_j $(1 \leq j \leq c)$ comprises the same operations as I_j except for their iteration indices which may be different. The iteration indices of the operations in C' are assigned sequentially, that is for each $op \in O$ the first occurrence of op is assigned index 1, the second index 2, etc.

Clearly C' satisfies the stated requirements. It remains to show that $C' \overset{m}{\Leftrightarrow\!\!\rightarrow} \mathcal{L}(n)$. Assume this is not the case. Then there must exist two operations $op[i], op'[j] \in O[1, n]$ such that

$$op[i] \prec op'[j] \quad \text{and} \quad t(op'[j], C') \Leftrightarrow \tau(op) < t(op[i], C')$$

Because $\mathcal{L}$ is recurrent it must be that $i \leq j \Leftrightarrow d(op, op')$. Furthermore since $C \overset{m}{\Leftrightarrow\!\!\rightarrow} \mathcal{L}(n)$:

$$\forall\, k \in [1, n \Leftrightarrow d(op, op')] \quad \Rightarrow \quad t(op[k], C) + \tau(op) \leq t(op'[k + d(op, op')], C)$$

For $k = j \Leftrightarrow d(op, op')$ the previous inequality implies that there are at least $j \Leftrightarrow d(op, op') \geq i$ instances of operations op executing on or before cycle $t(op'[j], C') \Leftrightarrow \tau(op)$ and therefore every instance of op which starts executing strictly after cycle $t(op'[j], C') \Leftrightarrow \tau(op)$ must have an index strictly greater than i. This is in contradiction with the fact $op[i]$ has iteration index i. Thus $C \overset{m}{\Leftrightarrow\!\!\rightarrow} \mathcal{L}(n)$ implies that $C' \overset{m}{\Leftrightarrow\!\!\rightarrow} \mathcal{L}(n)$ and the proof is complete. □

4.5 Consistent and Periodic Schedules

An m-admissible set of schedules for some cyclic system can contain very disparate elements. In practice, however, no man made scheduler is likely to generate completely irregular schedules. This section introduces a family of schedules, called consistent, whose purpose is to approximate the behavior of a dynamic scheduler. Informally a set of schedules $\mathcal{C}$ is consistent if for every $n \geq 1$, $\mathcal{C}(n+1)$ is identical to $\mathcal{C}(n)$ when operations in iteration $n+1$ are disregarded.

Definition 4.5.1 *Let $\mathcal{L}$ be a cyclic system with core operation set O and let $\mathcal{C}$ be a set of schedules. $\mathcal{C}$ is said to be consistent for $\mathcal{L}$ if and only if $\mathcal{C} \overset{m}{\mapsto\!\!\!\Leftrightarrow\!\!\!\rightarrow} \mathcal{L}$ and:*

$$\forall\, n_1, n_2 \geq 1 \quad \forall\, op[i] \in O[1, \min(n_1, n_2)] \qquad t(op[i], \mathcal{C}(n_1)) = t(op[i], \mathcal{C}(n_2))$$

Thus a set of schedules $\mathcal{C}$ is consistent if and only if the cycle in which an operation $op[i]$ of $\mathcal{L}$ starts executing, is the same in all of $\mathcal{C}$'s schedules containing $op[i]$. The following theorem characterizes consistent schedules.

Theorem 4.5.1 *Let $\mathcal{L}$ be a cyclic system with core operation set O and $\mathcal{C} \overset{m}{\mapsto\!\!\!\Leftrightarrow\!\!\!\rightarrow} \mathcal{L}$ a set of schedules. For $m < \infty$, $\mathcal{C}$ is consistent if and only if there exists a unique infinite length m-schedule C^∞ such that:*

$$\forall\, op[i] \in O[1, \infty] \quad \forall\, n \geq i \qquad t(op[i], \mathcal{C}(n)) = t(op[i], C^\infty)$$

Proof: If an infinite schedule C^∞ verifying the above claim exists, $\mathcal{C}$ is per force consistent.

Conversely suppose that $\mathcal{C}$ is consistent and let $C^\infty = (I_1, \cdots)$ be the infinite schedule where

$$\forall\, k \geq 1 \quad I_k = \{op[i] \,|\, 0 \leq k \Leftrightarrow t(op[i], \mathcal{C}(i)) < \tau(op)\}$$

Because $\mathcal{C}$ is consistent $|I_k| \leq m$, furthermore it is clear that for all $op[i] \in O[1, \infty]$ and $n \geq i$, $t(op[i], \mathcal{C}(n)) = t(op[i], C^\infty)$. The uniqueness of C^∞ stems from this last equality.
□

The previous theorem shows that for $m < \infty$ any consistent set of schedules for a cyclic system $\mathcal{L}$ can be viewed as an infinite length schedule C^∞. Conversely let C^∞ be an infinite length schedule which is m-admissible for $\mathcal{L}$. It is easy to extract from C^∞ a consistent set of schedules for $\mathcal{L}$. This shows that for $m < \infty$ consistent sets of schedules and infinite length schedules portray the same concept and will, in the succeeding sections, be regarded as being the same.

In addition to consistency, another desirable property that a set of *statically* generated schedules should possess is regularity, that is the ability to generate all schedules in the set from a fix core schedule. A more stringent condition, periodicity, is therefore grafted onto consistency to model schedules generated by a static schedulers.

Definition 4.5.2 *Let O be some finite set of operations and $\mathcal{C}$ an infinite length schedule containing only operations in $O[1,\infty]$. $\mathcal{C}$ is said to be periodic if and only if there exist two integers $l \geq 1$ and $\lambda \geq 1$ such that:*

$$\forall\, op \in O \quad \forall\, i \geq 0 \qquad t(op[1+i], \mathcal{C}) = t(op[1 + i \bmod l], \mathcal{C}) + \lambda \cdot \left\lfloor \frac{i}{l} \right\rfloor$$

The numbers l and λ are respectively called the unfolding and initiation interval of $\mathcal{C}$. Let $\mathcal{L}$ be some cyclic system with core operation set O. If $\mathcal{C}$ is m-admissible for $\mathcal{L}$ one says that $\mathcal{C}$ is (l,λ)-periodic for $\mathcal{L}$.

The unfolding of a periodic schedule is closely related to the notion of unrolling of a cyclic system.

Property 4.5.1 *Let $\mathcal{L}$ be some cyclic system and $\mathcal{C}$ an infinite length schedule. $\mathcal{C}$ is (l,λ)-periodic for $\mathcal{L}$ if and only if $\mathcal{C}$ is $(1,\lambda)$-periodic for $u^l(\mathcal{L})$.*

Proof: $\mathcal{C}$ is $(1,\lambda)$-periodic for $u^l(\mathcal{L})$ if and only if

$$\forall\, op[a] \in O^l \quad \forall\, k \geq 0 \qquad t(op[a][1+k], \mathcal{C}) = t(op[a][1], \mathcal{C}) + \lambda \cdot k$$

replacing $op[a][k]$ by $op[a + (k \Leftrightarrow 1) \cdot l]$ yields

$$\forall\, op \in O \quad \forall\, a \in [1,l] \quad \forall\, k \geq 0 \qquad t(op[a + k \cdot l], \mathcal{C}) = t(op[a], \mathcal{C}) + \lambda \cdot k$$

which shows that $\mathcal{C}$ is (l,λ)-periodic if and only if $\mathcal{C}$ is $(1,\lambda)$-periodic for $u^l(\mathcal{L})$ for $\mathcal{L}$. $\square$

A (l, λ)-periodic schedule $\mathcal{C}$ for $\mathcal{L}$, is perfectly determined by λ and the starting cycles of operations in $O[1, l]$. It should therefore be possible to characterize the admissibility of $\mathcal{C}$ for $\mathcal{L}$ solely in terms of λ, l and $t(op[i], \mathcal{C})$ for each $op[i] \in O[1, l]$. Indeed Lam's software pipelining algorithm [38] is based on the following result:

Theorem 4.5.2 *Let $\mathcal{L}$ be a cyclic system and $\mathcal{C}$ an infinite periodic m-schedule. $\mathcal{C}$ is (l, λ)-periodic for $\mathcal{L}$ if and only if:*

$$\forall\, op[i], op'[j] \in O[1, l] \quad t(op'[j], \mathcal{C}) - t(op[i], \mathcal{C}) \geq \tau(op) - d^l(op[i], op'[j]) \cdot \lambda$$

Note that for $d^l(op[i], op'[j]) = \infty$ the constraint reads $t(op'[j], \mathcal{C}) - t(op[i], \mathcal{C}) \geq \tau(op) - \infty \cdot \lambda$ which is by definition always true.

Proof: Suppose that $\mathcal{C}$ is (l, λ)-periodic for $\mathcal{L}$ but

$$\exists\, op[i], op'[j] \in O[1, l] \quad t(op'[j], \mathcal{C}) + d^l(op[i], op'[j]) \cdot \lambda < t(op[i], \mathcal{C}) + \tau(op)$$

For convenience denote $d^l(op[i], op'[j])$ by d^l_{ij}. Because of the definition of d^l_{ij}

$$\exists\, k \geq 0 \quad op[i + k \cdot l] \prec op'[j + k \cdot l + d^l_{ij} \cdot l]$$

This implies that $t(op[i+k \cdot l], \mathcal{C}) + \tau(op) \leq t(op'[j + k \cdot l + d^l_{ij} \cdot l], \mathcal{C})$. Because $\mathcal{C}$ is periodic one can write

$$t(op[i], \mathcal{C}) + k \cdot \lambda + \tau(op) \leq t(op'[j], \mathcal{C}) + (k + d^l_{ij}) \cdot \lambda$$

which is in contradiction with $t(op'[j], \mathcal{C}) + d^l_{ij} \cdot \lambda < t(op[i], \mathcal{C}) + \tau(op)$.

Conversely let $\mathcal{C}$ be a periodic schedule with unfolding l and initiation interval λ. Then for all $op[i], op'[j] \in O[1, l]$ and positive integers k, k', where $k \leq k'$

$$t(op[i], \mathcal{C}) + \tau(op) \leq t(op'[j], \mathcal{C}) + d^l_{ij} \cdot \lambda \Rightarrow t(op[i+k \cdot l], \mathcal{C}) + \tau(op) \leq t(op'[j + (k' + d^l_{ij}) \cdot l], \mathcal{C})$$

Because the positive integer n_0 of property 4.3.1 can be assumed to be 1 without any loss in generality

$$\forall\, q, q' \geq 0 \quad op[i + q \cdot l] \prec op[j + q' \cdot l] \;\Rightarrow\; q + d^l_{ij} \leq q'$$

and consequently $\mathcal{C}$ is admissible for $\mathcal{L}$. □

Periodicity captures the intuitive notion of schedules which are expressible in the form of loops. Before proving this result one needs to introduce concatenation, a simple operation that can be applied to m-schedules.

Definition 4.5.3 *Let $C = (I_1, \cdots, I_{|C|})$ and $C' = (I'_1, \cdots)$ be two m-schedules such that C is of finite length. The concatenation of C and C' is the m-schedule $C \odot C'$ whose first $|C|$ instructions coincide with those of C and the remaining instructions coincide with those of C': $C \odot C' = (I_1, \cdots, I_{|C|}, I'_1, \cdots)$. Schedule C is said to be a prefix of $C \odot C'$.*

The operation $\odot$ is associative and parentheses can therefore be omitted when concatenating more than two schedules. Let $(B_i)_{i \geq 1}$ be an infinite sequence of finite length schedules. Then $\bigodot_{i \geq 1} B_i$ denotes the unique infinite length schedule B^∞ such that for any $k \geq 1$, $(\bigodot_{i \geq 1}^{k} B_i)$ is a prefix of B^∞.

After the following technical definition the theorem characterizing periodic schedules is given.

Definition 4.5.4 *Let I be some instruction and k some integer. $I(k)$ denotes the instruction in which every operation $op[j]$ in I, where j can be negative, has been deleted if $j + k \leq 1$ or replaced by $op[j + k \Leftrightarrow 1]$ if $j + k > 1$. Let S be some schedule. $S(k)$ denotes the schedule where each instruction I in S has been replaced by $I(k)$.*

Theorem 4.5.3 *Let $\mathcal{L}$ be some cyclic system and $\mathcal{C} \overset{m}{\leftrightarrow} \mathcal{L}$ an infinite length schedule. Then $\mathcal{C}$ is (l, λ)-periodic for $\mathcal{L}$ if and only if there exists an m-schedule B, called the body of $\mathcal{C}$, such that $|B| = l \cdot \lambda$ and*

$$\mathcal{C} = \bigodot_{j=0}^{\infty} B(1 + l \cdot j)$$

Proof: Because of property 4.5.1 one can assume, without any loss in generality, that $l = 1$. Let $\mathcal{L} = (O[1, \infty], \tau, \prec)$. Suppose there exists a m-schedule B such that $B^\infty = \bigodot_{j=1}^{\infty} B(j) \overset{m}{\leftrightarrow} \mathcal{L}$. Then for every operation $op \in O$, there exists some integer $k_{op} \geq 1$ such that $op[1]$ starts executing in $B(k_{op}$. Let t_{op} be the cycle in $B(k_{op})$ where $op[k_{op}]$ starts executing. Then

$$\forall\, i \geq 0 \quad t(op[1 + i], B^\infty) = (t_{op} + k_{op} \cdot |B|) + i \cdot |B|$$

and consequently $t(op[1+i], B^\infty) = t(op[1], B^\infty) + i \cdot |B|$. Thus B^∞ is $(1, |B|)$-periodic.

Conversely let $\mathcal{C}$ be a $(1,\lambda)$-periodic schedule for $\mathcal{L}$. Let $B = (I_1, \cdots, I_\lambda)$ where:

$$a \in [1,\lambda] \quad I_a = \left\{ op\left[1 \Leftrightarrow \left\lfloor \frac{t(op[1],\mathcal{C})}{\lambda} \right\rfloor\right] \;\middle|\; 0 \le a \Leftrightarrow (1 + t(op[1],\mathcal{C}) \bmod \lambda) < \tau(op) \right\}$$

Because for all $t \ge 0$, $I_a(t) = \{op[k] \in O[1,\infty] \,|\, t(op[k],\mathcal{C}) \le a + t\cdot\lambda < t(op[k],\mathcal{C}) + \tau(op)\}$ it is clear that $|I_a| \le m$ and that

$$\mathcal{C} = (I_1(1), \ldots, I_\lambda(1), I_1(2), \ldots, I_\lambda(2), \cdots) = \bigodot_{j=1}^{\infty} B(j)$$

□

4.6 Asymptotic Performance

Given a cyclic system $\mathcal{L}$ and a set of schedules $\mathcal{C} \overset{m}{\Leftrightarrow\!\!\rightarrow} \mathcal{L}$, the duration of a particular n-instance of $\mathcal{L}$ in $\mathcal{C}$ is given by $|\mathcal{C}(n)|$. It is quite possible that the set of schedules $\mathcal{C}$ yields optimum performance for some instances of $\mathcal{L}$ but not for all. If $\mathcal{C}(n)$ is optimum for every $n \ge 1$ then one says that $\mathcal{C}$ is m-optimum for $\mathcal{L}$. In general optimum sets of schedules do not possess the consistency and regularity introduced in the previous section.

Theorem 4.6.1 *There exists a cyclic system $\mathcal{L}$ such that for all unrolling $x \ge 1$, number of processors $1 < m < \infty$ and consistent schedule $\mathcal{C}^x \overset{m}{\Leftrightarrow\!\!\rightarrow} u^x(\mathcal{L})$ there are infinitely many n-instances of $u^x(\mathcal{L})$ for which $\mathcal{C}^x(n)$ is not m-optimum.*

Proof: Let $\mathcal{L} = (O[1,\infty], \tau, \prec)$ where $O = \{op\}$, $\tau(op) = 1$ and $\prec$ is such that

$$\forall\, i \ge 0 \quad \forall\, j \in [1, 2^i \Leftrightarrow 1] \quad op[2^i + j] \prec op[2^i + j + 1]$$

These dependences are portrayed in figure 4.2.

For $m \ge 2$ and $x \ge 1$ let $\mathcal{C}^x_{\text{opt}}$ denote the m-optimum set of schedules for $u^x(\mathcal{L})$. Clearly

$$\forall\, i \ge 0 \quad |\mathcal{C}^1_{\text{opt}}(2^{i+1})| = 2^i$$

$u^1(\mathcal{L})(2^{i+1})$ contains a dependence chain of 2^i cycles, thus $|\mathcal{C}^1_{\text{opt}}(2^{i+1})| \ge 2^i$. Furthermore $|\mathcal{C}^1_{\text{opt}}(2^{i+1})| \le 2^i$ for one can schedule $op[1]$ through $op[2^i]$ sequentially in the first 2^i

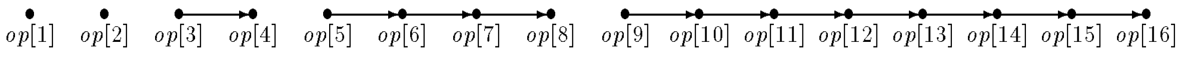


Figure 4.2: Dependences of some cyclic system.

cycles and operations $op[2^i + 1]$ through $op[2^{i+1}]$ also sequentially in the first 2^i cycles. Consequently

$$\forall\, i \geq 0 \qquad \left|\mathcal{C}^x_{\text{opt}}\left(\left\lfloor\frac{2^{i+1}}{x}\right\rfloor\right)\right| \leq 2^i$$

Let $\mathcal{C}^x$ be a consistent m-schedule for $u^x(\mathcal{L})$ and 2^k the smallest power of two such that all operations scheduled in the first x cycles of $\mathcal{C}^x$ have iteration index less than 2^k. Then for $i \geq k$

$$\left|\mathcal{C}^x\left(\left\lfloor\frac{2^{i+1}}{x}\right\rfloor\right)\right| \geq x + \left\lfloor\frac{2^i}{x}\right\rfloor \cdot x > 2^i \geq \left|\mathcal{C}^x_{\text{opt}}\left(\left\lfloor\frac{2^{i+1}}{x}\right\rfloor\right)\right|$$

As the above inequality holds for any $i \geq k$ there are infinitely many n-instances of $u^x(\mathcal{L})$ for which $\mathcal{C}^x(n)$ is not m-optimum. $\square$

Despite this negative result one can show the existence of consistent schedules which are only a constant factor away from the m-optimum.

Theorem 4.6.2 *Let $\mathcal{L}$ be some cyclic system and $\mathcal{C}_{\text{opt}}$ an m-optimum set of schedules for $\mathcal{L}$. Then there exists a consistent schedule $\mathcal{C}$ for $\mathcal{L}$ such that:*

$$\forall\, n \geq 1 \qquad \frac{|\mathcal{C}(n)|}{|\mathcal{C}_{\text{opt}}(n)|} \leq 2 \Leftrightarrow \frac{1}{m}$$

Proof: Let $\mathcal{L} = (O[1,\infty], \tau, \prec)$ and $\mathcal{C}$ a consistent schedule for $\mathcal{L}$ produced by a list scheduling algorithm where for all $op, op' \in O$ and $i \geq 1$, $op[i]$ is given priority over $op'[i+1]$. More precisely let I_k be the k-th instruction of $\mathcal{C}$, for $k \geq 1$. Then I_{k+1} contains the operations scheduled in I_k which have not completed their execution in I_k and as many unscheduled operations as resource constraints permit. These unscheduled operations are randomly selected except for the fact that their iteration indices are chosen to be as little as possible. Because for all $n \geq 1$ operations in $O[1,n]$ are given priority over operations in $O[n+1,\infty]$, $\mathcal{C}(n)$ is a list schedule, that is if for some $k \geq 1$ $|I_k| < m$ then every operation scheduled after I_k must depend on some operation scheduled in I_k. Thus Graham's theorem that every list schedule is within $2 \Leftrightarrow 1/m$ of an m-optimum holds [10]. □

The previous result does not generalize to periodic schedules. Intuitively this is because m-optimum sets of schedules can be very disparate whereas periodic schedules are not. What more it can be shown that cyclic systems exist for which any periodic schedule is almost a factor m away from the optimum. Thus in the presence of arbitrary dependences dynamic schedules can, in theory, severely outperform static ones.

Theorem 4.6.3 *There exists a cyclic system $\mathcal{L}$ with m-optimum set of schedules $\mathcal{C}_{\text{opt}}$ such that for all $1 < m$ and periodic schedule $\mathcal{C} \overset{m}{\Leftrightarrow\!\!\rightarrow} \mathcal{L}$:*

$$\lim_{n\to\infty} \frac{|\mathcal{C}(n)|}{|\mathcal{C}_{\text{opt}}(n)|} = m$$

Proof: Let $(v_k)_{k\geq 1}$ be the sequence of numbers where $v_1 = 1$ and for all $i \geq 1$:

$$\begin{array}{rcl} v_{2i} & = & 1 + 2^i + v_{2i-1} \\ v_{2i+1} & = & i + v_{2i} \end{array}$$

The cyclic system $\mathcal{L}$ has core operation set $\{op\}$, duration function τ, where $\tau(op) = 1$ and dependence relation $\prec$ where

$$\forall\, j \in [v_{2i}, v_{2i+1} \Leftrightarrow 1] \qquad op[j] \prec op[j+1]$$

The dependences are portrayed in figure 4.3. The dependence distances are as follows:

$$\forall\, x \geq 1 \quad \forall\, i,j \in [1,x] \qquad d^x(op[i], op[j]) = \begin{cases} 0 & \text{if } i < j \\ 1 & \text{otherwise} \end{cases}$$

Because of theorem 4.5.2 the body of every periodic schedule $\mathcal{C}$ for $\mathcal{L}$ must be sequential and thus for all $n \geq 1$ $|\mathcal{C}(n)| = n$. Let $\mathcal{C}_{\text{opt}}$ be an m-optimum set of schedules for $\mathcal{L}$. Then

$$|\mathcal{C}_{\text{opt}}(n)| \leq \begin{cases} \left\lceil \frac{2^{i+1}}{m} \right\rceil + i \cdot \frac{i+1}{2} & \text{if } v_{2i} \leq n \leq v_{2i+1} \text{ for some } i \\[2ex] \left\lceil \frac{2^{i+1}}{m} \right\rceil + i \cdot \frac{i-1}{2} & \text{if } v_{2i-1} < n < v_{2i} \text{ for some } i \end{cases}$$

Thus $|\mathcal{C}_{\text{opt}}| \leq n/m + o(n)$, where $o(n)$ designates a function such that $\lim_{n\to\infty} o(n)/n = 0$. □

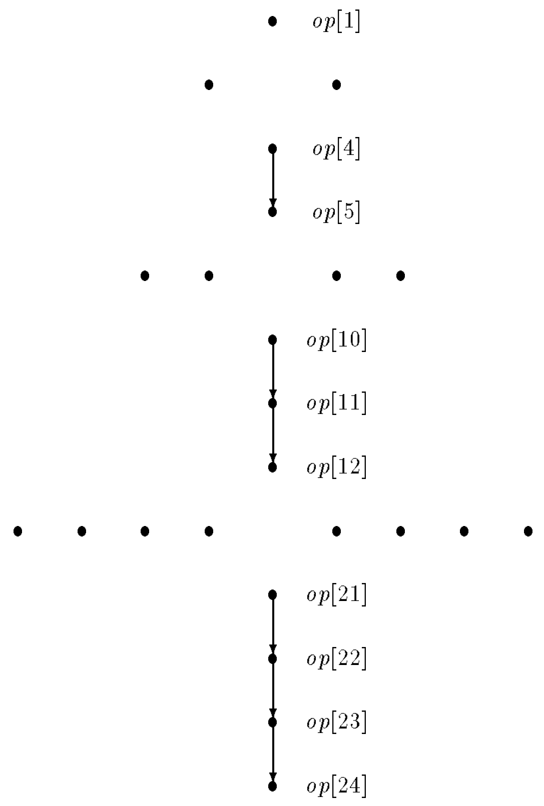


Figure 4.3: Dependences of some cyclic system. For clarity certain operations are not labeled.

The previous negative results are based on the fact that dependences can be quite irregular. It is therefore natural to investigate the performance of periodic schedules when dependences are recurrent. The notion of asymptotic performance will play a key role in this investigation.

Definition 4.6.1 *Let $\mathcal{L}$ be a cyclic system and $\mathcal{C} \overset{m}{\Leftrightarrow\!\!\!\rightarrow} \mathcal{L}$ a set of schedules. The asymptotic performance of $\mathcal{C}$ for $\mathcal{L}$, denoted $\Delta(\mathcal{C},\mathcal{L})$ is defined as $\lim_{n\to\infty} |\mathcal{C}(n)|/n$ if such limit exists. If there exists no $\mathcal{C}' \overset{m}{\Leftrightarrow\!\!\!\rightarrow} \mathcal{L}$ with smaller asymptotic performance, $\mathcal{C}$ is said to be an asymptotic m-optimum for $\mathcal{L}$.*

For all practical purposes asymptotic performance characterizes m-optimum set of schedules when dependences are recurrent.

Theorem 4.6.4 *Let $\mathcal{L}$ be a cyclic system with recurrent dependences. Then every m-optimum set of schedules for $\mathcal{L}$ is also asymptotically m-optimum for $\mathcal{L}$. Conversely let $\mathcal{C}$ be an asymptotically m-optimum set of schedules for $\mathcal{L}$ then for every arbitrarily small positive real ϵ there exists a constant n_ϵ such that*

$$\forall\, n \geq n_\epsilon \quad \frac{|\mathcal{C}(n)|}{|\mathcal{C}_{\mathrm{opt}}(n)|} \leq 1+\epsilon$$

Proof: Let $\mathcal{C}_{\mathrm{opt}}$ be an m-optimum set of schedules for $\mathcal{L}$. Because dependences are recurrent

$$\forall\, n,k \geq 1 \quad |C_{\mathrm{opt}}(n+k)| \leq |C_{\mathrm{opt}}(n)| + |C_{\mathrm{opt}}(k)|$$

for if one poses $C_1 = C_{\mathrm{opt}}(n)$ and $C_2 = C_{\mathrm{opt}}(k)$, then schedule $C_1 \odot C_2(n+1)$ is an m-admissible schedule for $\mathcal{L}(n+k)$. This inequality implies that the set $A_{\mathrm{opt}} = \{|\mathcal{C}_{\mathrm{opt}}(n)|/n \,|\, n \geq 1\}$ has a single accumulation point. A real number a is an accumulation point of A_{opt} if and only if

$$\forall\, \epsilon > 0 \quad \forall\, n \geq 0 \quad \exists\, n' \geq n \quad \left| \frac{|\mathcal{C}_{\mathrm{opt}}(n')|}{n'} \Leftrightarrow a \right| < \epsilon$$

Because A_{opt} is both lower and upper bounded, A_{opt} has at least one accumulation point. Suppose it has two: a_1 and $a_2 > a_1$. Then for all ϵ there exists an arbitrarily big integer n and an integer $n' > n$ arbitrarily bigger than n such that:

$$|\mathcal{C}_{\mathrm{opt}}(n)| \leq (a_1+\epsilon)\cdot n \quad \text{and} \quad (a_2 \Leftrightarrow \epsilon)\cdot n' \leq |\mathcal{C}_{\mathrm{opt}}(n')|$$

Let $n' = k \cdot n + c$, where $0 \leq c < n$. Then

$$\begin{aligned}(a_2 \Leftrightarrow \epsilon) \cdot k \cdot n &\leq |\mathcal{C}_{\text{opt}}(n')| \leq k \cdot |\mathcal{C}_{\text{opt}}(n)| + |\mathcal{C}_{\text{opt}}(c)| \\ &\leq (k+1) \cdot n \cdot (a_1 + \epsilon)\end{aligned}$$

this yields

$$a_2 \Leftrightarrow a_1 \leq \frac{a_1}{k} + \epsilon \cdot \frac{2 \cdot k + 1}{k}$$

As k can be chosen arbitrarily big and ϵ arbitrarily small one obtains a contradiction. Thus A_{opt} has a single accumulation point a. Furthermore

$$\forall\, n \geq 1 \qquad a \leq \frac{|\mathcal{C}_{\text{opt}}(n)|}{n}$$

for otherwise there would exists an $n \geq 1$ such that the set $\{\, |\mathcal{C}_{\text{opt}}(k \cdot n)|/k \cdot n \mid k \geq 1\} \subseteq A_{\text{opt}}$ would have an accumulation point different than a since

$$\frac{|\mathcal{C}_{\text{opt}}(k \cdot n)|}{k \cdot n} \leq \frac{|\mathcal{C}_{\text{opt}}(n)|}{n} < a$$

The final step in proving that $\lim_{n \to \infty} |\mathcal{C}_{\text{opt}}(n)|/n$ exists is to show

$$\forall\, \epsilon > 0 \quad \exists\, n_\epsilon \geq 1 \quad \forall\, n' > n_\epsilon \qquad \left| \frac{|\mathcal{C}_{\text{opt}}(n)|}{n} \Leftrightarrow a \right| < \epsilon$$

For all $k, n \geq 1$ and $c \in [0, n \Leftrightarrow 1]$ one can write

$$\begin{aligned}a \leq \frac{|\mathcal{C}_{\text{opt}}(k \cdot n + c)|}{k \cdot n + c} &\leq \frac{|\mathcal{C}_{\text{opt}}(n)|}{n + \frac{c}{k}} + \frac{|\mathcal{C}_{\text{opt}}(c)|}{k \cdot n + c} \\ &\leq \frac{|\mathcal{C}_{\text{opt}}(n)|}{n} + \frac{|\mathcal{C}_{\text{opt}}(n)|}{k}\end{aligned}$$

Because a is the accumulation point n can be chosen such that

$$\frac{|\mathcal{C}_{\text{opt}}(n)|}{n} < a + \frac{\epsilon}{2}$$

Since k is independent of n, there exists a k_ϵ such that

$$\frac{|\mathcal{C}_{\text{opt}}(n)|}{k} < \frac{\epsilon}{2}$$

Thus

$$a \leq \frac{|\mathcal{C}_{\text{opt}}(k \cdot n + c)|}{k \cdot n + c} \leq a + \epsilon$$

Because every number $n' \geq n$ can be written in the form $k \cdot n + c$, by letting $n_\epsilon = (k_\epsilon + 1) \cdot n$ the desired result is obtained. Consequently $\Delta(\mathcal{C}_{\text{opt}}, \mathcal{L}) = \lim_{n \to \infty} |\mathcal{C}_{\text{opt}}(n)|/n$ is defined and furthermore such asymptotic performance must be optimum for an m processor machine.

Conversely let $\mathcal{C}$ be an asymptotically m-optimum set of schedules for $\mathcal{L}$ and ϵ' an arbitrarily small positive real. Then there exists an $n_{\epsilon'} \geq 1$ such that for all $n \geq n_{\epsilon'}$

$$\left| \frac{|\mathcal{C}_{\text{opt}}(n)|}{n} \Leftrightarrow \frac{|\mathcal{C}(n)|}{n} \right| \leq \left| \frac{|\mathcal{C}_{\text{opt}}(n)|}{n} \Leftrightarrow \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L}) \right| + \left| \frac{|\mathcal{C}(n)|}{n} \Leftrightarrow \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L}) \right| < 2 \cdot \epsilon'$$

Thus

$$\left| \frac{|\mathcal{C}(n)|}{|\mathcal{C}_{\text{opt}}(n)|} \Leftrightarrow 1 \right| < \frac{n}{|\mathcal{C}_{\text{opt}}(n)|} \cdot 2 \cdot \epsilon' \leq 2 \cdot m \cdot \epsilon'$$

by letting $\epsilon' < \epsilon/(2 \cdot m)$ one obtains the desired result. □

Because of its regularity, the asymptotic performance of a periodic schedule is always defined and can be expressed in terms of its unfolding and initiation interval.

Theorem 4.6.5 *Let $\mathcal{L}$ be a cyclic system and $\mathcal{C}$ a (l, λ)-periodic schedule for $\mathcal{L}$. Then*

$$\Delta(\mathcal{C}, \mathcal{L}) = \frac{\lambda}{l}$$

Proof: Because $\mathcal{C}$ is (l, λ)-periodic

$$\forall\, i \geq 0 \quad |\mathcal{C}(i+1)| = |\mathcal{C}(1 + i \bmod l)| + \lambda \cdot \left\lfloor \frac{i}{l} \right\rfloor$$

Thus

$$\frac{\lambda}{l} \cdot \left(\frac{i \Leftrightarrow i \bmod l}{i+1} \right) \leq \frac{|\mathcal{C}(i+1)|}{i+1} \leq \frac{\lambda}{l} \cdot \left(\frac{i}{i+1} \right) + \frac{|\mathcal{C}(1 + i \bmod l)|}{i+1}$$

As i tends toward infinity both the upper and the lower bounds of $|\mathcal{C}(i+1)|/(i+1)$ tend towards λ/l and this proves the claim. □

Theorem 4.6.3 showed that performance of periodic schedules can be poor if the cyclic system at hand has irregular dependences. The next result proves that for recurrent cyclic systems there exist periodic schedules which can come arbitrarily close to the asymptotic optimum. Hence when dependences are recurrent static schedules can be as good as dynamic ones.

Theorem 4.6.6 *Let $\mathcal{L}$ be a recurrent cyclic system and $\mathcal{C}_{\text{opt}}$ a set of m-optimum schedules for $\mathcal{L}$. Then for every arbitrarily small $\epsilon > 0$ there exists a periodic schedule $\mathcal{C}$ for $\mathcal{L}$ such that:*

$$\Delta(\mathcal{C}, \mathcal{L}) \Leftrightarrow \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L}) < \epsilon$$

Proof: Let ϵ be some positive real. Then there exists a positive l such that

$$\left| \frac{|\mathcal{C}_{\text{opt}}(l)|}{l} \Leftrightarrow \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L}) \right| < \epsilon$$

Let $B = \mathcal{C}_{\text{opt}}(l)$. Because dependences are recurrent $B^\infty = \bigodot_{j=0}^{\infty} B(1 + j \cdot l)$ is a periodic schedule and is m-admissible for $\mathcal{L}$. Furthermore theorem 4.6.5 has shown that the asymptotic performance of B^∞ is $|B|/l = |\mathcal{C}_{\text{opt}}(l)|/l$. $\square$

Because periodic schedules can yield almost asymptotically optimum performance for recurrent cyclic systems it is natural to wonder whether there exists a periodic schedule which achieves m-optimum asymptotic performance. This question can be answered affirmatively in the restricted case where the dependence graph of a recurrent cyclic system is strongly connected.

Theorem 4.6.7 *Let $\mathcal{L}$ be a recurrent cyclic system. If $\mathcal{L}$'s dependence graph is strongly connected for every positive m there exists an asymptotically m-optimum schedule which is periodic.*

Proof: Let $\mathcal{C}_{\text{opt}}$ an m-optimum set of schedules for $\mathcal{L}$, ϵ some positive real and l_ϵ the smallest positive integer such that

$$\Delta(\mathcal{C}_{\text{opt}}, \mathcal{L}) \leq \frac{|\mathcal{C}_{\text{opt}}(l_\epsilon)|}{l_\epsilon} \leq \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L}) + \epsilon$$

Let O be the core operation set of $\mathcal{L}$. Because of property 4.4.2 one can assume that for all ϵ

$$\forall\, op \in O \quad \forall\, i \in [1, l_\epsilon \Leftrightarrow 1] \quad t(op[i], \mathcal{C}_{\text{opt}}(l_\epsilon)) \leq t(op[i+1], \mathcal{C}_{\text{opt}}(l_\epsilon))$$

If l_ϵ is bounded for all ϵ then

$$\exists\, l \geq 1 \quad \frac{|\mathcal{C}_{\text{opt}}(l)|}{l} = \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L})$$

by letting $B = \mathcal{C}_{\text{opt}}(l)$, as in the proof of the previous theorem, $B^\infty = \bigodot_{j=0}^{\infty} B(1 + j \cdot l)$ is a periodic schedule for $\mathcal{L}$ with asymptotically m-optimum performance.

Let X be any consecutive block of $\mathcal{C}_{\text{opt}}(l_\epsilon)$'s instructions. If l_ϵ is unbounded, $|X|$ can be arbitrarily big. Theorem 4.4.1 shows the existence of a constant c such that if $|X| = c + 1$ then there exist $I_1, I_2 \in X$ such that every strong component in $\mathcal{L}'s$ dependence graph repeats an integral number of times between I_1 and I_2. Let H, Y, T be three schedules built of consecutive $\mathcal{C}_{\text{opt}}(l_\epsilon)$'s instructions as follows. H comprises the instructions up to but not including I_1, Y the instruction from I_1 included to I_2 excluded and T the remaining instructions: $\mathcal{C}_{\text{opt}}(l_\epsilon) = H \odot Y \odot T$. Because $\mathcal{L}$'s dependence graph is strongly connected Y and Y contains an integral number of strong components, $\mathcal{L}$'s dependence graph must be schedules k times, for some positive integer k, in Y. Thus the schedule $H \odot T(\Leftrightarrow k)$ is m-admissible for $\mathcal{L}(l_\epsilon \Leftrightarrow k)$. Since l_ϵ was chosen to be the smallest integer such that

$$\Delta(\mathcal{C}_{\text{opt}}, \mathcal{L}) \leq \frac{|\mathcal{C}_{\text{opt}}(l_\epsilon)|}{l_\epsilon} \leq \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L}) + \epsilon$$

one must have

$$\frac{|\mathcal{C}_{\text{opt}}(l_\epsilon)|}{l_\epsilon} \leq \frac{|H \odot T(\Leftrightarrow k)|}{l_\epsilon \Leftrightarrow k}$$

and consequently

$$\frac{|Y|}{k} \leq \frac{|H \odot Y \odot T|}{l_\epsilon}$$

As the length of Y is bounded by the constant c there are only finitely many possible Y. Thus there must exist an epsilon for which

$$\frac{|Y|}{k} \leq \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L})$$

Let y be the biggest of the iteration indices in Y of any operation all of whose incoming edges in $\mathcal{L}$'s dependence graph have non zero distance. Let B be the schedule equal to Y except for the iteration indices of the operations in Y which are decremented by $y \Leftrightarrow 1$. Because Y contains the dependence graph of $\mathcal{L}$ exactly k times and because dependences are recurrent, the infinite schedule $B^\infty = \bigodot_{j=0}^{\infty} B(1 + j \cdot l)$ is a periodic schedule m-admissible for $\mathcal{L}$. Its asymptotic performance is $|Y|/k \leq \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L})$, thus B^∞ must be asymptotically m-optimum for $\mathcal{L}$. $\Box$

In an attempt to characterize recurrent cyclic systems for which asymptotically optimum periodic schedules exist, a result characterizing such cyclic systems is presented. The general problem of whether there exists an asymptotically optimum periodic schedule for every cyclic system with recurrent dependences remains open.

Theorem 4.6.8 *Let $\mathcal{L}$ be a cyclic system with recurrent dependences. Then there exists an asymptotically m-optimum periodic schedule for $\mathcal{L}$ if and only if there exists an asymptotically m-optimum set of schedules for $\mathcal{L}$ where the maximum time delay between the execution of any two operations in a same iteration is bounded.*

Proof: For every periodic schedule the maximum delay between the execution of any two operations in a same iteration is clearly bounded. Thus if a periodic schedule is asymptotically m-optimum for $\mathcal{L}$ then there exists an asymptotically m-optimum set of schedules for $\mathcal{L}$ where the maximum time delay between the execution of any two operations in a same iteration is bounded.

Conversely let O be the core operation set of $\mathcal{L}$ and suppose the existence of an asymptotically m-optimum set of schedules $\mathcal{C}$ for $\mathcal{L}$ for which there exists a positive integer c such that

$$\forall op, op' \in O \quad \forall\, i \geq 1 \quad \forall\, n \geq i \qquad |t(op[i], \mathcal{C}(n)) \Leftrightarrow t(op'[i], \mathcal{C}(n))| < c$$

Due to property 4.4.2 it can be assumed that

$$\forall\, op \in O \quad \forall\, i \geq 1 \qquad t(op[i], \mathcal{C}) \leq t(op[i+1], \mathcal{C})$$

Let $\tau_{\max} = \max_{op \in O} \tau(op)$ and consider the recurrent cyclic system $\mathcal{L}_c$ which has same the operation set and duration function as $\mathcal{L}$ but whose dependence graph is obtained from $\mathcal{L}$'s by adding an edge with distance $m \cdot (c + \tau_{\max})$ between any two operations in O. Let $op, op' \in O$. Because for all $i \geq 1$ $op'[i]$ must start executing in $\mathcal{C}$ at least c cycles after $op[i]$ does, operation $op[i + m \cdot (c + \tau_{\max})]$ cannot be executed before $op'[i]$ is completed, thus the additional dependences added in $\mathcal{L}_c$ are respected in $\mathcal{C}$ and $\mathcal{C}$ is m-admissible for $\mathcal{L}_c$. Because $\mathcal{L}_c$ dependences are more stringent then those of $\mathcal{L}$, any m-admissible set of schedules for $\mathcal{L}_c$ is also m-admissible for $\mathcal{L}$. Thus $\mathcal{L}_c$ and $\mathcal{L}$ have same optimum asymptotic performance. Because $\mathcal{L}_c$'s dependence graph is strongly connected one can apply the previous theorem to $\mathcal{L}_c$ and obtain the desired result. $\Box$

4.7 Approximating Optimal Asymptotic Performance

The goal of this section is to generate, in polynomial time, a periodic schedule which is within a small constant factor of the asymptotic m-optimum of any cyclic system with recurrent dependences. The overall strategy adopted is to transform the dependence graph D of a cyclic system into an acyclic dependence graph D' with less dependence edges and then invoke an acyclic scheduling algorithm on D' to construct the body of the periodic schedule. The dependence graph D' is obtained by deleting edges from D. The difficult part when deleting edges is to shorten the dependence paths in D as much as possible while preserving semantic correctness of any scheduling algorithm operating on D'. The following result introduces the strategy for deleting edges and proves its correctness.

Theorem 4.7.1 *Let $\mathcal{L} = (O[1,\infty], \tau, \prec)$ be a recurrent cyclic system and $T = (O, \tau, \prec')$ an acyclic task system where the dependence graph D' associated with $\prec'$ is obtained from $\mathcal{L}$'s dependence graph D by deleting between one and $d(Q)$ edges from every simple cycle Q of D and deleting all the edges which are not part of any cycle of D. Let B an m-admissible schedule for T and for $op, op' \in O$ let $b(op, op') = 1$ if op' starts executing in B before op terminates and $b(op, op') = 0$ otherwise. Then there exists a function χ mapping O into $[\Leftrightarrow\infty, 1]$ such that the following indexing constraint is met*

$$\forall\, op, op' \in O \quad \chi(op) \Leftrightarrow \chi(op') \geq b(op, op') \Leftrightarrow d(op, op')$$

and χ is computable in a time linear in the size of D. Furthermore let B_χ be the schedule where every operation op in B is replaced by $op[\chi(op)]$. Then $B_\chi^\infty = \bigodot_{j=1}^{\infty} B_\chi(j)$ is a periodic schedule m-admissible for $\mathcal{L}$.

Proof: The existence of a function χ satisfying the indexing constraint is shown by a constructive argument. Theorem 4.3.3 has shown that for recurrent cyclic systems dependence distances can be viewed as dependence graphs. More precisely given two operations $op, op' \in O$ it was proved that $d(op, op') = d(Q)$, where Q is the shortest path in the weighted graph D from op to op'. This property will be employed throughout the proof.

Let E be the edge set of D and $E_b \subseteq E$ the back edges of some depth first search of D. For $op' \in O$ let $pred_b(op') = \{op \in O \mid (op, op') \in E \Leftrightarrow E_b\}$. The indexing function χ is defined as follows

$$\chi(op') = \begin{cases} 1 & \text{if } pred_b(op) = \emptyset \\ \min\limits_{op \in pred_b(op')} \chi(op) + d(op, op') \Leftrightarrow b(op, op') & \text{otherwise} \end{cases}$$

Because the subgraph $(O, E \Leftrightarrow E_b)$ of D is acyclic the function χ is well defined. The function χ can easily be computed in $O(|O|+|E|)$ time by performing a depth first search on D initiated at any one of its vertices to construct E_b, and then traversing the graph $(O, E \Leftrightarrow E_b)$ in topological order computing $\chi(op)$ when all operations in $pred_b(op)$ have been visited. The function χ may map an operation into an integer greater than 1. To ensure that χ maps O into $[\Leftrightarrow\infty, 1]$ let $\chi_M = \max_{op \in O} \chi(op)$. Then for all $op \in O$, $\chi(op)$ is decremented by $\chi_M \Leftrightarrow 1$.

It remains to prove that χ satisfies the indexing constraint. Let $op, op' \in O$. If no path exists in D from op to op' then $d(op, op') = \infty$ and the indexing constraint is true by definition. Otherwise let $Q = ((op, op_1), \cdots, (op_q, op'))$ be the shortest path from op to op' in D. Suppose that the indexing constraint is satisfied for any two operations connected by an edge in D. Then let $op_0 = op$ and $op_{q+1} = op'$ then

$$\begin{aligned} \chi(op) \Leftrightarrow \chi(op') &= \sum_{i=0}^{q} \chi(op_i) \Leftrightarrow \chi(op_{i+1}) \\ &\geq \underbrace{\sum_{i=0}^{q} b(op_i, op_{i+1})}_{\geq b(op, op')} \Leftrightarrow \underbrace{\sum_{i=0}^{q} d(op_i, op_{i+1})}_{\substack{= d(op, op') \\ \text{by theorem 4.3.3}}} \end{aligned}$$

Thus to show that the indexing constraint holds for any two operations, it suffices to prove that it holds for any two operations connected by an edge in D.

Let $e = (op, op')$ be an edge in D. If $e \in E \Leftrightarrow E_b$ then $\chi(op) \Leftrightarrow \chi(op') \geq b(op, op') \Leftrightarrow d(op, op')$ by definition.

If $e \in E_b$ then e is a back edge. Thus there exists at least one simple cycle Q containing e but no other back edges. Let

$$b(Q) = \sum_{(op_1, op_2) \in Q} b(op_1, op_2)$$

Because at most $d(Q)$ edges have been deleted from Q there are at most $d(Q)$ operations in Q that do not comply with Q's dependences in B. Thus

$$0 \geq b(Q) \Leftrightarrow d(Q)$$

Let $P = Q \Leftrightarrow e$ be the path obtained from Q by removing edge $e = (op, op')$. By purely algebraic manipulation one can write

$$\sum_{(op_1,op_2)\in Q} (\chi(op_1) \Leftrightarrow \chi(op_2)) = \chi(op) \Leftrightarrow \chi(op') + \sum_{(op_1,op_2)\in P} (\chi(op_1) \Leftrightarrow \chi(op_2)) = 0$$

thus

$$\begin{aligned}\chi(op) \Leftrightarrow \chi(op') &\geq b(Q) \Leftrightarrow d(Q) \Leftrightarrow \sum_{(op_1,op_2)\in P} (\chi(op_1) \Leftrightarrow \chi(op_2)) \\ &\geq \underbrace{\left(b(Q) \Leftrightarrow \sum_{(op_1,op_2)\in P} b(op_1, op_2) \right)}_{=b(op,op')} \Leftrightarrow \underbrace{\left(d(Q) \Leftrightarrow \sum_{(op_1,op_2)\in P} d(op_1, op_2) \right)}_{=d(op,op')}\end{aligned}$$

Thus the function χ satisfies the indexing constraint for all $op, op' \in O$.

For the second part of the claim let $op, op' \in O$. If $t(op, B) + \tau(op) \leq t(op', B)$ then $b(op, op') = 0$ by definition and

$$t(op'[1], B_\chi^\infty) \Leftrightarrow t(op[1], B_\chi^\infty) \geq \tau(op) + |B| \cdot \underbrace{(\chi(op) \Leftrightarrow \chi(op'))}_{\geq -d(op,op')}$$

If $t(op, B) + \tau(op) > t(op', B)$ then $b(op, op') = 1$ and

$$t(op'[1], B_\chi^\infty) \Leftrightarrow t(op[1], B_\chi^\infty) \geq (\tau(op) \Leftrightarrow |B|) + |B| \cdot \underbrace{(\chi(op) \Leftrightarrow \chi(op'))}_{\geq 1 - d(op,op')}$$

In both cases since $|B| = |B_\chi|$

$$t(op'[1], B_\chi^\infty) \Leftrightarrow t(op[1], B_\chi^\infty) \geq \tau(op) \Leftrightarrow |B_\chi| \cdot d(op, op')$$

which by virtue of theorem 4.5.2 shows that B_χ^∞ satisfies $\mathcal{L}$'s dependence constraints. □

The previous result shows that if $\mathcal{L}$'s dependence graph D is acyclic then D' is edgeless. Thus any list schedule B generated from D' will guarantee good asymptotic performance for B_χ^∞. When D contains a cycle Q comprising edges $(op_1, op_2), \ldots, (op_q, op_1)$, theorem 4.3.3 shows that

$$\forall\ i \geq 1 \quad op_1[i] \prec op_2[i + d(op_1, op_2)] \prec \cdots \prec op_1[i + d(Q)]$$

If for a path P comprising edges $(op'_1, op'_2), \ldots, (op'_p, op'_{p+1})$ one denotes

$$\tau(P) = \sum_{j=1}^{p} \tau(op'_j)$$

then the above set of precedence relations shows that at most $d(Q)$ iterations can be executed every $\tau(Q)$ cycles. Let K_D denote the critical cycle of D, that is the cycle in D such that

$$\frac{\tau(K_D)}{d(K_D)} = \max_{\substack{Q \text{ cycle} \\ \text{of } D}} \frac{\tau(Q)}{d(Q)}$$

Let ρ_D denote the above critical ratio. Then for every m-optimum set of schedules $\mathcal{C}_{\text{opt}}$ for $\mathcal{L}$

$$\rho_D \leq \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L})$$

Let $\tau_{\max} = \max_{op \in O} \tau(op)$ and $K_{D'}$ the critical path of D', that is the path for which $\tau(K_{D'})$ is maximum. Because the length of schedule B is lower bounded by $\tau(K_{D'})$ good asymptotic performance cannot be guaranteed if $\tau(K_{D'})$ is much bigger than ρ_D. Fortunately it is possible to devise an algorithm that deletes edges from D in polynomial time and guarantees that

$$\tau(K_{D'}) < \lceil \rho_D \rceil$$

The algorithm is given below.

Algorithm 4.7.1

Input: A recurrent cyclic system $\mathcal{L} = (O[1,\infty], \tau, \prec)$ with dependence graph $D = (O, E)$.

Output: An acyclic task system $T = (O, \tau, \prec')$ with dependence graph D' such that D' is obtained from D by deleting all the edges which are not part of any cycle in D and by deleting between one and $d(Q)$ edges from every simple cycle Q of D. Furthermore let ρ_D be the critical ratio of D and $K_{D'}$ the critical path of D' then

$$\tau(K_{D'}) < \lceil \rho_D \rceil$$

Method: Find the strong components of D in $O(|O| + |E|)$ by employing any of the standard algorithms [56,54]. Delete the edges which are not part of any strong component. For each strong component $G = (O_G, E_G)$ of D do the following:

1. Compute the critical cycle K_G of G and the critical ratio $\rho_G = \tau(K_G)/d(K_G)$. To find the critical cycle proceed as follows. Replace each operation $op \in O_G$ by a chain of $\tau(op)$ operations $op_1, \ldots, op_{\tau(op)}$ each of duration 1. The dependence distance of each edge in the chain is set to zero. Apply Karp's minimum cycle mean algorithm [31] on the transformed graph. This step takes $O(\tau_{\max} \cdot |O_G| \cdot |E_G| + \tau_{\max}^2 \cdot |O_G|)$ time.

2. For each edge $e = (op, op') \in E_G$ set $w(e) = \tau(op) - \lceil \rho_G \rceil \cdot d(e)$. Select any operation op_s in O_G and compute the longest path with respect to w from op_s to every operation $op \in O_G$. Denote such longest path $\delta(op)$. Because K_G is the critical cycle all cycles in the graph G weighted with $-w$ are non-negative. Therefore one can invoke any of the shortest path algorithms, such as Ford's [40], on G weighted with $-w$. This step takes $O(|O_G| \cdot |E_G|)$ time.

3. An edge $e = (op, op') \in E_G$ is deleted from G if and only if
$$\delta(op') \bmod \lceil \rho_G \rceil < \delta(op) \bmod \lceil \rho_G \rceil + \tau(op)$$
This step takes $O(|E_G|)$ time.

The overall algorithm requires $O(|O| \cdot |E|)$ time if $\tau_{\max}$ is a fixed constant.

The next result proves the correctness of the above algorithm.

Theorem 4.7.2 *The dependence graph D' generated by algorithm 4.7.1 is such that every cycle Q of D gets deleted between one and $d(Q)$ edges. Furthermore let K_D be the critical cycle of D and $K_{D'}$ the critical path of D' then*
$$\tau(K_{D'}) < \left\lceil \frac{\tau(K_D)}{d(K_D)} \right\rceil$$

Proof: It suffices to show that the above claim holds in the case D has a single strongly connected component G. Let G, K_G, δ and $\lceil \rho_G \rceil$ as defined in algorithm 4.7.1 and let $Q = (e_1, \cdots, e_q)$ where $e_i = (op_i, op_{1+i \bmod q})$, for $i \geq 1$, be a cycle in G. Suppose that no edge is deleted from Q. Then
$$\forall\, i \in [1, q] \quad \delta(op_i) \bmod \lceil \rho_G \rceil + \tau(op_i) \leq \delta(op_{1+i \bmod q}) \bmod \lceil \rho_G \rceil$$

Thus $\delta(op_1) \bmod \lceil \rho_G \rceil + \tau(Q) \leq \delta(op_1) \bmod \lceil \rho_G \rceil$, a contradiction.

Suppose that $\tau(Q)/d(Q) = \lceil \rho_G \rceil$. Then

$$\begin{aligned} \forall\, i \in [1,q] \quad \delta(op_{1+i \bmod q}) &= \delta(op_i) + w(e_i) \\ &= \delta(op_i) + \tau(op_i) \Leftrightarrow \lceil \rho_G \rceil \cdot d(e_i) \end{aligned}$$

In fact

$$\forall\, i \in [1,q] \quad \delta(op_{1+i \bmod q}) \geq \delta(op_i) + \tau(op_i) \Leftrightarrow \lceil \rho_G \rceil \cdot d(e_i)$$

If one of the above inequalities was strict one would have

$$\tau(Q) \Leftrightarrow \lceil \rho_G \rceil \cdot d(Q) < 0$$

which contradicts our initial assumption $\tau(Q)/d(Q) = \lceil \rho_G \rceil$. Thus if $op_1 \rightarrow op_i$ denotes the subpath of Q from op_1 to op_i one can write

$$\delta(op_i) = \delta(op_1) + \tau(op_1 \rightarrow op_i) \Leftrightarrow \lceil \rho_G \rceil \cdot d(op_1, op_i)$$

and consequently the condition for cutting edge e_i becomes

$$(\delta(op_1) + \tau(op_1 \rightarrow op_i) + \tau(op_i)) \bmod \lceil \rho_G \rceil < (\delta(op_1) + \tau(op_1 \rightarrow op_i)) \bmod \lceil \rho_G \rceil + \tau(op_i)$$

thus if

$$\left\lfloor \frac{\tau(op_1 \rightarrow op_i) + \tau(op_i)}{\lceil \rho_G \rceil} \right\rfloor > \left\lfloor \frac{\tau(op_1 \rightarrow op_i)}{\lceil \rho_G \rceil} \right\rfloor$$

then an edge has necessarily been cut. Therefore at most

$$\left\lceil \frac{\tau(op_1 \rightarrow op_q)}{\lceil \rho_G \rceil} \right\rceil = \left\lceil \frac{d(Q) \cdot \lceil \rho_G \rceil \Leftrightarrow \tau(op_q)}{\lceil \rho_G \rceil} \right\rceil \leq d(Q)$$

edges are cut.

If $\tau(Q)/d(Q) < \lceil \rho_G \rceil$ then one can write

$$\forall\, i \in [1,q] \quad \delta(op_{1+i \bmod q}) = \delta(op_i) + \tau(op_i) \Leftrightarrow \lceil \rho_G \rceil \cdot d(e_i) + a(e_i)$$

for some $a(e_i) \geq 0$. If one poses $\alpha(op_1 \rightarrow op_i) = \tau(op_1 \rightarrow op_i) + a(op_1 \rightarrow op_i)$ then the condition for cutting edge e_i becomes

$$\begin{aligned} &(\delta(op_1) + \alpha(op_1 \rightarrow op_i) + \alpha(op_i)) \bmod \lceil \rho_G \rceil < \\ &\qquad (\delta(op_1) + \alpha(op_1 \rightarrow op_i)) \bmod \lceil \rho_G \rceil + \tau(op_i) \end{aligned}$$

Since $\tau(op_i) \leq \alpha(op_i)$ the condition

$(\delta(op_1) + \alpha(op_1 \rightarrow op_i) + \alpha(op_i)) \bmod \lceil \rho_G \rceil < (\delta(op_1) + \alpha(op_1 \rightarrow op_i)) \bmod \lceil \rho_G \rceil + \alpha(op_i)$

deletes more edges from Q then does algorithm 4.7.1. Now by employing exactly the same reasoning in the case $\tau(Q)/d(Q) = \lceil \rho_G \rceil$ one can show that this last condition deletes at most $d(Q)$.

It remains to show $\tau(P) < \lceil \rho_G \rceil$ for every path P of D'. Let P comprise edges (op_1, op_2), ..., (op_p, op_{p+1}) and suppose that $\lceil \rho_G \rceil \leq \tau(P)$. Then

$$\forall\, i \in [1, p] \quad \delta(op_i) \bmod \lceil \rho_G \rceil + \tau(op_i) \leq \delta(op_{i+1}) \bmod \lceil \rho_G \rceil$$

and therefore

$$\lceil \rho_G \rceil \leq \delta(op_1) \bmod \lceil \rho_G \rceil + \tau(P) \leq \delta(op_{p+1}) \bmod \lceil \rho_G \rceil$$

a contradiction. □

The last result of this chapter bounds the asymptotic performance of the scheduling algorithm which combines the results of theorems 4.7.1 and 4.7.2.

Theorem 4.7.3 *Let $\mathcal{L}$ be a recurrent cyclic system with core operation set O and $\mathcal{C}_{\text{opt}}$ an m-optimum set of schedules for $\mathcal{L}$. Let D', B and B_χ^∞ denote the same entities as in theorem 4.7.1 and $\tau_{\max} = \max_{op \in O} \tau(op)$. If the dependence graph D' is generated by algorithm 4.7.1 and the schedule B is produced by a list scheduling algorithm then*

$$\frac{\Delta(B_\chi^\infty, \mathcal{L})}{\Delta(\mathcal{C}_{\text{opt}}, \mathcal{L})} \leq (2 \Leftrightarrow \frac{1}{m}) \cdot \frac{(\tau_{\max} \Leftrightarrow 1) + \lceil \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L}) \rceil}{\Delta(\mathcal{C}_{\text{opt}}, \mathcal{L})}$$

Proof: The asymptotic performance $\Delta(B_\chi^\infty, \mathcal{L})$ is simply $|B|$. Let O be $\mathcal{L}$'s core operation set. The asymptotic performance $\Delta(\mathcal{C}_{\text{opt}}, \mathcal{L})$ is clearly bounded by the duration of operations in O and the number of available processors m:

$$\frac{1}{m} \cdot \sum_{op \in O} \tau(op) \leq \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L})$$

Furthermore consider the constraint imposed by $\mathcal{L}$'s dependence graph D on the scheduling of operations. Every cycle Q in D forces the asymptotic performance $\Delta(\mathcal{C}_{\text{opt}}, \mathcal{L})$ to be at least $\tau(Q)/d(Q)$, thus if K_D denotes the critical cycle of D

$$\frac{\tau(K_D)}{d(K_D)} \leq \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L})$$

Let $\dot{I}_1, \dot{I}_2, \ldots, \dot{I}_c$ respectively be the first, second, ..., last instruction in B containing less than m operations and let $K_{D'}$ the critical path of D'. Because B is generated by a list scheduling algorithm there must exist a dependence path P in D' dependence graph such that $c \leq \tau(P) + \tau_{\max}$. Thus the critical path of D' is such that $c \leq \tau(K_{D'}) + \tau_{\max}$. Because D' was constructed by algorithm 4.7.1 it must be that

$$c \leq \tau(K_{D'}) + \tau_{\max} < \tau_{\max} + \left\lceil \frac{\tau(K_D)}{d(K_D)} \right\rceil$$

Thus

$$\begin{aligned} m \cdot |B| &\leq \sum_{op \in O} \tau(op) + (m \Leftrightarrow 1) \cdot c \\ &\leq m \cdot \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L}) + (m \Leftrightarrow 1) \cdot ((\tau_{\max} \Leftrightarrow 1) + \lceil \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L}) \rceil) \end{aligned}$$

Finally

$$\begin{aligned} \frac{|B|}{\Delta(\mathcal{C}_{\text{opt}}, \mathcal{L})} &\leq 1 + \frac{m \Leftrightarrow 1}{m} \cdot \frac{(\tau_{\max} \Leftrightarrow 1) + \lceil \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L}) \rceil}{\Delta(\mathcal{C}_{\text{opt}}, \mathcal{L})} \\ &\leq (2 \Leftrightarrow \frac{1}{m}) \cdot \frac{(\tau_{\max} \Leftrightarrow 1) + \lceil \Delta(\mathcal{C}_{\text{opt}}, \mathcal{L}) \rceil}{\Delta(\mathcal{C}_{\text{opt}}, \mathcal{L})} \end{aligned}$$

$\square$

Note that if $\mathcal{L}$'s core operation set contains few operations or if some operation requires long execution time then the bound on optimum asymptotic performance may be poor. To improve it it suffices to operate on $u^x(\mathcal{L})$ for some sufficiently large $x > 1$.

By coupling theorem 4.7.1 with algorithm 4.7.1 it is therefore possible to construct in polynomial time a periodic schedule with close to optimum asymptotic performance.

Chapter 5

Branching Task System

This chapter extends the task system model of chapter 3 by adding conditionals. A branching task system formalizes the intuitive notion of an acyclic program containing branches. In order to avoid confusion between the tasks and schedules of chapter 3 and those of this chapter, one qualifies the first as "straight line". Furthermore all branching entities will be hatted (ˆ) whereas straight line entities will not.

5.1 Preliminaries

A control flow graph is a single entry, single exit possibly infinite di-graph G such that no vertex has out-degree greater than two and the entry has out-degree one. The entry of a control flow graph is denoted ξ and the exit ζ.

A basic block of G is a single entry single exit subgraph of G with maximal number of vertices such that each vertex has in-degree and out-degree one, except may be for the entry, which can have in-degree greater than one and the exit which can have out-degree greater than one.

Let G be an acyclic, possibly infinite, control flow graph. A path P in G is perfectly determined by the vertices that it traverses. In fact for any subset of G's vertices there exists at most one path which traverses exactly those vertices. Thus if G is an acyclic control flow graph one can consider every path P to be a set of vertices instead of being a sequence of edges.

5.2 Branching Task System

The tasking model of chapter 3 is extended by introducing conditionals, that is operations whose outcome determines the next set of operations to execute.

Definition 5.2.1 *A branching task system, or more briefly branching task, $\hat{T}$ is a quadruple $\hat{T} = (O, \tau, G, ,)$ where:*

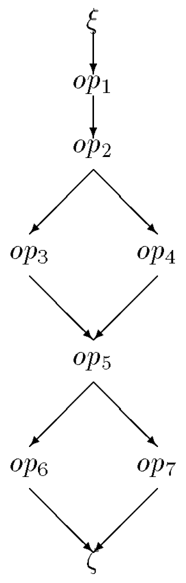
1. *O, the operation set of $\hat{T}$, is a non empty set of operations. Note that O could be infinite in which case $\hat{T}$ is said to be infinite.*
2. *τ, the duration function of $\hat{T}$, is a function mapping O into $[1, \infty]$. Like for straight line task systems, τ specifies the number of cycles required to execute each operation in O.*
3. *G, the control flow graph of $\hat{T}$, is an acyclic control flow graph with vertex set $O \cup \{\xi, \zeta\}$. Operations with an out-degree of two are called conditionals. For simplicity it is assumed that the duration $\tau(cj)$ of a conditional cj is 1 cycle. A path from the entry ξ to the exit ζ is called an execution path of $\hat{T}$. The set of all such execution paths is denoted $Path(\hat{T})$. For $op \in O$, $Path(op, \hat{T})$ denotes the set of execution paths traversing op.*
4. *$, = \{\prec_P\}_{P \in Path(\hat{T})}$ is the set of dependence relations of $\mathcal{L}$. For each execution path $P \in Path(\hat{T})$, $\prec_P$ is a partial order on P compatible with its linear ordering, that is for any $op, op' \in P$, $op \prec_P op'$ only if op precedes op' in P.*

For every $P \in Path(\hat{T})$ the function τ can trivially be restricted to the operations in P. The triple $\hat{T}(P) = (P, \tau, \prec_P)$ is a straight line task system called the restriction of $\hat{T}$ to P.

An example of a branching task system $\hat{T}$ is given in figure 5.1(a).

The limitation on the duration of conditionals was imposed for the sake of formalism simplicity. The results stated in the following sections are unaffected should conditionals require several cycles to complete. In practice several horizontal systems [15,12,30] require that multicycle tests such as

```
if a = b then goto L1
```



control flow graph of $\hat{T}$

$Path(\hat{T}) = \{P_1, P_2, P_3, P_4\}$

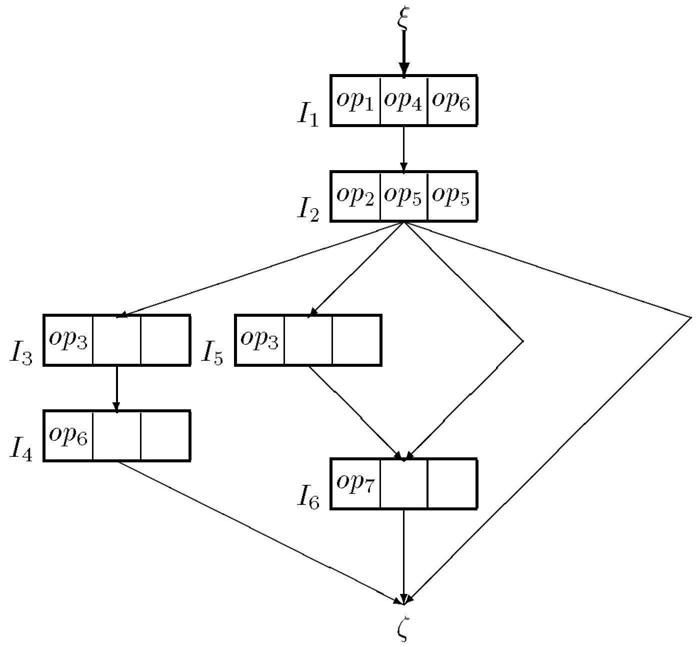
$P_1 : \xi\, op_1\, op_2\, op_3\, op_5\, op_6\, \zeta$
$P_2 : \xi\, op_1\, op_2\, op_3\, op_5\, op_7\, \zeta$
$P_3 : \xi\, op_1\, op_2\, op_4\, op_5\, op_6\, \zeta$
$P_4 : \xi\, op_1\, op_2\, op_4\, op_5\, op_7\, \zeta$

$\tau(op_1) = \cdots = \tau(op_7) = 1$

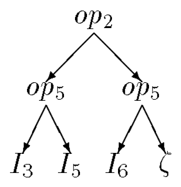
$, = \{\prec_{P_1}, \prec_{P_2}, \prec_{P_3}, \prec_{P_4}\}$

$\prec_{P_1}$: $op_1 \prec_{P_1} op_2$, $op_1 \prec_{P_1} op_5$, $op_3 \prec_{P_1} op_6$
$\prec_{P_2}$: $op_1 \prec_{P_2} op_2$, $op_1 \prec_{P_2} op_5$, $op_3 \prec_{P_2} op_7$
$\prec_{P_3}$: $op_1 \prec_{P_3} op_2$, $op_1 \prec_{P_3} op_5$
$\prec_{P_4}$: $op_1 \prec_{P_4} op_2$, $op_1 \prec_{P_4} op_5$

(a)



decision tree in I_2



(b)

Figure 5.1: (a) A branching task system $\hat{T}$. (b) A 3-admissible schedule $\hat{C}$ for $\hat{T}$.

be separated in two operations:

```
cc := a = b
if cc then goto L1
```

so that the actual branching can be executed by the hardware in a single cycle.

The last point in definition 5.2.1 requires some explanation and justification. For a given execution path P the partial order $\prec_P$ models the flow of values among P's operations, should this path be taken at run time (flow dependences [36]). Definition 5.2.1 allows for operations op, op' and execution paths P, Q to verify $op \prec_P op'$ and $op \not\prec_Q op'$ even though P and Q may share the same subpath from op to op'. Such situation is possible in computer programs where aliasing is permitted. Consider for instance the two programs given in figure 5.2. Each one has two execution paths P and Q. In both programs operation a(i):= $\cdots$ must precede operation $\cdots$:= a(j) on execution path P but not Q.

In practice, however, state of the art memory disambiguators are not able to pin point such differences in flow dependences, although some can in the special context of trace scheduling [44,18]. Usually worst case assumptions are made. In the above example $\cdots$:= a(j) would be considered to depend on a(i):= $\cdots$ on all execution paths. The next definition models such state of affairs.

Definition 5.2.2 *Let $\hat{T} = (O, \tau, G, \{\prec_P\}_{P \in Path(\hat{T})})$ be a branching task system. The dependence relations of $\hat{T}$ are said to be memoryless if and only if for all $P, Q \in Path(\hat{T})$ with some common subpath R one has:*

$$\forall op, op' \in R \qquad op \prec_Q op' \ \textit{ if and only if } \ op \prec_P op'$$

Intuitively a dependence between two operations op and op' is memoryless if it is independent of the particular path by which op is reached or by which op' is left. Memoryless dependences will be needed in section 5.7.

5.3 Machine Model and Branching Schedules

To allow execution of conditional operations, the machine model of chapter 3 is extended by permitting instructions to form a graph. Within an instruction conditionals are arranged to form a decision tree that specifies what instructions should be executed next. Conceptually one introduces the notion of a branching m-instruction.

```
if (cc) {            -- $cj$
    i = j;           -- $op_1$
}
else {
    i = j + 1;       -- $op_2$
};
a(i) = ···;          -- $op_3$
··· = a(j);          -- $op_4$
```

Execution paths:

$$\begin{aligned} P &= \{cj, op_1, op_3, op_4\} \\ Q &= \{cj, op_2, op_3, op_4\} \end{aligned}$$

(a)

```
a(i) = ···;          -- $op_1$
··· = a(j);          -- $op_2$
cc = (i == j);       -- $op_3$
if (cc) {            -- $cj$
    ···;             -- $op_4$
}
else {
    ···;             -- $op_5$
}
```

Execution paths:

$$\begin{aligned} P &= \{op_1, op_2, op_3, cj, op_4\} \\ Q &= \{op_1, op_2, op_3, cj, op_5\} \end{aligned}$$

(b)

Figure 5.2: Operation precedences may be execution path dependent.

Definition 5.3.1 *A branching m-instruction $\hat{I}$ is a finite rooted tree with at most m vertices. If $m = \infty$ then $\hat{I}$ can contain an unbounded but finite number of vertices. For notational simplicity $\hat{I}$ will also denote the set of its vertices. Every vertex $v \in \hat{I}$ is associated with some operation op(v). The operations associated with the vertices of $\hat{I}$ are said to be executing in $\hat{I}$.*

The notion of straight line schedule put forth in section 3.3 is adapted to allow for conditional operations. This is achieved by coupling a straight line schedule with a control flow graph and introducing the notion of instruction covering.

Definition 5.3.2 *Let G be a possibly infinite di-graph and $\hat{\mathcal{I}}$ a set of branching m-instructions. $\hat{\mathcal{I}}$ is said to cover G or to be an m-instruction covering of G if and only if:*

1. *Every $\hat{I} \in \hat{\mathcal{I}}$ is a subgraph of G.*
2. *Every vertex of G belongs to a unique $\hat{I} \in \hat{\mathcal{I}}$.*
3. *Let $\hat{I}, \hat{I}' \in \hat{\mathcal{I}}$. If an edge e in G goes from a vertex $u \in \hat{I}$ to a vertex $v \in \hat{I}'$ then either v is the entry of $\hat{I}'$ or $\hat{I} = \hat{I}'$ and e is an edge of $\hat{I}$.*

Definition 5.3.3 *A branching m-schedule $\hat{C}$ is a pair $\hat{C} = (G, \hat{\mathcal{I}})$ where G is a control flow graph and $\hat{\mathcal{I}}$ is an m-instruction covering of G. The code size of $\hat{C}$ is the cardinality of G's vertex set. A path from the entry to the exit of G is called an execution path of $\hat{C}$. For notational simplicity $\hat{C}$ will also denote the vertex set of its control flow graph G and the m-instruction covering $\hat{\mathcal{I}}$. For each vertex $v \in \hat{C}$, Path$(v, \hat{C})$ denotes the set of execution paths traversing v.*

A branching m-instruction $\hat{I}$ is portrayed as a straight line m-instruction I where each slot contains one of the operations executing in $\hat{I}$. If an operation *op* is associated with $k \in [1, m]$ vertices of $\hat{I}$ then *op* occupies k slots in I. If the decision tree in $\hat{I}$ is non trivial, that is it contains out-degree two vertices, the actual tree is given next to I. A branching m-schedule $\hat{C}$ is portrayed as a graph where each node represents a branching m-instruction. See figure 5.1(b).

The machine model previously introduced is directly inspired from the branching paradigms of Karplus & Nicolau and Ebcioğlu [34,15].

5.4 Admissibility

Given some branching task system $\hat{T}$ and the target machine $\mathcal{M}(m)$, one would like to extend the notion of m-admissibility introduced in section 3.4.

Definition 5.4.1 *Let $\hat{T}$ be some branching task system. A branching m-schedule $\hat{C}$ is said to be m-admissible for $\hat{T}$, denoted $\hat{C} \overset{m}{\Leftrightarrow} \hat{T}$, if and only if the following constraints are met:*

Branching: *A vertex v of $\hat{C}$ is of out-degree two if and only if $op(v)$ is a conditional of $\hat{T}$. Furthermore there exists a bijective function mapping the execution paths of $\hat{T}$ into those of $\hat{C}$. In the sequel the execution paths of $\hat{T}$ and those of $\hat{C}$ will be identified. Thus for each vertex $v \in \hat{C}$, $Path(v, \hat{C}) \subseteq Path(\hat{T})$.*

Semantic: *For each vertex $v \in \hat{C}$ there exists a set*

$$Use(v, \hat{T}) \subseteq Path(v, \hat{C}) \cap Path(op(v), \hat{T})$$

denoting the execution paths for which $op(v)$ is useful in $\hat{C}$, such that:

dependence: *For each $P \in Path(\hat{T})$ define $\hat{C}(P)$ to be the straight line m-schedule whose i-th instruction, $i \geq 1$, contains operation $op(v)$ if and only if vertex v belongs to the i-th branching instruction traversed by P in $\hat{C}$ and $P \in Use(v, \hat{T})$. The straight line schedule $\hat{C}(P)$ is called the restriction of $\hat{C}$ to P. The dependence constraint requires that:*

$$\forall\, P \in Path(\hat{T}) \quad \hat{C}(P) \overset{m}{\Leftrightarrow} \hat{T}(P)$$

flow of values: *Let v, v' two vertices in $\hat{C}$ for which there exists a path P in $Use(v, \hat{T}) \cap Use(v', \hat{T})$ such that $op(v) \prec_P op(v')$. Then the flow of values constraint requires that:*

$$\forall\, P' \in Path(v, \hat{C}) \cap Use(v', \hat{C}) \quad P' \in Path(op(v), \hat{T})$$

If for $v \in \hat{C}$, $Use(v, \hat{T}) \neq Path(v, \hat{C})$, $op(v)$ is said to be speculatively executed in $\hat{C}$.

An example of a 3-admissible branching schedule for the branching task of figure 5.1(a) is given in 5.1(b).

The branching constraint has very strong implications on the scheduling of conditionals. In fact the existence of a bijection between the execution paths of $\hat{T}$ and $\hat{C}$ entails that no conditional of $\hat{T}$ can be executed speculatively in $\hat{C}$. The reason for this constraint is illustrated in the program of figure 5.3. Assume that b is at the end

```
⋮
if (a>b) {
   d = c div a;
   if (d>2) {
     ⋮
   }
}
⋮
```

Figure 5.3: A fragment of a real life sequential program.

of a long chain of dependences whereas the values of c and a are readily available. A scheduler may decide to execute operations d=c div a and if (d>2) before if (a>b). If the value of a is zero d will be undefined. An exception should be raised only after test if (a>b) has taken the true branch. Thus if no special hardware mechanisms are provided d=c div a and if (d>2) cannot be speculatively executed. Such hardware mechanisms are relatively inexpensive to implement for simple operations and are often provided in VLIW machines [12,15]. However hardware mechanisms to circumvent the problem of branching on undefined conditions are more expensive and no current design incorporates such feature.

The purpose of the semantic constraint of definition 5.4.1 is to ensure that correct flow of values occurs in branching schedule $\hat{C}$. It may appear strange to speak about values when operations are seen as purely atomic entities. However if the partial orders of $\hat{T}$ are interpreted as flow dependences, not only does the dependence constraint ensure that flow dependences are respected in each execution path of $\hat{C}$ but the flow of values constraint enforces that no operation in $\hat{C}$ receives the "wrong value". More specifically if

$$\exists\, v, v' \in \hat{C} \quad \exists\, P \in Use(v,\hat{T}) \cap Use(v',\hat{T}) \qquad op(v) \prec_P op(v')$$

that is there exists flow of values along execution path P from a vertex v to a vertex v' then for every other execution path P' traversing both v and v' either v' is not useful in P', that is $P' \notin Use(v', \hat{T})$, or P' must also traverse operation $op(v)$ in $\hat{T}$. The following example illustrates the flow of values constraint. Consider the program fragment of figure 5.4(a). Let P_1 be the execution path following the true branch of test if (!a) and P_2 the execution path following the false branch. Consider the schedule of figure 5.4(b) and let v, v' denote respectively the vertices with which operations x=f(k) and z=g(x) are associated. Clearly $Use(v, \hat{T}) = \{P_1\}$ and if there is only one copy of operation z=g(x) one must have $Use(v', \hat{T}) = \{P_1, P_2\}$. However path P_2 traverses v whereas it does not traverse operation x=f(k) in the original program so the schedule of figure 5.4(b) is semantically incorrect. Consider schedule 5.4(c) and let v, v' and v'' denote respectively the vertices with which operations x'=f(k) z=g(x') and z=g(x) are associated. By having $Use(v, \hat{T}) = \{P_1\}$, $Use(v', \hat{T}) = \{P_1\}$ and $Use(v'', \hat{T}) = \{P_2\}$ it is easily seen that definition 5.4.1 accepts schedule 5.4(c) as an admissible schedule for the branching task in 5.4(a). Note that renaming is implicit in the model and is always possible if sufficient registers are available.

5.5 Performance Criteria

Depending on the outcome of the conditionals contained in a branching task system $\hat{T}$, the actual path followed during execution varies. Consequently an m-admissible branching schedule for $\hat{T}$ may require different completion times for different executions. It may therefore be that for two execution paths P_1, P_2 and two m-admissible branching schedules $\hat{C}$, $\hat{C}'$ for $\hat{T}$, $|\hat{C}(P_1)| < |\hat{C}'(P_1)|$ and $|\hat{C}(P_2)| > |\hat{C}'(P_2)|$. Because of this phenomenon performance is initially defined as a relative measure, that is a partial order.

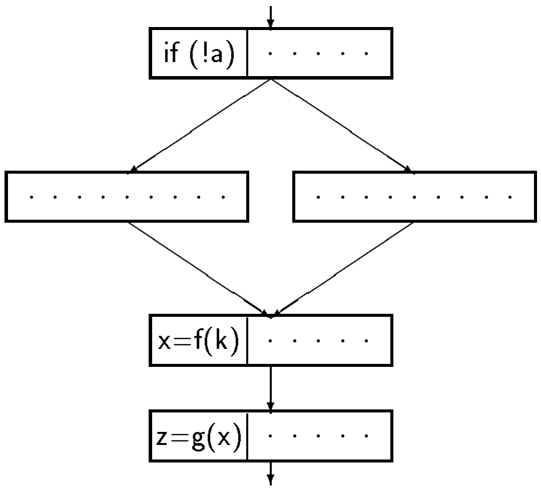
Definition 5.5.1 *Let $\hat{T}$ be a branching task system and $\hat{C}$, $\hat{C}'$ two m-admissible branching schedules for $\hat{T}$. $\hat{C}$ is said to have better performance than $\hat{C}'$ if and only if:*

$$\forall\, P \in Path(\hat{T}) \quad |\hat{C}(P)| \le |\hat{C}'(P)| \quad \text{and} \quad \exists\, P_0 \in Path(\hat{T}) \quad |\hat{C}(P_0)| < |\hat{C}'(P_0)|$$

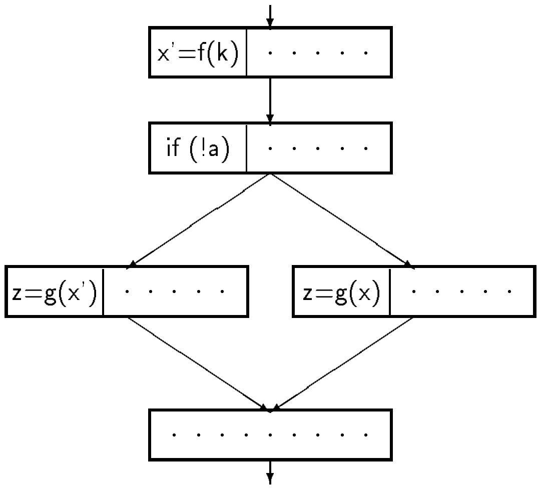
Furthermore $\hat{C}$ is said to be a branching m-optimum for $\hat{T}$ if and only if there exists no branching schedule with better performance than $\hat{C}$.

```
if (!a) {
    x=f(k);
    ⋮
}
else {
    ⋮
}
z=g(x);
```

(a)



(b)



(c)

Figure 5.4: (a) Some fragment of a real life sequential program. (b) A semantically incorrect schedule for the program given in (a). (c) An admissible schedule.

For $\mathcal{M}(\infty)$ and $\mathcal{M}(1)$ any execution path P of a branching task system $\hat{T}$ takes the same amount of time to complete in all branching optima for $\hat{T}$. For $\mathcal{M}(\infty)$ this time is the length of the longest dependence chain in $\hat{T}(P)$. In general however, $1 < m < \infty$, an execution path of $\hat{T}$ may require disparate running times in different branching optima. Furthermore the number of such paths and the difference in speeds can be significant.

Theorem 5.5.1 *There exists a branching task system $\hat{T}$ and two branching m-optima $\hat{C}_o$, $\hat{C}'_o$ for $\hat{T}$ such that:*

$$\frac{m}{4} < \frac{1}{|Path(\hat{T})|} \cdot \sum_{P \in Path(\hat{T})} \frac{|\hat{C}_o(P)|}{|\hat{C}'_o(P)|}$$

Proof: Consider the branching task $\hat{T} = (O, \tau, \{\prec_P\}_{P \in Path(\hat{T})}, G)$ where:

1. $O = \{cj_1, op_{11}, \cdots, op_{1m-1}, \cdots, cj_m op_{m1}, \cdots, op_{mm-1}\}$.
2. For all $op \in O$, $\tau(op) = 1$.
3. The control flow graph G is given in figure 5.5(a). For $1 \leq i \leq m$, basic block B_i contains conditional cj_i whereas basic block B'_i contains operation $op_{i1}, \cdots, op_{im-1}$.
4. The precedence relations $\{\prec_P\}_{P \in Path(\hat{T})}$ are all empty.

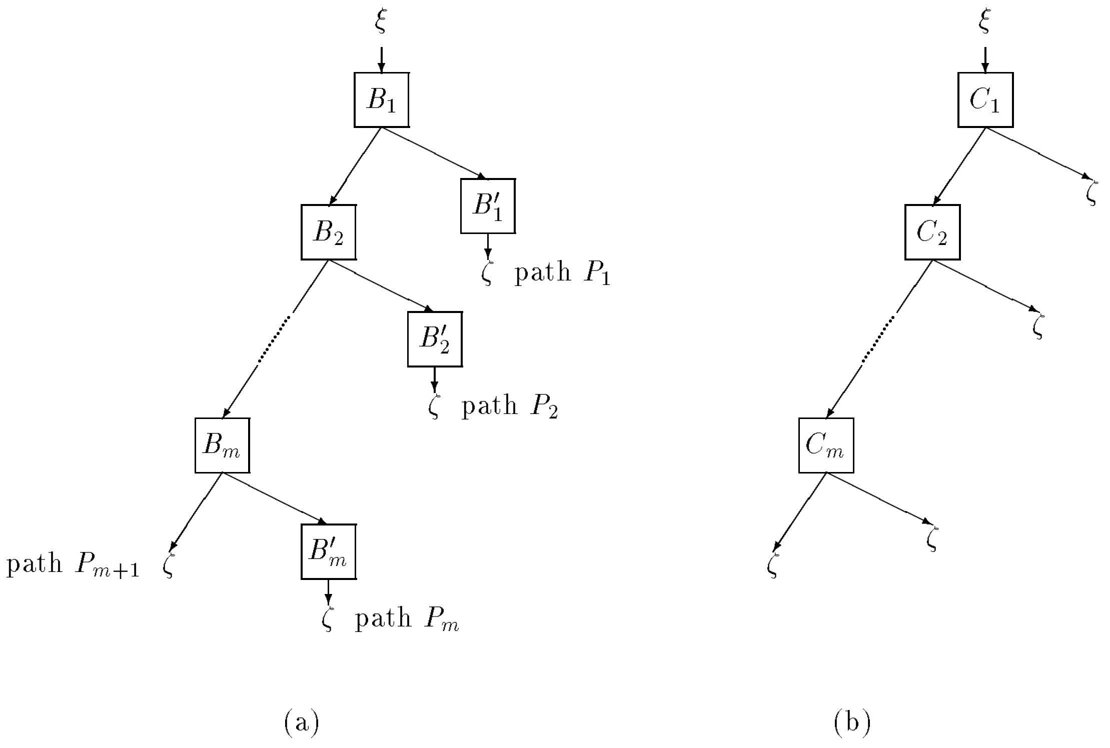
The set $Path(\hat{T}) = \{P_1, \cdots, P_{m+1}\}$.

The control flow graphs of m-optima $\hat{C}_o$ and $\hat{C}'_o$ are given in 5.5(b) and 5.5(c) respectively. In $\hat{C}_o$ basic block C_i, $1 \leq i \leq m$, contains a single instruction comprising operations $cj_1, op_{i1}, \cdots, op_{im-1}$. As in $\hat{C}_o$, all basic blocks of $\hat{C}'_o$ also contain a single instruction. For $1 \leq i \leq m$, the instruction in C'_i contains operations $op_{i1}, \cdots, op_{im-1}$, whereas the instruction in C'_0 comprises conditionals $cj_1, \cdots, cj_m$. For $1 \leq i \leq m$, $\hat{C}_o(P_i) = i$ and $\hat{C}'_o(P_i) = 2$, whereas $\hat{C}_o(P_{m+1}) = m$ and $\hat{C}'_o(P_{m+1}) = 1$. Thus

$$\frac{1}{|Path(\hat{T})|} \cdot \sum_{P \in Path(\hat{T})} \frac{|\hat{C}_o(P)|}{|\hat{C}'_o(P)|} = \frac{1}{m+1} \cdot \left(m + \sum_{i=1}^{m} \frac{i}{2} \right) > \frac{m}{4}$$

□

Because for $1 < m < \infty$ branching optima may require quite different completion times for the same execution path, global performance measures have to be superimposed to the partial order defined in 5.5.1. To this purpose the notion of weighting function is introduced.



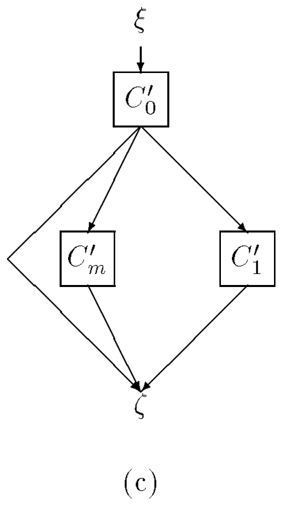


Figure 5.5: (a) Control flow graph of $\hat{T}$. Squares indicate basic blocks. (b) Control flow graph of $\hat{C}_o$. (c) Control flow graph of $\hat{C}_o'$.

Definition 5.5.2 *Let $\hat{T}$ be a branching task and G its control flow graph. A function w mapping $Path(\hat{T})$ into the non-negative reals is called a weighting function for $\hat{T}$ if and only if:*

$$\sum_{P \in Path(\hat{T})} w(P) = 1$$

If there exists a function ϖ mapping the edges of G into the non-negative reals such that:

$$\forall\ P = (e_1, e_2, \cdots, e_k) \in Path(\hat{T}) \qquad w(P) = \prod_{i=1}^{k} \varpi(e_i)$$

one says that the weighting function w is markovian or has the Markov property. For every real $\alpha \geq 1$, a markovian weighting function is said to be α-skewed, or have a skew factor of α, if and only if for every vertex of G with two outgoing edges e_1 and e_2, $\varpi(e_1) = \alpha \cdot \varpi(e_2)$. A 1-skewed weighting function is also called isotropic.

It is easy to show that the weighting function of any branching schedule whose control flow graph minus ζ is a tree is markovian. In general, however, this is not the case. Weighting functions are used in defining global performance measures.

Definition 5.5.3 *Let $\hat{T}$ be a branching task, w a weighting function for $\hat{T}$ and $\hat{C} \overset{m}{\Leftrightarrow\!\!\rightarrow} \hat{T}$ a branching schedule. The weighted average running time of $\hat{C}$, denoted $w(\hat{C})$, is defined as:*

$$w(\hat{C}) = \sum_{P \in Path(\hat{T})} w(P) \cdot |\hat{C}(P)|$$

If $\hat{T}$ is finite then w is always defined, otherwise it need not be. $\hat{C}$ is said to be m-optimum for w, or w-optimum, if and only if $w(\hat{C})$ converges and there exists no $\hat{C}' \overset{m}{\Leftrightarrow\!\!\rightarrow} \hat{T}$ such that $w(\hat{C}') < w(\hat{C})$.

If none of the weights of the execution paths of a branching task system $\hat{T}$ is zero, an m-admissible schedule for $\hat{T}$ which is optimum for w must also be a branching optimum.

Usually weights are taken to be path probabilities. In practice, however, accurate execution probabilities might not be easy to obtain because of the amount of profiling information that needs to be collected. In fact simple branch probabilities, i.e. the probability that a conditional will take the true or false branch, are insufficient as path probabilities do not necessarily possess the Markov property. When no profiling information is available the weighting function can solely be based on the structural properties of the execution paths such as their sequential length or their optimum length on the target machine.

It is possible to express the weighted average running time of a branching schedule in a different form than that of definition 5.5.3. This form will be quite useful in the proofs of theorems to follow.

Property 5.5.1 *Let $\hat{T}$ be a branching task, w a weighting function for $\hat{T}$, $\hat{C} \overset{m}{\Leftrightarrow\!\!\!\rightarrow} \hat{T}$ a branching schedule and $\hat{I}$ an instruction of $\hat{C}$. If one defines the weight of $\hat{I}$, denoted $w(\hat{I})$, as the sum of the weights of the execution paths traversing $\hat{I}$ then*

$$w(\hat{C}) = \sum_{\hat{I} \in \hat{C}} w(\hat{I})$$

Proof: Let $v_{\hat{I}}$ denote the root of branching instruction $\hat{I} \in \hat{C}$. Then

$$\begin{aligned} w(\hat{C}) &= \sum_{P \in Path(\hat{T})} w(P) \cdot |\hat{C}(P)| \\ &= \sum_{P \in Path(\hat{T})} w(P) \sum_{\hat{I} \in \hat{C}(P)} 1 \\ &= \sum_{\hat{I} \in \hat{C}} \sum_{P \in Path(v_{\hat{I}}, \hat{C})} w(P) \\ &= \sum_{\hat{I} \in \hat{C}} w(\hat{I}) \end{aligned}$$

□

5.6 Performance Limitations and Trade-Offs

This section establishes a series of negative results concerning w-optimum branching schedules. The first of such results is the existence of branching tasks for which space performance has to be sacrificed in order to obtain even the more modest speedups. More specifically exponential code size is necessary to obtain speedups as little as two. Thus for branching tasks time and space performance can be antipodal.

Theorem 5.6.1 *For every real $\alpha \geq 1$ there exists a branching task $\hat{T}$ with arbitrarily big operation set O such that for every α-skewed weighting function w, m-optimum for w $\hat{C}_{\text{opt}}$ and branching schedule $\hat{C} \overset{m}{\Leftrightarrow} \hat{T}$ with code size polynomial in $|O|$, one has:*

$$m \Leftrightarrow 1 < \frac{w(\hat{C})}{w(\hat{C}_{\text{opt}})}$$

Proof: Let $\hat{T} = (O, \tau, \{\prec_P\}_{P \in Path(\hat{T})}, G)$ where

1. $O = \{cj_1, op_1^t, op_1^f, \cdots, cj_n, op_n^t, op_n^f\} \cup \bigcup_{j=1}^{m} \{op_1^j, \cdots, op_q^j, \tilde{op}_1^j, \cdots, \tilde{op}_n^j\}$
   where n and $q = n^2$ are two positive integers and m denotes as usual the number of available processors.
2. For all $op \in O$, $\tau(op) = 1$.
3. The control flow graph G is given in figure 5.6. Basic block B contains operations
   $$\bigcup_{j=1}^{m} \{op_1^j, \cdots, op_q^j, \tilde{op}_1^j, \cdots, \tilde{op}_n^j\}$$
4. The precedence relations $\{\prec\}_{P \in Path(\hat{T})}$ are as follows:

$$\begin{array}{rcl}
\forall\, i \in [1, n \Leftrightarrow 1] & \forall\, P \in Path(op_i^t, \hat{T}) & cj_i \prec_P op_i^t \prec_P cj_{i+1} \\
 & \forall\, P \in Path(op_i^f, \hat{T}) & cj_i \prec_P op_i^f \prec_P cj_{i+1} \\
\forall\, j \in [1, m] & \forall\, P \in Path(op_n^t, \hat{T}) & cj_n \prec_P op_n^t \prec_P op_i^j \\
 & \forall\, P \in Path(op_n^f, \hat{T}) & cj_n \prec_P op_n^f \prec_P op_1^j \\
\forall\, j \in [1, m] & \forall\, P \in Path(\hat{T}) & op_1^j \prec_P op_2^j \prec_P \cdots \prec_P op_q^j \\
\forall\, i \in [1, n] & \forall\, j \in [1, m \Leftrightarrow 1] & op_q^j \prec_P \tilde{op}_i^j \text{ if } P \in Path(op_i^t, \hat{T}) \\
 & & \tilde{op}_i^j \prec_P op_1^{j+1} \text{ if } P \in Path(op_i^f, \hat{T})
\end{array}$$

Consider two distinct execution paths $P_1, P_2 \in Path(\hat{T})$. Because $P_1 \neq P_2$ there exists at least one $i \in [1, n]$ such that $P_1 \in Path(op_i^t, \hat{T})$ and $P_2 \in Path(op_i^f, \hat{T})$. If P_1 and P_2 where to share the same schedule for basic block B then such schedule would have to be sequential since

$$\forall\, j \in [1, m \Leftrightarrow 1] \quad op_q^j \prec_{P_1} \tilde{op}_i^j \prec_{P_2} op_1^{j+1}$$

Thus if one disregards the exit node ζ, the control flow graph of an m-optimum $\mathcal{C}_{\text{opt}}$ for w must be a tree, furthermore

$$2 \cdot n + q \leq w(\mathcal{C}_{\text{opt}}) \leq 2 \cdot n + (q + n)$$

Let $\mathcal{C}$ be an m-admissible schedule for $\hat{T}$ with code size polynomial in $|O| = 3 \cdot n + m \cdot (n + q) = 3 \cdot n + m \cdot (n + n^2)$, since $q = n^2$. Let $\mathcal{P}_a$ be the set of execution paths which do not share the schedule for basic block B with any other path. Clearly $|\mathcal{P}_a|$ must be polynomial in n. Let d be the degree of such polynomial, then there exists a constant $k \geq 1$ such that

$$\forall n \geq 1 \qquad w_a = \sum_{P \in \mathcal{P}_a} w(P) \leq \left(\frac{\alpha}{\alpha + 1}\right)^n \cdot k \cdot n^d$$

Because for all $\alpha \geq 1$

$$\lim_{n \to \infty} \left(\frac{\alpha}{\alpha + 1}\right)^n \cdot n^d = 0$$

by selecting a sufficiently large n, the weight w_a can be made as small as one desires. The ratio of $w(\hat{C})$ and $w(\hat{C}_{\text{opt}})$ can be expressed as a function of w_a. More precisely

$$\begin{aligned}
\frac{w(\hat{C})}{w(\hat{C}_{\text{opt}})} &\geq \frac{w_a \cdot (2 \cdot n + q) + (1 \Leftrightarrow w_a) \cdot (2 \cdot n + m \cdot q)}{2 \cdot n + (q + n)} \\
&\geq \frac{2 + (1 \Leftrightarrow w_a) \cdot m \cdot n + w_a \cdot n}{3 + n} \quad \text{since } q = n^2 \\
&> m \Leftrightarrow 1 \quad \text{for a sufficiently large } n
\end{aligned}$$

□

The above result shows that in the worst case space performance has to be sacrificed even for the more modest speedups. The next negative result concerns the time performance of optimum schedules. The optimum weighted average execution time of a branching task system is clearly limited by dependences. It is not immediately apparent, however, that conditionals may equally inhibit parallel execution. Indeed one can show that only logarithmic speedups can be achieved when the weighting function is slightly skewed and conditionals are abundant. This result is based on the following theorem.

Theorem 5.6.2 *Let $\hat{T}(n)$ be a branching task system such that all of its execution paths contain at least $n \geq 1$ conditionals. Let w be a α-skewed weighting function for $\hat{T}(n)$ and*

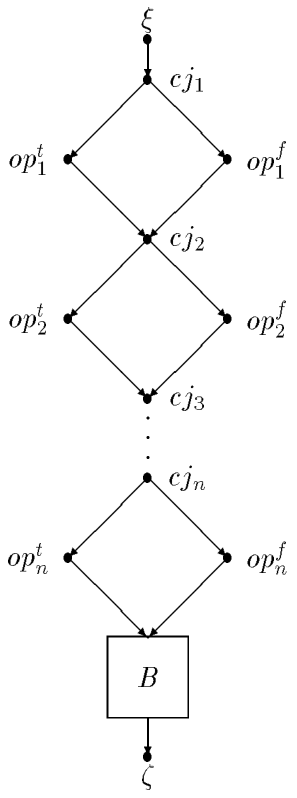


Figure 5.6: Control Flow graph of $\hat{T}$. B indicates a basic block of operations.

$\hat{C}_{\text{opt}}(n) \overset{m}{\Leftrightarrow} \hat{T}(n)$ an m-optimum for w. For $m \geq 3$ let $\alpha_1(m)$ and $\alpha_2(m)$ respectively denote the roots of the following two equations

$$x^{\lceil \log_2(m+1) \rceil} \Leftrightarrow x \Leftrightarrow 1 = 0 \qquad\qquad x^{m-1} \Leftrightarrow (x+1)^{m-2} = 0$$

Then:

$$\frac{w(\hat{C}_{\text{opt}}(n))}{n} \geq a(\alpha) \quad \textit{where} \quad a(\alpha) = \begin{cases} \frac{1}{\lceil \log_2(m+1) \rceil} & \textit{if } \alpha \in [1, \alpha_1(m)] \\ \frac{(\alpha+1)^{m-1}}{(\alpha+1)^m - \alpha^m} & \textit{if } \alpha \in [\alpha_2(m), \infty] \end{cases}$$

The numbers $\alpha_1(m)$ and $\alpha_2(m)$ are tabulated in figure 5.7 for some values of m.

Proof: In the following proof all weighting functions are assumed to be α-skewed. Furthermore these functions are all denoted by w although they refer to different branching tasks.

Consider the new task system $\hat{T}_t(n)$ whose control flow graph minus the exit is a completely balanced binary tree of height n and each operation, except for the leaves of the binary tree, is a conditional. There are no dependences between the operations of $\hat{T}_t(n)$. Let $\mathcal{C}^t_{\text{opt}}(n)$ be an m-optimum for $\hat{T}_t(n)$ and w. Because the computational model employed requires branching instructions to be trees it is clear that

$$w(\mathcal{C}^t_{\text{opt}}(n)) \leq w(\mathcal{C}_{\text{opt}}(n))$$

Furthermore for $i, j \geq 1$ let $\mathcal{C}^t_{\text{opt}}(i+j)$, $\mathcal{C}^t_{\text{opt}}(i)$ and $\mathcal{C}^t_{\text{opt}}(j)$ be respectively m-optimum schedules for $\hat{T}_t(i+j)$, $\hat{T}_t(i)$ and $\hat{T}_t(j)$, and the α-skewed weighting function w. Then

$$\forall\, i, j \geq 1 \qquad w(\mathcal{C}^t_{\text{opt}}(i+j)) \leq w(\mathcal{C}^t_{\text{opt}}(i)) + w(\mathcal{C}^t_{\text{opt}}(j))$$

for one can construct an m-admissible schedule for $\hat{T}_t(i+j)$ with weighted average execution time equal to $w(\mathcal{C}^t_{\text{opt}}(i)) + w(\mathcal{C}^t_{\text{opt}}(j))$ by having every incoming edge of $\mathcal{C}^t_{\text{opt}}(i)$'s exit point to a copy of $\mathcal{C}^t_{\text{opt}}(j)$. Thus the same type of proof employed in chapter 4 theorem 4.6.4 can be adopted here to show the existence of a constant $a(\alpha)$ such that

$$\forall\, n \geq 1 \qquad a(\alpha) \leq \frac{w(\mathcal{C}^t_{\text{opt}}(n))}{n} \qquad \text{and} \qquad \lim_{n \to \infty} \frac{w(\mathcal{C}^t_{\text{opt}}(n))}{n} = a(\alpha)$$

Because $w(\mathcal{C}^t_{\text{opt}}(n)) \leq w(\mathcal{C}_{\text{opt}}(n))$ the above shows that

$$a(\alpha) \leq \frac{w(\mathcal{C}_{\text{opt}}(n))}{n}$$

Let $\hat{I}$ be the first instruction in $\mathcal{C}^t_{\text{opt}}(n)$, that is the instruction just after the entry node ξ of $\mathcal{C}^t_{\text{opt}}(n)$. For $i \geq 1$ let $\{\hat{I}^i_1, \cdots, \hat{I}^i_p\}$ denote the instructions in $\mathcal{C}^t_{\text{opt}}(n)$ immediately following $\hat{I}$ such that exactly i conditionals need to be traversed in $\hat{I}$ from its root to reach the root of an $\hat{I}^i_j$. Finally let w_i be the sum of the weights of the $\hat{I}^i_j$, $1 \leq j \leq p$:

$$w_i = \sum_{j=1}^{p} w(\hat{I}^i_j)$$

Note that $\sum_{i=1}^{m} w_i = 1$. Because the weighting function w of $\hat{T}_t(n)$ is α-skewed if $w(\mathcal{C}^t_{\text{opt}}(n))$ is denoted by t_n one can write:

$$t_n = 1 + \sum_{i=1}^{m} w_i t_{n-i}$$

For a theoretical treatment and the common terminology revolving around such recurrence equations see for instance [26,48,24]. If one sets $w_0 = \Leftrightarrow 1$, the characteristic equation of the above linear recurrence is:

$$\sum_{i=0}^{m} w_{m-i} \cdot z^i = 0$$

Let $\lambda_1, \ldots, \lambda_r$ be the roots of the above characteristic equation and $d_1, \ldots, d_r$ their multiplicities. The general theory of linear recurrences dictates that t_n has the form:

$$t_n = \sum_{i=1}^{r} \sum_{j=0}^{d_i - 1} k_{ij} \cdot n^j \cdot \lambda_i^n$$

where k_{ij} is some unknown constant. Because $\lim_{n \to \infty} t_n / n = a(\alpha)$ it must be that $t_n = a(\alpha) \cdot n + k + o(1)$, where k is some constant and $o(1)$ denotes a function which tends towards 0 as n tends towards infinity. By using this equation in conjunction with the recurrence equation, $a(\alpha)$ can be expressed in terms of the w_i:

$$1 + a(\alpha) \cdot \sum_{i=1}^{m} w_i \cdot (\Leftrightarrow i) = 0$$

To compute the smallest possible value of $a(\alpha)$ it suffices to maximize

$$W(\hat{I}) = \sum_{i=1}^{m} w_i \cdot i$$

Consider once more instruction $\hat{I}$, the first instruction in $\mathcal{C}^t_{\text{opt}}(n)$. $W(\hat{I})$ cannot be maximum unless the greatest weight first condition is satisfied, that is for all conditional cj executing in $\hat{I}$ and for all instruction $\hat{I}'$ immediately following $\hat{I}$ in $\mathcal{C}^t_{\text{opt}}(n)$ one has

$$w(cj) \geq w(\hat{I}') \quad \text{where} \quad w(cj) = \sum_{P \in Path(cj, \hat{T}_t(n))} w(P)$$

In fact let cj be the conditional in $\hat{I}$ with smallest $w(cj)$ and let $\hat{I}'$ be the instruction immediately following $\hat{I}$ with the greatest $w(\hat{I}')$. Clearly both branches of cj point to operations scheduled outside of $\hat{I}$. Let r and p respectively denote the number of conditionals that need to be traversed in $\hat{I}$ from $\hat{I}$'s root to cj and from $\hat{I}$'s root to the root of $\hat{I}'$. If $w(\hat{I}') > w(cj)$, the root of $\hat{I}'$ cannot be reached from cj. Consider the instruction $\hat{I}_{\text{new}}$ which includes the root of $\hat{I}'$ and excludes cj and is otherwise equal to $\hat{I}$. Then

$$\begin{aligned} W(\hat{I}_{\text{new}}) &= W(\hat{I}) + [\Leftrightarrow p \cdot w(\hat{I}') + (p+1) \cdot w(\hat{I}')] + [\Leftrightarrow (r+1) \cdot w(cj) + r \cdot w(cj)] \\ &= w(\hat{I}') \Leftrightarrow w(cj) \end{aligned}$$

Thus if $w(cj) < w(\hat{I}')$, $W(\hat{I})$ cannot be maximum.

Consider now the two limit cases where the greatest weight first condition is satisfied and $\hat{I}$ is either a single path or every level of $\hat{I}$ is complete except may be for the last. This last case is analyzed first. For $\hat{I}$ to satisfy the greatest weight first condition and have every level completely filled except may be for the last, it is sufficient that

$$\left(\frac{\alpha}{1+\alpha}\right)^{\lceil \log_2(m+1) \rceil} \leq \left(\frac{1}{1+\alpha}\right)^{\lceil \log_2(m+1) \rceil - 1}$$

since $\lceil \log_2(m+1) \rceil \Leftrightarrow 1$ is the height of a binary tree where each level, except may be the last one, is complete. Thus if $\alpha_1(m)$ denotes the root of the equation

$$x^{\lceil \log_2(m+1) \rceil} \Leftrightarrow x \Leftrightarrow 1 = 0$$

it is sufficient that $\alpha \in [1, \alpha_1(m)]$. If the last level of $\hat{I}$ is allowed to be complete, even though $\hat{I}$ may contain more then m operations, one can only increase the value of $W(\hat{I})$. In this case the only non zero w_i is $w_{\lceil \log_2(m+1) \rceil}$ and therefore

$$W(\hat{I}) = \lceil \log_2(m+1) \rceil \cdot \underbrace{\sum_{i=0}^{\lceil \log_2(m+1) \rceil} \binom{\lceil \log_2(m+1) \rceil}{i} \cdot \frac{\alpha^i}{(1+\alpha)^{\lceil \log_2(m+1) \rceil}}}_{= w_{\lceil \log_2(m+1) \rceil}}$$

which implies $a(\alpha) = 1/\lceil \log_2(m+1) \rceil$ since $w_{\lceil \log_2(m+1) \rceil} = 1$.

Consider now the case where $\hat{I}$ is a path with m nodes. Then the greatest weight first condition is satisfied only if

$$\left(\frac{\alpha}{1+\alpha}\right)^{m-1} \leq \left(\frac{1}{1+\alpha}\right)$$

that is if $\alpha_2(m)$ denotes the root of the equation

$$x^{m-1} \Leftrightarrow (x+1)^{m-2} = 0$$

then $\alpha \in [\alpha_2(m), \infty]$. When $\hat{I}$ is a path containing m operations all of the weights $w_1, \cdots, w_{m+1}$ are non zero, and $W(\hat{I})$ is

$$W(\hat{I}) = m \cdot \left(\frac{\alpha}{\alpha+1}\right)^m + \sum_{l=0}^{m-1} (l+1) \cdot \frac{1}{\alpha+1} \cdot \left(\frac{\alpha}{\alpha+1}\right)^l$$

By posing $\alpha/(\alpha+1) = x$ and employing the following equalities

$$\begin{aligned} \sum_{l=0}^{m-1} (l+1) \cdot x^l &= \left(\sum_{l=0}^{m-1} x^{l+1}\right)' \\ &= \left(x \cdot \frac{1 \Leftrightarrow x^m}{1 \Leftrightarrow x}\right)' \\ &= \frac{m \cdot x^{m+1} \Leftrightarrow (m+1) \cdot x^m + 1}{(x \Leftrightarrow 1)^2} \end{aligned}$$

one obtains

$$W(\hat{I}) = \frac{(\alpha+1)^m \Leftrightarrow \alpha^m}{(\alpha+1)^{m-1}}$$

and therefore

$$a(\alpha) = \frac{(\alpha+1)^{m-1}}{(\alpha+1)^m \Leftrightarrow \alpha^m}$$

□

It would be interesting to give a closed formula for $a(\alpha)$ in the case $\alpha \in [\alpha_1(m), \alpha_2(m)]$. Such computation involves generalized binomial series $\mathcal{B}_t(z)$, where

$$\mathcal{B}_t(z) = \sum_{k=0}^{\infty} \binom{t \cdot k + 1}{k} \frac{1}{t \cdot k + 1} \cdot z^k$$

Unfortunately generalized binomial series cannot, in general, be expressed in closed form [24].

m	$\alpha_1(m)$	$\alpha_2(m)$
4	1.62	2.1
8	1.32	3.9
16	1.22	6.8
32	1.17	11.9
64	1.14	20.2

Figure 5.7: The numbers $\alpha_1(m)$ and $\alpha_2(m)$ for some values of m.

Thus if the weighting function is isotropic or close to, theorem 5.6.2 shows that only limited speedups can be achieved for branching tasks having a high fraction of conditionals.

Corollary 5.6.1 *Let $\hat{T}(n)$ be a branching task system where each execution path contains at least $n \geq 1$ conditionals and at most $f \cdot n$ operations, for a real $f \geq 1$. Let w be a α-skewed weighting function for $\hat{T}(n)$, $\hat{C}_{\text{seq}}(n)$ a branching 1-optimum for $\hat{T}$ and $\hat{C}_{\text{opt}}(n) \overset{m}{\Leftrightarrow} \hat{T}$ an m-optimum for w. Then:*

$$\frac{w(\hat{C}_{\text{seq}}(n))}{w(\hat{C}_{\text{opt}}(n))} \leq f \cdot s(\alpha) \quad \textit{where} \quad s(\alpha) = \begin{cases} \lceil \log_2(m+1) \rceil & \textit{if } \alpha \in [1, \alpha_1(m)] \\ \frac{(\alpha+1)^m - \alpha^m}{(\alpha+1)^{m-1}} & \textit{if } \alpha \in [\alpha_2(m), \infty] \end{cases}$$

Proof: This result can easily be obtained by combining theorem 5.6.2 with the fact that

$$w(\hat{C}_{\text{seq}}(n)) \leq f \cdot n$$

□

The above result shows that the machine model is ineffective when the weighting function is isotropic or almost so, and there is a high number of conditionals on every path. Fortunately, when the weighting function is skewed, conditionals do not inhibit parallelism as the next result shows.

Corollary 5.6.2 *Let $\hat{T}(n)$ be a dependenceless branching task system whose control flow graph minus the exit is a completely balanced binary tree of height n and each operation, except for the leaves of the binary tree, is a conditional. Let w be a α-skewed weighting* *function for $\hat{T}(n)$ and $\hat{C}(n) \overset{m}{\Leftrightarrow\!\rightarrow} \hat{T}(n)$ a trace scheduling schedule, that is an m-schedule where every branching instruction is restricted to be a path. If the greatest weight constraint is respected in $\mathcal{C}(n)$, that is for any two instructions $\hat{I}, \hat{I}' \in \hat{C}(n)$ where $\hat{I}'$ is reachable from $\hat{I}$, no operation in $\hat{I}$ has a weight less than the weight of $\hat{I}'$, then:*

$$\frac{w(\hat{C}(n))}{n} = \frac{(\alpha+1)^{m-1}}{(\alpha+1)^m \Leftrightarrow \alpha^m}$$

Furthermore

$$\lim_{\alpha \to \infty} \frac{(\alpha+1)^{m-1}}{(\alpha+1)^m \Leftrightarrow \alpha^m} = \frac{1}{m}$$

Proof: The first claim was proved in the proof of theorem 5.6.2 in the case where instructions where restricted to be paths. The second result is a straightforward consequence of the fact that when $\alpha \to \infty$, $(1 + 1/\alpha)^m \approx 1 + m/\alpha$. □

The results of the last two corollaries are pictured in figure 5.8. The chart gives the best possible speedup for a branch intensive program as a function of α, the skew factor of the weighting function. For α close to 1 the speedup of the general model is logarithmic, whereas the speedup of the trace scheduling model, as employed in the TRACE machines [12], is only two. However as α increases the speedup tends to m and the general model degenerates into the trace scheduling model when $\alpha \geq \alpha_2(m)$. Consequently if the applications that are targeted exhibit a skew factor close to or greater than $\alpha_2(m)$ then a simple trace scheduling model suffices for extracting parallelism. The factor $\alpha_2(m)$ increases as m, the number of resources of the machine, increases. On the other hand if applications are branch intensive and the weighting function is isotropic or no runtime information can be gathered, then additional increases in the machine parallelism will only bring logarithmic improvements in the speedup.

5.7 Approximating Optimum Performance

As section 3.5 of chapter 3 has pointed out the problem of generating optimum schedules for straight line tasks is either open when operations have the same duration, or computationally hard (NP-complete). The approach used for straight line tasks is to adopt heuristics that are within a constant factor from the optimum.

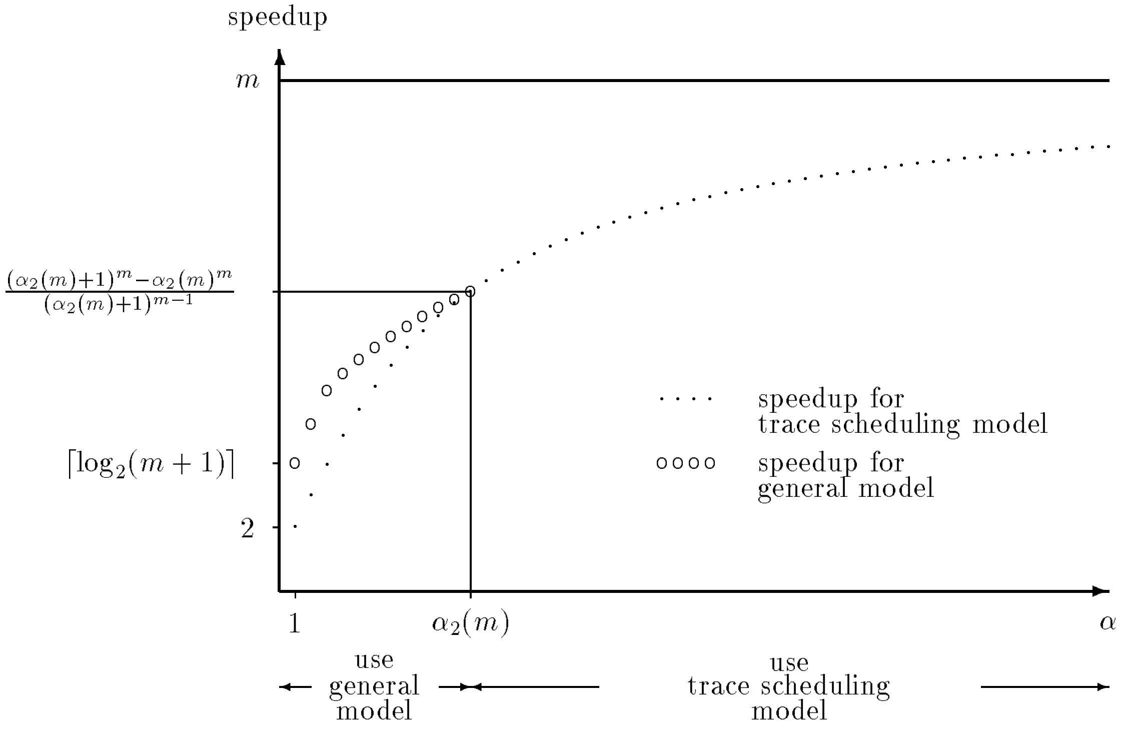


Figure 5.8: Chart showing maximum speedup as a function of α.

When conditionals are allowed simple basic block compaction, i.e. generating an independent schedule for each basic block of a branching task, can theoretically yield arbitrarily poor performance. Several studies have shown that this is also the case in practice [51,46]. The goal of this section is to bound the weighted average running time performance of a simple scheduling heuristic based solely on the weighting function and not on the structure of the underlying dependence relations. More precisely Graham's result on the performance of list scheduling algorithms for straight line tasks [10] will be extended to branching tasks. A new heuristic called greatest weight first, or GWF, generalizes the list scheduling approach to allow for conditionals. The bound on optimality guaranteed by GWF algorithms is constant if the weighting function is skewed and is logarithmic in m in the worst case.

When generating instructions for branching or straight line schedules, it is often the case that several operations are available for execution in the same cycle. In the case where such operations cannot all be executed together a selection criterion must be employed. For straight line tasks a random choice guarantees a bound of $2 \Leftrightarrow 1/m$ from the optimum. When conditionals are present the selection should not be completely random as available operations may belong to different computational paths with disparate execution weights. The obvious generalization of the random heuristic is to give priority to operations belonging to the execution paths with greatest weight. Such heuristic is termed greatest weight first or GWF.

Definition 5.7.1 *Let $\hat{T}$ be a branching task, w a weighting function for $\hat{T}$, $\hat{C} \overset{m}{\Leftrightarrow\rightarrow} \hat{T}$ a branching schedule and v a vertex in some instruction $\hat{I}$ of $\hat{C}$. The weight of v and the smallest weight of $\hat{I}$, respectively denoted $w(v)$ and $w_{\min}(\hat{I})$ are defined as*

$$w(v) = \sum_{P \in Use(v,\hat{T})} w(P) \qquad w_{\min}(\hat{I}) = \begin{cases} 0 & \text{if } |\hat{I}| < m \\ \min_{v \in \hat{I}} w(v) & \text{otherwise} \end{cases}$$

$\hat{C}$ is said to be a GWF schedule if and only if for every instruction $\hat{I} \in \hat{C}n$ and every $\hat{C}' \overset{m}{\Leftrightarrow\rightarrow} \hat{T}$ there exists no $\hat{I}' \in \hat{C}'$ which is traversed by the same execution paths as $\hat{I}$, is such that $w_{\min}(\hat{I}) < w_{\min}(\hat{I}')$ and the instructions in $\hat{C}'$ from the entry up to $\hat{I}'$ are exactly the same as those in $\hat{C}$ from the entry up to $\hat{I}$.

GWF schedules may yield arbitrarily poor performance if the duration of operations is not bounded by some small integer.

Theorem 5.7.1 *For every $m \geq 2$ there exists a branching task $\hat{T}$ where operations require at most m cycles to complete and there exists a weighting function w for $\hat{T}$ such that if $\hat{C}_{\text{opt}}$ denotes a m-optimum for $\hat{T}$ and w then every GWF schedule $\hat{C} \overset{m}{\Leftrightarrow\!\!\rightarrow} \hat{T}$ is such that*

$$m \Leftrightarrow 1 < \frac{w(\hat{C})}{w(\hat{C}_{\text{opt}})}$$

Proof: Let $\hat{T} = (O, \tau, \{\prec_P\}_{P \in Path(\hat{T})}, G)$ where

1. $O = O_1 \cup O_2$ where $O_1 = \{op_0, op_1^1, \cdots op_1^m, \cdots, op_k^1, \cdots op_k^m, cj\}$ and $O_2 = \{\bar{o}p_1, \bar{o}p_2, \cdots, \bar{o}p_{k(m-1)}\}$, for some integer k greater than 1.
2. $\forall\, op \in O_1$, $\tau(op) = 1$ and $\forall\, op \in O_2$, $\tau(op) = m$.
3. The control flow graph G is given in figure 5.9. Basic block B_1 contains operations in O_1. Operation cj is the conditional in B_1. Basic block B_2 contains operations in O_2.
4. The dependence relations $\{\prec_P\}_{P \in Path(\hat{T})}$ are as follows:

   $$\forall P \in Path(\hat{T}) \quad \forall i_1, i_2, \cdots i_k \in [1, m] \qquad op_0 \prec_P op_1^{i_1} \prec_P op_2^{i_2} \prec_P \cdots \prec_P op_k^{i_k} \prec_P cj$$

Let P_1 the execution path of $\hat{T}$ traversing B_1 but not B_2 and P_2 the execution path traversing B_2. The weighting function w is such that $w(P_1) = 1 \Leftrightarrow \epsilon$ and $w(P_2) = \epsilon$ for some real ϵ between zero and one. It is clear that for any m-optimal schedule $\hat{C}_{\text{opt}}$ for $\hat{T}$ and w one has $w(\hat{C}_{\text{opt}}) \leq (k+2) + k \cdot m \cdot \epsilon$. However, any GWF schedule for $\hat{T}$ is forced to schedule op_0 in the first instruction along with $m \Leftrightarrow 1$ operations from basic block B_2. As these operations take m cycles to execute, operations $op_1^1, \cdots, op_1^m$ have to be executed sequentially in cycles $2, \cdots, m+1$ respectively. In cycle $m+1$ no other operation op_i^j, for $i \in [2, k]$ and $j \in [1, m]$ can be executed and consequently the GWF rule requires that $m \Leftrightarrow 1$ other operations from basic block B_2 be started in cycle $m+1$. This process repeats for every set of operations $op_i^1, \cdots, op_i^m$. Thus the weighted average execution time of every GWF schedule $\hat{C} \overset{m}{\Leftrightarrow\!\!\rightarrow} \hat{T}$ is at least $k \cdot m$ cycles. Thus

$$\frac{w(\hat{C})}{w(\hat{C}_{\text{opt}})} \geq \frac{k \cdot m}{(1 + m \cdot \epsilon) \cdot k + 2} > m \Leftrightarrow 1 \quad \text{for sufficiently large } k \text{ and small } \epsilon$$

□

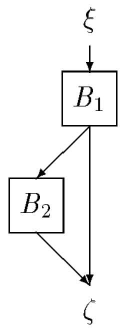


Figure 5.9: Control flow graph of $\hat{T}$.

When operations require a single execution cycle to complete, the problem illustrated in the proof of theorem 5.7.1 disappears. However, even in this event GWF schedules are not necessarily optimal. Two fundamental errors can take place when choosing operations to schedule. The first error can occur when two or more operations from the same basic block could be executing in a same slot of an instruction. In this case the GWF heuristic suffers from the same problem as list scheduling, namely the selection is random and consequently critical operations may be delayed. The second error is a generalization of the previous one to the case where tasks contain conditionals. If two operations op_1, op_2 not lying on a common path are ready for scheduling, GWF systematically selects the one with highest weight, say op_1, whereas their weights might be close and op_2 could be a critical operation for the execution paths containing it. These two errors are illustrated in the following example.

Example 5.7.1 Consider the branching task $\hat{T}$ whose control flow graph is given in figure 5.10(a). The weighting function w is assumed to be isotropic and operations to take one cycle. Let δ be some positive integer. Basic block B_1 contains a $24 \cdot \delta$ independent operations (denoted $\times$), $8 \cdot \delta$ operations (denoted $\odot$) forming a dependence chain of $8 \cdot \delta$ cycles and a conditional depending on the $\odot$ operations. Basic blocks B_2 and B_3 each contain a conditional. Basic blocks $B_4, \ldots, B_7$ each contain $8 \cdot \delta$ operations (respectively denoted $*, \star, \circ, \bullet$) each forming a dependence chain of $8 \cdot \delta$ cycles. In addition B_4 contains $56 \cdot \delta$ independent operations (denoted $\otimes$). Figure 5.10(b) shows a possible

GWF schedule $\hat{C}$ 8-admissible for $\hat{T}$. Figure 5.10(c) shows an 8-optimal schedule $\hat{C}_{\text{opt}}$ for $\hat{T}$ and the isotropic weighting function w. The ratio $w(\hat{C})/w(\hat{C}_{\text{opt}}) = (19\cdot\delta+O(1))/((8+7/4)\cdot\delta+O(1))$ which tends to 1.95 as δ grows to infinity.

Although not optimal it is possible to bound the performance of GWF schedules. Before demonstrating this result two lemmas needs to be established.

Lemma 5.7.1 *Let $(a_i)_{1\leq i\leq n}$ and $(b_i)_{1\leq i\leq n}$ two sequences of $n \geq 1$ positive numbers. Then*

$$\frac{\sum_{i=1}^{n} a_i}{\sum_{i=1}^{n} b_i} \leq \max_{i\in[1,n]} \frac{a_i}{b_i}$$

Proof: Trivially true for $n = 1$. For $n = 2$ assume without loss of generality that $a_1/b_1 < a_2/b_2$. Then one can write

$$\frac{a_1+a_2}{b_1+b_2} = \frac{a_1\cdot b_2 + a_2\cdot b_2}{(b_1+b_2)\cdot b_2} \leq \frac{(b_1+b_2)\cdot a_2}{(b_1+b_2)\cdot b_2} = \frac{a_2}{b_2}$$

and the lemma also holds for $n = 2$. Assume that the lemma is true for some k $(2 \leq k < n)$, because of the previous inequality and the inductive assumption

$$\frac{\sum_{i=1}^{k+1} a_i}{\sum_{i=1}^{k+1} b_i} \leq \max\left(\frac{\sum_{i=1}^{k} a_i}{\sum_{i=1}^{k} b_i}, \frac{a_{k+1}}{b_{k+1}}\right) \leq \max\left(\max_{i\in[1,k]} \frac{a_i}{b_i}\ ,\ \frac{a_{k+1}}{b_{k+1}}\right) = \max_{i\in[1,k+1]} \frac{a_i}{b_i}$$

and consequently the lemma also holds for $k+1$. □

Lemma 5.7.2 *Let $\mathcal{T}$ be a binary tree and ϖ a weight function which maps every edge of $\mathcal{T}$ into a non-negative real and is such that the sum of the weights of the edges sharing the same tail is 1. For a vertex x of $\mathcal{T}$ the weight of x, $w(x)$, is defined as 1 if x is $\mathcal{T}$'s root and is otherwise the product of the weights of the edges from $\mathcal{T}$'s root to x. Assume that k vertices $x_1, \cdots, x_k$ of $\mathcal{T}$ are marked. For every marked vertex x_i, $i \in [1,k]$ define $w'(x_i)$ as the maximum of the weights of the marked vertices reachable from x_i. Define the weight of marking $x_1, \cdots, x_k$ as*

$$w(x_1, \cdots, x_k) = \sum_{i=1}^{k} w(x_i) \Leftrightarrow w'(x_i)$$

then for all marking $x_1, \cdots, x_k$ one has

$$w(x_1, \cdots, x_k) \leq 1 + \frac{1}{2}\cdot\lceil\log_2 k\rceil$$

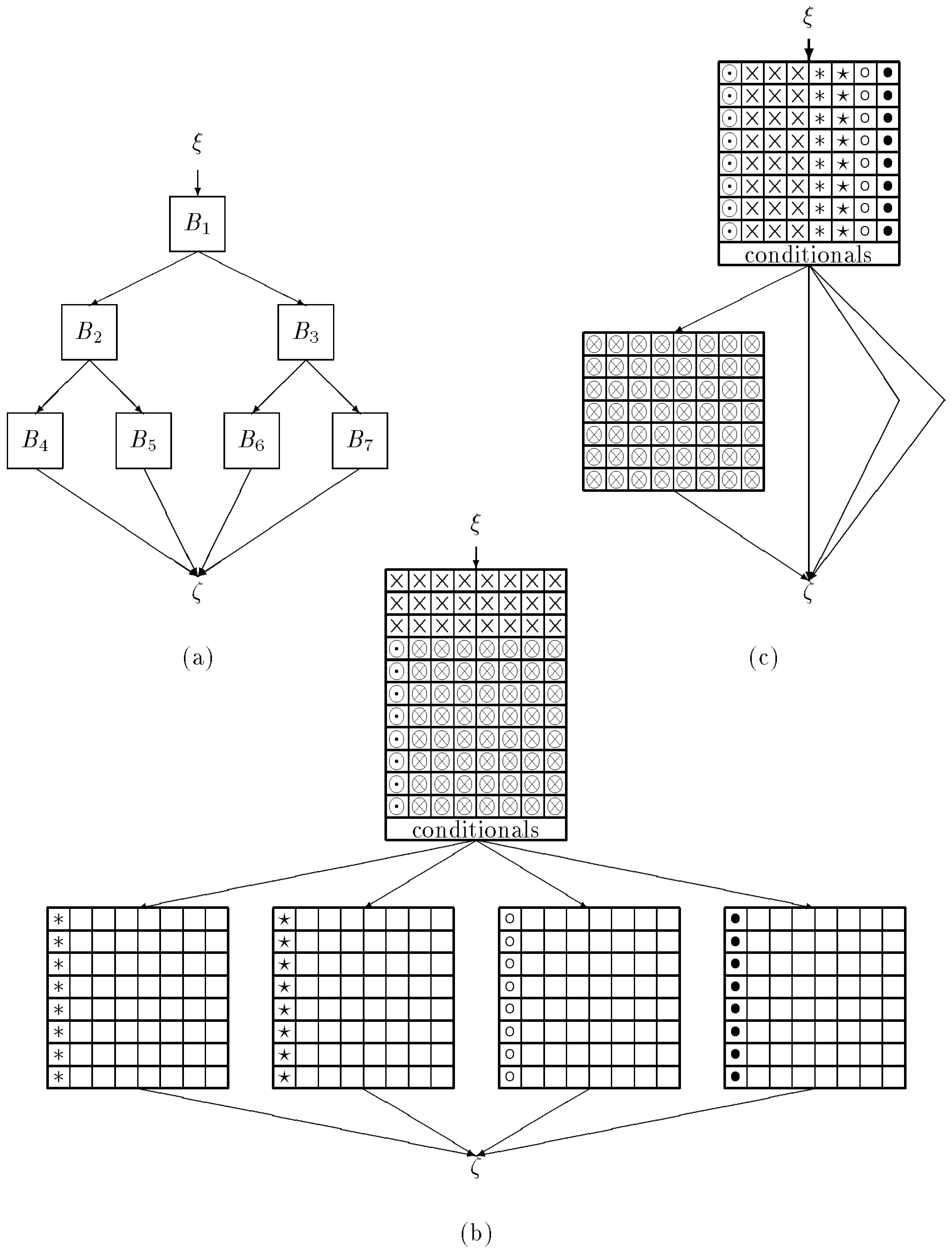


Figure 5.10: (a) Control flow graph of $\hat{T}$. (b) A GWF schedule 8-admissible for $\hat{T}$ when $\delta = 1$. (c) 8-Optimal schedule for $\hat{T}$ and the isotropic weighting function when $\delta = 1$.

Furthermore if the weighting function ϖ is alpha-skewed for some $\alpha \geq 1$ then

$$w(x_1, \cdots, x_k) \leq 1 + \frac{1}{\alpha+1} \cdot \lceil \log_2 k \rceil$$

Proof: Let $\mathcal{T} = (V, E)$ and for all internal vertex $x \in V$ assume that the left children of x has a weight greater or equal to the weight of x's right child. Let $x, y \in V$ be two marked nodes. If y is an ancestor of x and no marked node lies in the path from y to x then y is called an immediate marked ancestor of x and x an immediate marked descendent of y. A marked node x is said to be killed if its contribution in $w(x_1, \cdots, x_k)$, that is $w(x)$ is cancelled by $w'(y)$, where y is the immediate marked ancestor of x. In the case where y has two immediate marked descendents with same maximum weight it is assumed that the leftmost immediate marked descendent is killed. A marked node is said to be useful if it is not killed. Let U denote the set of useful nodes. Then

$$w(x_1, \cdots, x_k) = \sum_{x_i \in U} w(x_i)$$

If nodes and edges are added to $\mathcal{T}$ one can only increase the weight of the optimum marking. Thus it can be assumed, without any loss of generality, that $\mathcal{T}$ is a completely balanced binary tree. Consider a marking where the root of $\mathcal{T}$ and all the right children of the nodes in levels 0 through $\lceil \log_2 k \rceil$ are useful. Call such marking a k-right useful marking. It is readily seen that the maximum weight of a k-right useful marking is at most

$$1 + \frac{1}{2}\lceil \log_2 k \rceil$$

if the weighting function ϖ is arbitrary, and is exactly

$$1 + \frac{1}{\alpha+1}\lceil \log_2 k \rceil$$

if the weighting function ϖ is α-skewed.

To establish the lemma it suffices to show the existence of an injection f from the useful nodes of a marking $M = (x_1, \cdots, x_k)$ to the useful nodes of a k-right useful marking R such that

$$x_i \text{ is useful in } M \quad \Rightarrow \quad w(x_i) \leq w(f(x_i))$$

This claim will be shown by induction on h, the height of $\mathcal{T}$.

If $h = 0$, $\mathcal{T}$ is a single node and the claim is trivially true. Assume that $h > 0$. If the root r of $\mathcal{T}$ is useful in M then map the root into itself: $f(r) = r$. If no node on the path from $\mathcal{T}$'s root to its leftmost leaf, called path L, is useful in M then one can safely invoke the inductive hypothesis on each of the subtrees rooted at the right child of every node on path L. If the root of $\mathcal{T}$ is not useful in M and path L contains a node x useful in M then map x into $\mathcal{T}$'s root: $f(x) = r$. Again the inductive hypothesis can be invoked on the subtree rooted at x and the subtrees rooted at the right child of every node in the path from $\mathcal{T}$'s root down to but not including x.

The remaining case is when $\mathcal{T}$'s root is useful in M and L contains a useful node. One proceeds as follows. Let x be the useful node in L which is closest to $\mathcal{T}$'s root. Find a the closest ancestor of x which is in M. Find the node $x' \neq x$ which is reachable from a, is in M and has greatest weight among the nodes in M reachable from a. Such node x' must exist for otherwise x could not be useful in M. It is also clear that x' cannot be on the path from a to x due to the choice of a and that x' is not useful in M. Let P be the path from a to x' included. Then no node in P apart from a and x' belongs to M. Let z be the node in P closest to x' which is useful in R. Such node exists for P must take at least one right branch. Map x into z: $f(x) = z$. Note that $w(x) \leq w(x') \leq w(z)$ and that z could be x'. Before employing the inductive hypothesis one needs to modify R as follows. Every node $z' \neq z$ which is in P and is useful in R is deleted from R and is replaced by the first leftmost descendent which is not in P. Because every node deleted in R is replaced by a node with lesser weight the inductive hypothesis will not be violated. Now the inductive hypothesis can be invoked on the subtree rooted at x, the subtrees hanging off path P, the subtree rooted at x' and the subtrees rooted at the right child of every node in the path from $\mathcal{T}$'s root to x which is not also on path P. Note that these subtrees account for all of unprocessed subtrees in $\mathcal{T}$. The inductive hypothesis can be safely invoked since the root of all subtrees on which the inductive hypothesis is invoked are of three types:

1. for the subtree rooted at x, x is already mapped into a node of R
2. for the subtree rooted at x', x' is in M and is not useful in M
3. for the other subtrees their roots all belong to R and are not mapped into any

other node.

□

The result concerning the time performance of GWF schedules can now be established. Because the proof is fairly complex this result will be established only for branching task systems with memoryless dependences (see definition 5.2.2). However, the following result holds even for branching tasks with arbitrary dependences.

Theorem 5.7.2 *Let $\hat{T} = (O, \tau, G, \{\prec_P\}_{P \in Path(\hat{T})})$ be a non necessarily finite branching task with memoryless dependences, $\tau_{\max} = \max_{op \in O} \tau(op)$, w a weighting function for $\hat{T}$, $\hat{C} \overset{m}{\Leftrightarrow\!\!\rightarrow} \hat{T}$ a GWF schedule, and $\hat{C}_{\text{opt}}$ an m-optimum schedule for $\hat{T}$ and w. If $\hat{C}_{\text{opt}}$ exists, which is always the case if $\hat{T}$ is finite, then*

$$\frac{w(\hat{C})}{w(\hat{C}_{\text{opt}})} \leq \tau_{\max} \cdot \left(2 \Leftrightarrow \frac{1}{m} + \frac{m \Leftrightarrow 1}{2 \cdot m} \cdot \lceil \log_2 m \rceil\right)$$

furthermore if the weighting function w is α skewed then

$$\frac{w(\hat{C})}{w(\hat{C}_{\text{opt}})} \leq \tau_{\max} \cdot \left(2 \Leftrightarrow \frac{1}{m} + \frac{m \Leftrightarrow 1}{(\alpha + 1) \cdot m} \cdot \lceil \log_2 m \rceil\right)$$

Proof: Consider first the case where $\tau_{\max} = 1$. Let $\hat{I}$ be some instruction in $\hat{C}$ containing $k \leq m$ vertices. Then by simple algebraic manipulation one can write

$$m \cdot w(\hat{I}) = \sum_{v \in \hat{I}} w(v) + \sum_{v \in \hat{I}} w(\hat{I}) \Leftrightarrow w(v) + (m \Leftrightarrow k) \cdot w(\hat{I})$$

Because $\hat{C}$ is a GWF schedule at least one non speculative operation must be executing in $\hat{I}$. Thus there exists at least one vertex $v_0 \in \hat{I}$ such that $w(v_0) = w(\hat{I})$. This implies

$$m \cdot w(\hat{I}) \leq \sum_{v \in \hat{I}} w(v) + (m \Leftrightarrow 1) \cdot (w(\hat{I}) \Leftrightarrow w_{\min}(\hat{I}))$$

where $w_{\min}(v)$ was defined in 5.7.1. Because of property 5.5.1 and the previous result one can write

$$m \cdot w(\hat{C}) = m \cdot \sum_{\hat{I} \in \hat{C}} w(\hat{I}) \leq \sum_{\hat{I} \in \hat{C}} \sum_{v \in \hat{I}} w(v) + (m \Leftrightarrow 1) \cdot \sum_{\hat{I} \in \hat{C}} (w(\hat{I}) \Leftrightarrow w_{\min}(\hat{I}))$$

For every operation $op \in O$ let $w(op)$ denote the overall weight of the execution paths traversing op in $\hat{T}$, that is

$$w(op) = \sum_{P \in Path(op,\hat{T})} w(P)$$

Because it has been assumed that $\tau_{\max} = 1$ and because $\hat{C}$ is m-admissible for $\hat{T}$

$$\forall\, op \in O \quad \forall\, P \in Path(op, \hat{T}) \quad \exists!\, v \in \hat{C} \qquad op(v) = op \;\text{ and }\; P \in Use(v, \hat{T})$$

thus

$$\sum_{\hat{I} \in \hat{C}} \sum_{v \in \hat{I}} w(v) = \sum_{op \in O} w(op) \leq m \cdot w(\hat{C}_{\text{opt}})$$

To bound $w(\hat{C})$ in terms of $w(\hat{C}_{\text{opt}})$ it suffices to bound

$$(m \Leftrightarrow 1) \cdot \sum_{\hat{I} \in \hat{C}} (w(\hat{I}) \Leftrightarrow w_{\min}(\hat{I}))$$

Let $\hat{C}_t$ be the branching schedule constructed from $\hat{C}$ by replacing every instruction $\hat{I}$ of $\hat{C}$ whose root has $d > 1$ incoming edges, $e_1, \cdots, e_d$ by d instructions $\hat{I}_1, \cdots, \hat{I}_d$ each isomorphic to $\hat{I}$ and such that $\hat{I}_i$ has e_i as sole incoming edge. Let v be a vertex in $\hat{I}$ and v_i the vertex corresponding to v in $\hat{I}_i$, $i \in [1, d]$. If $Use(v, \hat{T})$ contains a path traversing edge e_i then $op(v_i) = op(v)$, otherwise $op(v_i) = \emptyset$. Note that if $op(v)$ is a conditional then the branching constraint of definition 5.4.1 requires that $Use(v, \hat{T}) = Path(v, \hat{C})$ and therefore for all $i \in [1, d]$, $op(v_i) = op(v)$. Clearly $\hat{C}_t \overset{m}{\Leftrightarrow\!\rightarrow} \hat{T}$, furthermore $\hat{C}_t$ is a GWF schedule whose control flow graph minus the exit is a tree. Because of the way $op(v_i)$ has been defined one has

$$w(v) = \sum_{i=1}^{d} w(v_i) \quad \Rightarrow \quad w_{\min}(\hat{I}) \geq \sum_{i=1}^{d} w_{\min}(\hat{I}_i)$$

furthermore

$$w(\hat{I}) = \sum_{i=1}^{d} w(\hat{I}_i)$$

thus

$$H = (m \Leftrightarrow 1) \cdot \sum_{\hat{I} \in \hat{C}} (w(\hat{I}) \Leftrightarrow w_{\min}(\hat{I})) \leq (m \Leftrightarrow 1) \cdot \sum_{\hat{I} \in \hat{C}_t} (w(\hat{I}_t) \Leftrightarrow w_{\min}(\hat{I}_t)) = H_t$$

To bound H_t consider the new branching task $\hat{T}_t = (O_t, \tau, G_t, , _t)$ where

1. Let vertex $v_{\hat{I}}$ denote the root of instruction $\hat{I} \in \hat{C}_t$, $V_r = \{v_{\hat{I}} \mid \hat{I} \in \hat{C}_t, \text{ and } w(\hat{I}) > w_{\min}(\hat{I})\}$ and V_b the set of vertices in $\hat{C}_t$ of out-degree two which are not in V_r. Then $O_t = V_r \cup V_b$.
2. For all $v \in O_t$, $\tau(v) = 1$.
3. The control flow graph G_t is obtained from the control flow graph of $\hat{C}_t$ by deleting every vertex not in O_t. When a vertex is deleted its immediate predecessor is set to point to its immediate successor. This step poses no difficulties since all the vertices deleted have out-degree and in-degree one.
4. The dependence relations in Γ_t are such that for all $v_{\hat{I}}, v_{\hat{I}'} \in V_r$, $v_{\hat{I}'}$ depends on $v_{\hat{I}}$ on all paths from $v_{\hat{I}}$ to $v_{\hat{I}'}$ if and only if $w_{\min}(\hat{I}) < w(\hat{I}')$.

The new branching task $\hat{T}_t$ has the same set of execution paths as $\hat{C}_t$ and hence of $\hat{T}$. Thus one can use w as a weighting function for $\hat{T}_t$. Consider the schedule $\hat{C}_o$ admissible for $\hat{T}_t$ where

1. Operations in V_b are not allowed to be scheduled concurrently with other operations in O_t.
2. Let $\hat{I}_o$ be an an instruction of $\hat{C}_o$. Two operations $v_{\hat{I}'}, v_{\hat{I}''} \in O_t$ executing in $\hat{I}_o$ are said to be unifiable if and only if $v_{\hat{I}'}$ and $v_{\hat{I}''}$ are not reachable from each other in G_t and for every operation $v_{\hat{I}} \in O_t$ executing in $\hat{I}_o$ such that $\hat{I}'$ and $\hat{I}''$ are reachable from $\hat{I}$ in G_t one has $w(\hat{I}') + w(\hat{I}'') \leq w_{\min}(\hat{I})$. Let u be the number of non unifiable operations in $\hat{I}_o$, then it is required that $u \leq m$.
3. $\hat{C}_o$ is optimal for $\hat{T}_t$ and w subject to the above two constraints.

Let $\hat{I}$ and $\hat{I}'$ be two instructions of $\hat{C}_t$ such that there exists a path in $\hat{C}_t$ first traversing $\hat{I}$ and then $\hat{I}'$. Because dependences are memoryless and because $\hat{C}_t$ is a GWF schedule

$$\forall\, v' \in \hat{I}' \quad w(v') > w_{\min}(\hat{I}) \;\Rightarrow\; \exists\, v \in \hat{I} \quad \forall\, P \in Use(v', \hat{T}) \quad op(v) \prec_P op(v')$$

Thus the dependences in $\hat{T}$ are at least as constraining, from a scheduling point of view, as the dependences in $\hat{T}_t$. Therefore if $\hat{R}$ denotes the set of instructions in $\hat{C}_o$ which do not contain operations from V_b

$$\sum_{\hat{I}_o \in \hat{R}} w(\hat{I}_o) \leq w(\hat{C}_{\text{opt}})$$

Because $\hat{C}_t$ is a GWF schedule every instruction $\hat{I} \in \hat{C}_t$ must contain a vertex v such that $w(v) = w(\hat{I})$, thus one can assume without any loss in generality that the root of $\hat{I}$, $v_{\hat{I}}$ is such vertex v. Define $w_{\min}(v_{\hat{I}}) = w_{\min}(\hat{I})$. Then

$$H_t = (m \Leftrightarrow 1) \cdot \sum_{\hat{I} \in \hat{C}_t} w(\hat{I}) \Leftrightarrow w_{\min}(\hat{I}) = (m \Leftrightarrow 1) \cdot \sum_{v_{\hat{I}} \in V_r} w(v_{\hat{I}}) \Leftrightarrow w_{\min}(v_{\hat{I}})$$

Let $\hat{I}_0 \in \hat{C}_0$. If $O_t(\hat{I}_o)$ denotes the operations of O_t executing in $\hat{I}_o$, then H_t can be rewritten as

$$H_t = (m \Leftrightarrow 1) \cdot \sum_{\hat{I}_o \in \hat{R}} \sum_{v_{\hat{I}} \in O_t(\hat{I}_o)} w(v_{\hat{I}}) \Leftrightarrow w_{\min}(v_{\hat{I}})$$

and consequently

$$\frac{H_t}{m \cdot w(\hat{C}_{\text{opt}})} \leq \frac{(m \Leftrightarrow 1) \cdot \sum_{\hat{I}_o \in \hat{R}} \sum_{v_{\hat{I}} \in O_t(\hat{I}_o)} w(v_{\hat{I}}) \Leftrightarrow w_{\min}(v_{\hat{I}})}{m \cdot \sum_{\hat{I}_o \in \hat{R}} w(\hat{I}_o)}$$

By lemma 5.7.1 one can therefore write

$$\frac{H_t}{m \cdot w(\hat{C}_{\text{opt}})} \leq \frac{m \Leftrightarrow 1}{m} \cdot \max_{\hat{I}_o \in \hat{R}} \underbrace{\sum_{v_{\hat{I}} \in O_t(\hat{I}_o)} \frac{w(v_{\hat{I}}) \Leftrightarrow w_{\min}(v_{\hat{I}})}{w(\hat{I}_o)}}_{\text{call this } X(\hat{I}_o)}$$

Because of $\hat{T}_t$'s dependences, for all $v_{\hat{I}}, v_{\hat{I}'} \in O_t(\hat{I}_o)$, $w(v_{\hat{I}'}) > w_{\min}(v_{\hat{I}})$ implies that $v_{\hat{I}'}$ is not reachable from $v_{\hat{I}}$ in $\hat{T}_t$. In addition for all $v_{\hat{I}} \in O_t(\hat{I}_o)$, $w(v_{\hat{I}}) \leq w(\hat{I}_o)$. Finally since $\hat{T}_t$'s control flow graph is a tree the weighting function w is markovian and to bound $X(\hat{I}_o)$ it suffices to give an upper bound to the solution of the following graph theoretical problem:

> **Heaviest subgraph in a tree:** One is given a binary tree $\mathcal{T}$, a positive integer m and a weight function which maps every edge in $\mathcal{T}$ into a non-negative real and is such that the sum of the weights of the edges sharing the same tail is 1. For a vertex x of $\mathcal{T}$ the weight of x, $w(x)$, is defined as 1 if x is $\mathcal{T}$'s root and is otherwise the product of the weights of the edges from $\mathcal{T}$'s root to x. Assume that certain vertices of $\mathcal{T}$ are marked. Then for every marked vertex x of $\mathcal{T}$ define $w'(x)$ as the maximum of the weights of the marked vertices which are reachable from x. The goal is to mark $k \leq m$ vertices in $\mathcal{T}$, $x_1, \cdots, x_k$ such that $w(x_1, \cdots, x_k)$, defined below, is maximum:
> $$w(x_1, \cdots, x_k) = \sum_{i=1}^{k} w(x_i) \Leftrightarrow w'(x_i)$$

Lemma 5.7.2 gives an upper bound to the solution of the above graph theoretical problem. Let U denote such upper bound. Then $X(\hat{I}_o) \leq U$ and consequently

$$H \leq H_t \leq (m \Leftrightarrow 1) \cdot U \cdot w(\mathcal{C}_{\mathrm{opt}})$$

Thus

$$m \cdot w(\hat{C}) \leq m \cdot w(\hat{C}_{\mathrm{opt}}) + (m \Leftrightarrow 1) \cdot U \cdot w(\mathcal{C}_{\mathrm{opt}})$$

Finally

$$\frac{w(\hat{C})}{w(\hat{C}_{\mathrm{opt}})} \leq 1 + \frac{m \Leftrightarrow 1}{m} \cdot U$$

which gives the desired result for the case $\tau_{\max} = 1$.

It remains to solve the case $\tau_{\max} > 1$. Consider the task system $\hat{\mathcal{T}}_{\max}$ which is identical to $\hat{\mathcal{T}}$ except for the duration of $\hat{\mathcal{T}}$'s operations which are all set to last $\tau_{\max}$ cycles in $\hat{\mathcal{T}}_{\max}$. Because in $\hat{\mathcal{T}}_{\max}$ all operations have the same duration the previously established result holds for any GWF schedule m-admissible for $\hat{\mathcal{T}}_{\max}$. If $\hat{C}_{\max}$ denotes an m-admissible GWF schedule for $\hat{\mathcal{T}}_{\max}$ and $\hat{C}^{\max}_{\mathrm{opt}}$ denotes an m-optimum schedule for $\hat{\mathcal{T}}_{\max}$ and w it is clear that

$$w(\hat{C}) \leq w(\hat{C}_{\max}) \quad \text{and} \quad w(\hat{C}^{\max}_{\mathrm{opt}}) \leq \tau_{\max} \cdot w(\hat{C}_{\mathrm{opt}})$$

from which the claim of the theorem is easily established. □

Chapter 6

Cyclic Branching Task System

This chapter combines the models of chapters 4 and 5. A cyclic branching task models the behavior of a system which needs to execute a branching task for an indeterminate number of times and formalizes the intuitive notion of a loop comprising conditionals. The scope of this chapter will be more restrained than that of chapter 4. The objective is to establish some `basic` results concerning cyclic branching task systems. As in chapter 5 all branching entities will be hatted whereas straight line entities will not.

6.1 Preliminaries

A weighted control flow graph $G = (V \cup \{\xi, \zeta\}, E)$ is a finite control flow graph where:

1. Each edge $e \in E$ is associated with some positive integer, denoted $\sigma(e)$ and termed the iteration increment of e. The sum of the iteration increments of the edges in a path P of G is called the number of iterations spanned by P and is denoted $\sigma(P)$.

2. Every cycle in G is required to span at least one iteration.

3. Let $e_s = (\xi, v_s) \in E$ be the unique edge with tail ξ, then $\sigma(e_s) = 0$.

For $a, b \in [1, \infty]$ define $G(a,b) = (V(a,b) \cup \{\xi, \zeta\}, E(a,b))$ to be the following acyclic control flow graph:

$$\begin{aligned} V(a,b) = \{v[i] \in V[a,b] \mid\ & \exists P \text{ path from } \xi \text{ to } v \text{ in } G \text{ s.t. } i = a + \sigma(P) \text{ and} \\ & \exists Q \text{ path from } v \text{ to } \zeta \text{ in } G \text{ s.t. } i + \sigma(Q) \leq b \qquad \} \end{aligned}$$

$$\begin{aligned} E(a,b) = \ & \{(\xi, v_s[a])\} \cup \{(u[i], \zeta) \mid u[i] \in V(a,b),\ \ (u,\zeta) \in E\} \ \ \cup \\ & \{(u[i], v[j]) \mid u[i], v[j] \in V(a,b),\ \ e = (u,v) \in E,\ \ j = i + \sigma(e)\} \end{aligned}$$

Let R be a path in G from some vertex v_1 to some vertex v_2. For $i \geq 1$ if $v_1[i]$ and $v_2[i + \sigma(R)]$ belong to $V(a, b)$ one defines $R(i)$ to be the unique path in $G(a, b)$ which goes from $v_1[i]$ to $v_2[i + \sigma(R)]$ and is such that for any $k \geq 1$, $v[j]$ is the k-th vertex traversed by $R(i)$ in $G(a, b)$ if and only if v is the k-th vertex traversed by R in G.

A set $\hat{\mathcal{I}}$ of branching m-instructions is said to cover the weighted control flow graph G if and only if $\hat{\mathcal{I}}$ is an m-instruction covering of G without consideration of the iteration increments and every edge belonging to a branching instruction in $\hat{\mathcal{I}}$ has null iteration increment in G. Let $\hat{I} = (V_{\hat{I}}, E_{\hat{I}}) \in \hat{\mathcal{I}}$. For $i \geq 1$ one defines $\hat{I}(i) = (V_{\hat{I}}(i), E_{\hat{I}}(i))$ to be the following branching instruction:

$$\begin{aligned} V_{\hat{I}}(i) &= V_{\hat{I}}[i, i] \cap V(a, b) \\ E_{\hat{I}}(i) &= \{(u[i], v[i]) \mid (u, v) \in E_{\hat{I}} \text{ and } (u[i], v[i]) \in E(a, b)\} \end{aligned}$$

Furthermore if $op(v) = op[k]$ then for $v[i] \in V_{\hat{I}}(i)$, $op(v[i]) = op[k + i \Leftrightarrow 1]$ if $k + i > 1$ and $op(v[i])$ is the null operation otherwise. For $a, b \in [1, \infty]$

$$\hat{\mathcal{I}}(a, b) = \{\hat{I}(i) \mid \hat{I} \in \hat{\mathcal{I}} \text{ and } V_{\hat{I}} \neq \emptyset\}$$

Note that $\hat{\mathcal{I}}$ covers the weighted control flow graph G if and only if $\hat{\mathcal{I}}(a, b)$ is an instruction covering of $G(a, b)$.

Let H be a di-graph and let $\sqsubset$ denote the sub-path relation. A set $\mathcal{P}$ of paths in H is said to be redundant if and only if there exist two different paths $P, Q \in \mathcal{P}$ such that $P \sqsubset Q$. The $\sqsubset$-minimum of $\mathcal{P}$, denoted $\min_{\sqsubset}$, is defined to be the non redundant subset of $\mathcal{P}$ such that for any $P \in \min_{\sqsubset}\mathcal{P}$, there exists a path $Q \in \mathcal{P}$ such that $P \sqsubset Q$.

6.2 Definition

Informally a cyclic branching system $\hat{\mathcal{L}}$ can be seen as an infinite sequence of branching tasks called iterations. Iterations share the same operation set O, duration function τ and control flow graph G but may have different dependence relations. In addition operations from an iteration may depend on operations in preceding iterations.

Definition 6.2.1 *A cyclic branching task system $\hat{\mathcal{L}}$ (or more briefly cyclic branching system) is an infinite branching task $\hat{\mathcal{L}} = (O[1, \infty], \tau, G(1, \infty), ,)$ such that:*

1. *O is a finite set called the core operation set of $\hat{\mathcal{L}}$. The operation set of $\hat{\mathcal{L}}$ is $O[1,\infty]$. For $op \in O$, $op[i]$ denotes operation op executed during iteration i. The integer i is termed the iteration index of $op[i]$.*
2. *For all $op \in O$ and $i,j \in [1,\infty]$, $\tau(op[i]) = \tau(op[j])$. In the sequel such unique number is denoted $\tau(op)$.*
3. *$G = (O \cup \{\xi,\zeta\}, E)$ is a weighted control flow graph such that $(O, E \cap O \times O)$ is strongly connected and the iteration increment of each edge is either 0 or 1. Let op_s denotes the immediate successor of ξ. One further requires that op_s be the head of all the edges with unitary iteration increment. The graph G is called the core control flow graph of $\hat{\mathcal{L}}$. The control flow graph of $\hat{\mathcal{L}}$ is $G(1,\infty)$. Note that $O(1,\infty) = O[1,\infty]$.*
4. *$, = \{\prec_P\}_{P \in Path(\hat{\mathcal{L}})}$ is the usual set of dependence relations of $\hat{\mathcal{L}}$.*

For $n \geq 1$ the function τ and the partial orders in $,$ can trivially be restricted to $O(1,n)$ and $G(1,n)$. The branching task $\hat{\mathcal{L}}(n) = (O(1,n), G(1,n), \tau, ,)$ is termed the n-instance of $\hat{\mathcal{L}}$.

Unlike cyclic straight line systems, the n-instance of $\hat{\mathcal{L}}$ formalizes the case where $\hat{\mathcal{L}}$ is required to iterate at most n times in order to complete.

As in chapter 4 it is important, for the purpose of generating periodic schedules, to identify the most stringent dependences between any two operations in O. In extending the notion of dependence distance introduced in definition 4.3.1 one needs to be attentive to the fact that dependences need to be expressed in terms of paths rather than simple numbers.

Definition 6.2.2 *Let $\hat{\mathcal{L}}$ be some cyclic branching system with core operation set O and core control flow graph G. For all $op, op' \in O$ the set of dependence paths from op to op', denoted $\hat{d}(op, op')$, is defined as:*

$$\hat{d}(op,op') = \min\nolimits_{\sqsubset}\{R \text{ path of } G \;\mid\; \begin{array}{l} \forall\, i \geq 1 \;\exists\, k \geq i \;\exists\, P \text{ path of } G(1,\infty) \\ R(k) \sqsubset P \text{ and } op[k] \prec_P op'[k+\sigma(R)]\} \end{array}$$

Furthermore if

$$\forall\, op, op' \in O \quad \forall\, i \geq 1 \quad \forall\, R \in \hat{d}(op,op') \quad \forall\, P \text{ path of } G(1,\infty)$$

$$R(i) \sqsubset P \;\Rightarrow\; op[i] \prec op'[i+\sigma(R)]$$

one says that $\hat{\mathcal{L}}$ is recurrent or has recurrent dependences.

As in chapter 4 the idea behind $\hat{d}(op, op')$ is to disregard dependences that do not repeat infinitely many times, as these do not pertain to the repetitive nature of cyclic systems. Note that the dependences of recurrent cyclic branching systems are memoryless. The definition of memoryless dependences was given in 5.2.2.

6.3 Infinite and Periodic Schedules

Like in the straight line case, real-life branching schedulers will mostly generate higly structured and regular schedules. As in chapter 4 infinite schedules model the behavior of dynamic schedulers and as in chapter 4 periodicity is grafted onto infinite schedules to model static schedulers. This last notion is extended as follows.

Definition 6.3.1 *Let $\hat{\mathcal{L}}$ be some cyclic branching system and $\hat{\mathcal{C}} = (G^\infty, \hat{\mathcal{I}}^\infty)$ an m-admissible infinite branching schedule for $\hat{\mathcal{L}}$. $\hat{\mathcal{C}}$ is said to be periodic if and only if there exists a weighted control flow graph $\hat{B}$ and an m-instructions covering $\hat{\mathcal{J}}$ of $\hat{B}$ such that $G^\infty = \hat{B}(1,\infty)$ and $\hat{\mathcal{I}}^\infty = \hat{\mathcal{J}}(1,\infty)$. The graph $\hat{B}$ is called the body of $\hat{\mathcal{C}}$ and the maximum number of iterations spanned by any cycle of $\hat{B}$ is called the unfolding of $\hat{\mathcal{C}}$.*

6.4 Asymptotic Performance

In chapter 5 the weighted average running time of a branching schedule was introduced. If the schedule is infinite this measure of time performance may be undefined. In order to guarantee convergence and because for cyclic systems one is more interested in asymptotic behavior, the performance criteria which will be this chapter focus is a combination of those defined in 4.6.1 and 5.5.3.

Definition 6.4.1 *Let $\hat{\mathcal{L}}$ be a cyclic branching system, w a weighting function for $\hat{\mathcal{L}}$ and $\hat{\mathcal{C}} \overset{m}{\hookleftarrow\!\!\rightarrow} \hat{T}$ an infinite branching schedule. The asymptotic weighted average running time of $\hat{\mathcal{C}}$ is defined as*

$$\overline{w}(\hat{\mathcal{C}}) = \sum_{P \in Path(\hat{\mathcal{L}})} w(P) \cdot \frac{|\hat{\mathcal{C}}(P)|}{\sigma(P)}$$

$\hat{\mathcal{C}}$ is said to be asymptotic m-optimum for $\hat{\mathcal{L}}$ and w if and only if there exists no $\hat{\mathcal{C}}' \overset{m}{\looparrowleft\!\!\!\rightarrow} \hat{\mathcal{L}}$ such that $\overline{w}(\hat{\mathcal{C}}') < \overline{w}(\hat{\mathcal{C}})$.

Note that $\overline{w}(\hat{\mathcal{C}})$ is not always defined. This may for instance be the case if $\hat{\mathcal{C}}$ contains an unbounded number of empty branching instructions. A branching instruction $\hat{I}$ is said to be empty if for every vertex $v \in \hat{I}$, $Use(v, \hat{\mathcal{L}}) = \emptyset$. Clearly for $\overline{w}(\hat{\mathcal{C}})$ to be defined it is sufficient, but not necessary, that there exist a k such that

$$\forall\, P \in Path(\hat{\mathcal{L}}) \qquad |\hat{\mathcal{C}}(P)| \leq k \cdot \sigma(P)$$

In practice typical schedules for cyclic branching systems satisfy such condition. This is true in particular if every instruction contains a non speculative operation.

The asymptotic time performance measure induces a total order on the m-admissible schedules of a cyclic branching system $\hat{\mathcal{L}}$. Because the set $\{\overline{w}(\hat{\mathcal{C}}) | \hat{\mathcal{C}} \overset{m}{\looparrowleft\!\!\!\rightarrow} \hat{\mathcal{L}}$ and $\overline{w}(\hat{\mathcal{C}}) < \infty\}$ is lower bounded by 0 it must have a greatest lower bound. If $\hat{\mathcal{L}}$ has an asymptotic m-optimum for w, its asymptotic performance must equal such lower bound. However there is no guarantee that an asymptotic m-optimum for $\hat{\mathcal{L}}$ and w exists. Indeed for $m = \infty$ such optimum need not exist. This is quite obvious if there are no dependences from one iteration to the next and consequently an unlimited number of $\hat{\mathcal{L}}$ iterations can be executed concurrently. However, even under the restrictive condition that for all operation op in $\hat{\mathcal{L}}$ and for every $i \geq 1$ there be a dependence chain from $op[i]$ to $op[i+1]$ one can show that cyclic branching systems with no asymptotic ∞-optimum for $\hat{\mathcal{L}}$ and w exist [53]. This solves an open problem posed by several researchers [4,59].

Theorem 6.4.1 *There exists a cyclic branching system $\hat{\mathcal{L}}$ with recurrent dependences, core operation set O and such that*

$$\forall\, op \in O \quad \hat{d}(op, op) \neq \emptyset \;\; \textit{and} \;\; \forall\, R \in \hat{d}(op, op) \quad \sigma(R) = 1$$

for which no asymptotic ∞-optimum exists for $\hat{\mathcal{L}}$ and any weighting function w mapping only finitely many execution paths of $\hat{\mathcal{L}}$ into zero.

Proof: Because resources are unbounded an infinite schedule $\hat{\mathcal{C}}$ admissible for $\hat{\mathcal{L}}$ is asymptotically ∞-optimum for w if and only if for all $P \in Path(\hat{\mathcal{L}})$, $w(P) > 0$ implies that

$|\hat{\mathcal{C}}(P)|$ be equal to the length of the longest dependence chain in the straight line task $\hat{\mathcal{L}}(P)$, that is the restriction of $\hat{\mathcal{L}}$ to P (see definition 5.2.1).

Let $\hat{\mathcal{L}} = (O[1,\infty], \tau, G(1,\infty), ,)$ be the cyclic branching system where

1. $O = \{cj_1, cj_2, op_1, op_2, op_3, op_3, op_4, op_5, op_6\}$.
2. $\forall\, op \in O\ \ \tau(op) = 1$.
3. G, the core control flow graph of $\hat{\mathcal{L}}$ is given in figure 6.1(a).
4. $\hat{\mathcal{L}}$ has recurrent dependences. The dependence relations are exactly those implied by the following sets of dependence paths and portrayed in figure 6.1(b).

$$\begin{array}{ll}
\hat{d}(op_1, op_2) = \{e_{12}\} & \hat{d}(op_2, op_1) = \{P_{21}\} \\
\hat{d}(op_1, op_1) = \{e_{12}P_{21}\} & \hat{d}(op_2, op_2) = \{P_{21}e_{12}\} \\
\hat{d}(op_4, op_6) = \{e_{46}\} & \hat{d}(op_6, op_4) = \{P_{64}\} \\
\hat{d}(op_5, op_6) = \{e_{56}\} & \hat{d}(op_6, op_5) = \{P_{65}\} \\
\hat{d}(op_4, op_4) = \{e_{46}P_{64}\} & \hat{d}(op_5, op_5) = \{e_{56}P_{65}\} \\
\hat{d}(op_6, op_6) = \hat{d}(op_4, op_4) \cup \hat{d}(op_5, op_5) & \\
\hat{d}(op_2, op_3) = \{e_{23}\} & \\
\hat{d}(op_6, op_3) = \{e_{63}\} & \\
\hat{d}(op_3, op_3) = \text{all simple cycles of } G & \\
\hat{d}(op_3, cj_1) = \{e_{31}\} & \\
\hat{d}(op_3, cj_2) = \{P_{32}\} &
\end{array}$$

All other sets of dependence paths are empty.

Consider an execution path $P \in Path(\hat{\mathcal{L}})$ such that $\sigma(P) = 2 \cdot n$, for some even integer $n \geq 2$ and P traverses op_1 for the first n iterations and cj_2 for the remaining n iterations. Let $\hat{\mathcal{C}}$ be an admissible branching schedule for $\hat{\mathcal{L}}$. Because n is arbitrary and only finitely many execution paths are mapped into zero by w, $\hat{\mathcal{C}}$ cannot be asymptotically ∞-optimum for w unless

$$|\hat{\mathcal{C}}(P)| = 2 \cdot n + 1 + n = 3 \cdot n + 1$$

In fact all the longest dependence chains in $\hat{\mathcal{L}}(P)$ are of length $3 \cdot n + 1$. For instance there is a dependence chain of $2 \cdot n$ cycles between op_1 and op_2 in the first n iterations, a dependence chain of n cycles from op_3 to itself in the remaining n iterations and the dependence from op_2 to op_3 in iteration n. Any copy of operation $op_3[2 \cdot n]$ has to be executed in cycle $3 \cdot n + 1$ in order to guarantee that $|\hat{\mathcal{C}}(P)| = 3 \cdot n + 1$. Because $op_3[2 \cdot n]$ depends on $op_6[2 \cdot n]$, and because of the $2 \cdot n$ cycles dependence chains involving op_4 & op_6 and op_5 & op_6 in iterations n through $2 \cdot n$, copies of operation $op_6[n + n/2]$ must start executing in cycle $2 \cdot n$ at the latest. Because of the dependence from op_3 to cj_2, conditional $cj_2[n+n/2]$ cannot be executed, and hence resolved, before cycle $2{\cdot}n{+}1{+}n/2$. This creates a gap of at least $n/2 + 1$ cycles between the point where $op_6[n + n/2]$ has to be executed, in order to ensure that $|\hat{\mathcal{C}}(P)| = 3{\cdot}n{+}1$, and the point where the exact path from which $op_6[n{+}n/2]$ is reached is known. One must therefore execute at least $2^{\lfloor \frac{n/2+1}{2} \rfloor}$ copies of operation $op_6[n + n/2]$. As only $2 \cdot n$ branching instructions are available for the first n iterations, the number of operations executed in some instruction must increase with n. Since in any branching schedule the number of operations per instruction is required to be finite, no admissible schedule $\hat{\mathcal{C}}$ for $\hat{\mathcal{L}}$ can be asymptotically ∞-optimum for w. $\square$

In the remainder of this chapter it will be assumed that $m < \infty$. If $m < \infty$ one can show that an asymptotic m-optimum always exists.

Theorem 6.4.2 *If $m < \infty$ then for any cyclic branching system $\hat{\mathcal{L}}$ and any weighting function w for $\hat{\mathcal{L}}$ the asymptotic m-optimum for $\hat{\mathcal{L}}$ and w exists.*

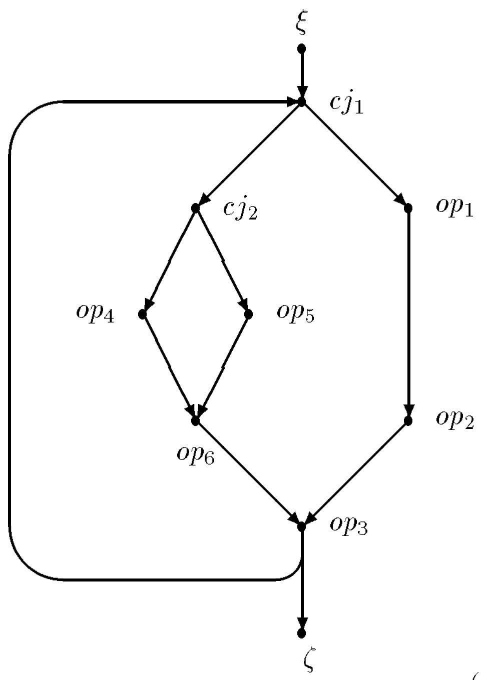
Proof: Recall that for $n \geq 1$, $\hat{\mathcal{L}}(n)$ denotes the n-instance of $\hat{\mathcal{L}}$ (see definition 6.2.1). Given a branching schedule $\hat{\mathcal{C}} \overset{m}{\Leftrightarrow} \hat{\mathcal{L}}$, $\hat{\mathcal{C}}(n)$ denotes the branching schedule extracted from $\hat{\mathcal{C}}$ which is m-admissible for $\hat{\mathcal{L}}(n)$. More formally if $\hat{\mathcal{C}} = (G^\infty, \hat{\mathcal{I}}^\infty)$ then $\hat{\mathcal{C}}(n) = (G_n, \hat{\mathcal{I}}_n)$ is the m-admissible schedule for $\hat{\mathcal{L}}(n)$ where G_n is a subgraph of G^∞ and $\hat{\mathcal{I}}_n \subset \hat{\mathcal{I}}^\infty$.

Let Φ_0 be the class of m-admissible branching schedules for $\hat{\mathcal{L}}$. For $n \geq 1$ one inductively defines the class $\Phi_n \subseteq \Phi_{n-1}$. Given any two $\hat{\mathcal{C}}, \hat{\mathcal{C}}' \in \Phi_{n-1}$ define the following relation $\equiv_n$:

$$\hat{\mathcal{C}} \equiv_n \hat{\mathcal{C}}' \quad \Leftrightarrow \quad \hat{\mathcal{C}}(n) = \hat{\mathcal{C}}'(n)$$

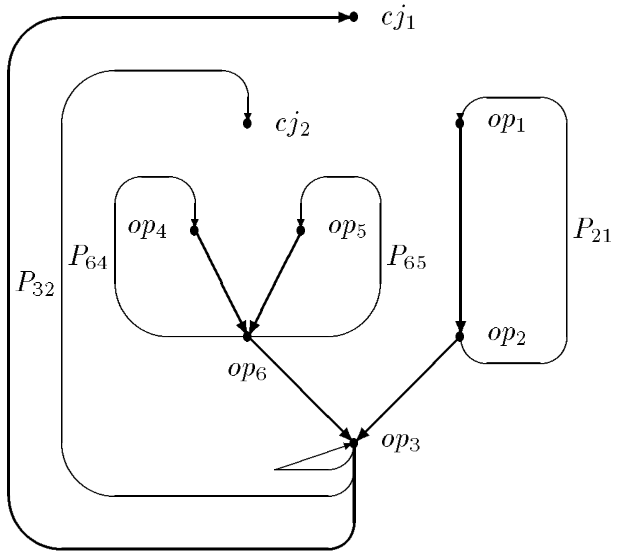
The relation $\equiv_n$ is an equivalence relation on the schedules in Φ_{n-1}. Define the following partial order $<\!\cdot$ on the equivalence classes of $\equiv_n$. Given Ψ and Ψ' two such equivalence classes

$$\Psi <\!\cdot \Psi' \quad \Leftrightarrow \quad \forall \hat{\mathcal{C}}' \in \Psi' \;\; \exists \hat{\mathcal{C}} \in \Psi \quad \overline{w}(\hat{\mathcal{C}}) < \overline{w}(\hat{\mathcal{C}}')$$



$e'_{11} = (cj_1, op_1)$ $e_{12} = (op_1, op_2)$
$e_{23} = (op_2, op_3)$ $e'_{31} = (op_3, cj_1)$
$e'_{12} = (cj_1, cj_2)$ $e'_{24} = (cj_2, op_4)$
$e'_{25} = (cj_2, op_5)$ $e_{46} = (op_4, op_6)$
$e_{56} = (op_5, op_6)$ $e_{63} = (op_6, op_3)$

(a)



$P_{21} = (e_{23}, e'_{31}, e'_{11})$
$P_{64} = (e_{63}, e'_{31}, e'_{12}, e_{24})$
$P_{65} = (e_{63}, e'_{31}, e'_{12}, e_{25})$
$P_{32} = (e'_{31}, e'_{12})$

(b)

Figure 6.1: (a) Core control flow graph of $\hat{\mathcal{L}}$. (b) Graph representing the dependence paths between the operations in $\hat{\mathcal{L}}$. Thin edges represent paths of more than one edge.

Because $m < \infty$ there are only finitely many equivalence classes for $\equiv_n$. Thus there exists a class $\Phi_n \subseteq \Phi_{n-1}$ such that for all other equivalence class Ψ of $\equiv_n$

$$\Psi \not\lessdot \Phi_n$$

that is

$$\exists\, \hat{\mathcal{C}} \in \Phi_n \quad \forall\, \hat{\mathcal{C}}' \in \Psi \qquad w(\hat{\mathcal{C}}) \leq w(\hat{\mathcal{C}}')$$

Thus it is possible to construct a sequence $(\Phi_n)_{n\geq 0}$ of sets of branching schedules m-admissible for $\hat{\mathcal{L}}$ such that:

1. $\forall\, n \geq 1 \quad \Phi_n \subseteq \Phi_{n-1}$.
2. $\forall\, i \geq 0 \quad \forall\, j \geq i \quad \forall\, \hat{\mathcal{C}} \in \Phi_i \quad \forall\, \hat{\mathcal{C}}' \in \Phi_j \qquad \hat{\mathcal{C}}(i) = \hat{\mathcal{C}}'(i)$
   Informally the schedule comprising only the first i iterations of $\hat{\mathcal{C}} \in \Phi_i$ is a sub-schedule of schedule $\hat{\mathcal{C}}' \in \Phi_j$, for $j \geq i$.
3. $\forall\, i \geq 0 \quad \exists\, \hat{\mathcal{C}} \in \Phi_i \quad \forall\, \hat{\mathcal{C}}' \xleftrightarrow{m} \hat{\mathcal{L}} \quad \hat{\mathcal{C}}' \notin \Phi_i \;\Rightarrow\; w(\hat{\mathcal{C}}) \leq w(\hat{\mathcal{C}}')$.

The sequence $(\Phi_n)_{n\geq 0}$ is lower bounded by the empty set, thus because of property 1 above

$$\lim_{n\to\infty} \Phi_n = \Phi_\infty \qquad \text{exists}$$

Furthermore, because of property 2 above, Φ_∞ must contain exactly one schedule, $\hat{\mathcal{C}}_{\mathrm{opt}}$. Finally the last of the above properties guarantees that $\hat{\mathcal{C}}_{\mathrm{opt}}$ is asymptotically m-optimum for $\hat{\mathcal{L}}$ and w. $\square$

6.5 Approximating Optimal Asymptotic Performance

It is natural to wonder whether for cyclic branching systems there exists a heuristic which is as simple as the GWF rule of the previous chapter and guarantees the same performance bound for $\overline{w}$ as GWF does for w.

Theorem 6.5.1 *Let $\hat{\mathcal{L}}$ be a cyclic branching system, $\tau_{\max}$ the longest duration of an operation in $\hat{\mathcal{L}}$, w a weighting function for $\hat{\mathcal{L}}$ and $\hat{\mathcal{C}}_{\mathrm{opt}} \xleftrightarrow{m} \hat{\mathcal{L}}$ an asymptotic m-optimum for w. Let w' be the weighting function for $\hat{\mathcal{L}}$ such that for $P \in Path(\hat{\mathcal{L}})$:*

$$w'(P) = \frac{w(P)}{\lambda \cdot \sigma(P)} \quad \textit{where} \quad \lambda = \sum_{P \in Path(P)} \frac{w(P)}{\sigma(P)}$$

Then for every branching schedule $\hat{\mathcal{C}} \overset{m}{\Leftrightarrow} \hat{\mathcal{L}}$ if $\hat{\mathcal{C}}$ is GWF with respect to w' then

$$\frac{\overline{w}(\hat{\mathcal{C}})}{\overline{w}(\hat{\mathcal{C}}_{\text{opt}})} \leq \tau_{\max} \cdot \left(2 \Leftrightarrow \frac{1}{m} + \frac{m \Leftrightarrow 1}{2 \cdot m} \cdot \lceil \log_2 m \rceil\right)$$

furthermore if the weighting function w is α skewed then

$$\frac{\overline{w}(\hat{\mathcal{C}})}{\overline{w}(\hat{\mathcal{C}}_{\text{opt}})} \leq \tau_{\max} \cdot \left(2 \Leftrightarrow \frac{1}{m} + \frac{m \Leftrightarrow 1}{(\alpha + 1) \cdot m} \cdot \lceil \log_2 m \rceil\right)$$

Proof: The proof of theorem 5.7.2 in chapter 4 was established for finite as well as infinite tasks as long as the m-optimum for the weighting function employed was defined. Because theorem 6.4.2 has shown that such is the case if w' is the weighting function employed and because the asymptotic m-optimum for w is the same as the asymptotic m-optimum for w' the above claim holds. □

As in chapter 4 the goal of a static scheduler is to approximate optimum performance with periodic schedules. Theorem 4.6.3 clearly indicates that for arbitrary dependences the asymptotic performance of all periodic schedules could be very poor. However, like for the straight line case, if dependences are recurrent there exist periodic branching schedules that are arbitrarily close to the asymptotic optimum.

Theorem 6.5.2 *Let $\hat{\mathcal{L}}$ be a cyclic branching system with recurrent dependences, w an arbitrary weighting function and $\hat{\mathcal{C}}_{\text{opt}}$ an asymptotic m-optimum for w. Then for every arbitrarily small ϵ there exists an infinite branching schedule $\hat{\mathcal{C}} \overset{m}{\Leftrightarrow} \hat{\mathcal{L}}$ which is periodic and is such that:*

$$\overline{w}(\hat{\mathcal{C}}, \hat{\mathcal{L}}) \Leftrightarrow \overline{w}(\hat{\mathcal{C}}_{\text{opt}}, \hat{\mathcal{L}}) < \epsilon$$

Proof: Let $\hat{B}_n$ be a branching schedule which is m-admissible for $\hat{\mathcal{L}}(n)$ and is such that for all $\hat{C} \overset{m}{\Leftrightarrow} \hat{\mathcal{L}}(n)$:

$$\sum_{\substack{P \in Path(\hat{\mathcal{L}}) \\ \sigma(P) \leq n}} w(P) \cdot \frac{|\hat{B}_n(P)|}{\sigma(P)} \leq \sum_{\substack{P \in Path(\hat{\mathcal{L}}) \\ \sigma(P) \leq n}} w(P) \cdot \frac{|\hat{C}(P)|}{\sigma(P)}$$

Such schedule $\hat{B}_n$ must exist because there are only finitely many m-admissible schedules for $\hat{\mathcal{L}}(n)$. Let $\hat{\mathcal{C}}_n$ be the periodic schedule with body $\hat{B}_n$. Because $\hat{\mathcal{L}}$'s dependences are periodic, $\hat{\mathcal{C}}_n \overset{m}{\Leftrightarrow} \hat{\mathcal{L}}$. Furthermore one can write

$$\begin{aligned} \overline{w}(\hat{\mathcal{C}}_n) \Leftrightarrow \overline{w}(\hat{\mathcal{C}}_{\text{opt}}) \quad = & \sum_{\substack{P \in Path(\hat{\mathcal{L}}) \\ \sigma(P) \leq n}} w(P) \cdot \frac{|\hat{B}_n(P)| \Leftrightarrow |\hat{\mathcal{C}}_{\text{opt}}(P)|}{\sigma(P)} \quad + \\ & \sum_{\substack{P \in Path(\hat{\mathcal{L}}) \\ \sigma(P) > n}} w(P) \cdot \frac{|\hat{\mathcal{C}}_n(P)| \Leftrightarrow |\hat{\mathcal{C}}_{\text{opt}}(P)|}{\sigma(P)} \end{aligned}$$

because of the choice of $\hat{B}_n$

$$\overline{w}(\hat{\mathcal{C}}_n) \Leftrightarrow \overline{w}(\hat{\mathcal{C}}_{\text{opt}}) \leq \sum_{\substack{P \in Path(\hat{\mathcal{L}}) \\ \sigma(P) > n}} w(P) \cdot \frac{|\hat{\mathcal{C}}_n(P)|}{\sigma(P)}$$

Let O the core operation set of $\hat{\mathcal{L}}$ and $\tau_{\max} = \max_{op \in O} \tau(op)$. Then

$$\forall\, P \in Path(\hat{\mathcal{L}}) \qquad |\hat{\mathcal{C}}_n(P)| \leq |O| \cdot \tau_{\max} \cdot \sigma(P)$$

Thus

$$\sum_{\substack{P \in Path(\hat{\mathcal{L}}) \\ \sigma(P) > n}} w(P) \cdot \frac{|\hat{\mathcal{C}}_n(P)|}{\sigma(P)} \leq |O| \cdot \tau_{\max} \cdot \sum_{\substack{P \in Path(\hat{\mathcal{L}}) \\ \sigma(P) > n}} w(P)$$

since

$$\lim_{n \to \infty} \sum_{\substack{P \in Path(\hat{\mathcal{L}}) \\ \sigma(P) > n}} w(P) = 0$$

for every $\epsilon > 0$ it suffices to select a branching schedule $\hat{B}_n$ for a sufficiently large n. □

Note that in the previous proof the body of the periodic schedule can contain an arbitrarily large number of iterations and consequently its unfolding can be enormous.

Chapter 7

Conclusion

The aim of the dissertation was the identification of theoretical advantages and bottlenecks of the VLIW architectural paradigm. In this respect several results have been established.

The performance of idealized dynamic and static schedulers have been compared. The schedulers are idealized in that both have access to the same set of operation dependences and both have infinite scheduling lookahead. Because of the idealistic assumptions, static and dynamic schedulers differ only in the presence of loops: Statically generated loop schedules are required to be periodic whereas dynamic ones are not.

In this framework it is shown that close to optimum dynamic schedules always exist. Furthermore such schedules can be generated by employing a simple heuristic. On the contrary if operation dependences in the original loop are irregular all static schedules can yield poor performance. More precisely let m denote the number of available processors. Then sequential loops have been exhibited for which all static schedules are approximately m times slower than the optimum. If loop dependences are regular, however, there always exist static schedules which are arbitrarily close to the optimum and the same heuristic used dynamically can be employed to generate static schedules with close to optimum performance.

In the case of straight line loops with regular dependences (sr-loops) it was shown that optimum schedules could be generated statically if all loop operations are interdependent. In an attempt to characterize sr-loops for which optimum schedules can be generated statically it was shown that this occurs if and only if an optimum schedule exists where the maximum number of execution cycles separating two operations of the same iteration is bounded. The general problem of whether optimum schedules for sr-loops can be generated statically remains open. When loops are not straight line an example was exhibited for which no optimum static schedules on unlimited resource VLIWs exist.

In the case of sr-loops an efficient loop parallelization algorithm has also been presented. The algorithm generates parallel loops with the same number of operations as the original loop and guarantees a worst case performance of roughly twice the optimum. The algorithm is based on a novel technique that deletes certain critical edges from the loop's dependence graph. Because dependence graphs have usually a number of edges linear in the number of operations, the algorithm runs in quadratic time on the average and in cubic time in the worst case.

Some loop parallelization algorithms developed for VLIWs assume that dependences between operations span at most one iteration. If this property does not hold, unrolling is invoked on the input loop to diminish dependence distances between operations. A side result has been to show that unrolling does not systematically guarantee smaller dependence distances. Consequently loop parallelizing algorithms must explicitly account for all type of dependences unless some degree of available parallelism is sacrificed.

Because static schedules are created at compile time all possible execution scenarios have to be generated. This opens the possibility for a time/space tradeoff. Indeed even to achieve modest speedups some sequential applications where shown to require exponential increases in code size.

Parallelism in sequential applications is clearly limited by operation dependences. It is not immediate, however, that conditionals may equally inhibit parallelism. In fact it was proved that only logarithmic speedups can be achieved when path probabilities are slightly skewed and conditionals are abundant. This result holds for both static and dynamic schedulers. If path probabilities are skewed then better speedups are possible. As the skew factor increases the general branching model degenerates into trace scheduling. Thus, depending on the applications envisioned, a complex multi-branches design may be unjustified.

When generating instructions for branching or straight line schedules, statically or dynamically, it is often the case that several operations are available for execution in the same cycle. In the case where such operations cannot all be executed together a selection criterion must be employed. For straight line schedules Graham has shown that a random choice guarantees a bound of $2 \Leftrightarrow 1/m$ from the optimum for an m processor machine. When conditionals are present a simple scheduling heuristic extending Graham's random rule is introduced. The heuristic, termed greatest weight first or GWF, gives priority to operations belonging to execution paths with greatest probability. If all operations have the same duration the bound on optimality guaranteed by the GWF rule is shown to vary between $2 \Leftrightarrow 1/m$ and $2 \Leftrightarrow 1/m + (m \Leftrightarrow 1)/2m \cdot \lceil \log_2 m \rceil$ depending on the skewing of branch probabilities.

The experiments performed by Riseman & Foster and Nicolau & Fisher estimated the parallelism that could be extracted from sequential applications by a clairvoyant dynamic scheduler. More precisely the parallelism was measured by assuming infinite hardware capabilities, perfect alias analysis and path prediction and was insensitive to dependence irregularities. Thus these studies pointed out what could be hoped for in the best of cases, not what could be realistically expected. In practice variable aliasing results in more constraining operation dependences for static schedulers whereas dynamic schedulers have limited operation lookahead. In addition this dissertation shows that irregularity of dependences, path prediction and space/time tradeoff may further reduce the parallelism statically or dynamically exploitable. The negative effect of various factors on statically/dynamically extractible parallelism is summarized in the following table. If a particular scheduler is unaffected by a given factor the corresponding slot is left blank, otherwise it contains the symbol $\surd$.

	variable aliasing	dependence irregularity	limited lookahead	path prediction	space/time tradeoff
Static scheduler	$\surd$	$\surd$		$\surd$	$\surd$
Dynamic scheduler			$\surd$	$\surd$	

In view of these results it is important that more realistic experiments be performed. Some simulations in this direction have recently been performed by Wall [60]. Wall shows that fine grained parallelism realistically available is an order of magnitude less than that reported by Riseman & Foster and Nicolau & Fisher. Unfortunately Wall's experiments do not inspect sequential applications but rely on a compiler to generate machine code which is then tested for parallelism. Furthermore the measurements report the parallelism that can be extracted with today's state of the art techniques. No attempt is made to classify applications according to technology independent parameters. Indeed it would be useful for future generation of hardware designers and parallelization experts to classify sequential applications according to parameters such as fraction of conditionals per execution path, skewness of path probabilities, degree of aliasing, regularity of loop dependences, space/time tradeoff factor and parallelism available as a function of lookahead size at the source. As long as such classification is missing the family of sequential applications whose fine grained parallelism can commercially be exploited remains unclear.

Bibliography

[1] A. V. Aho, J. D. Ullman, and J. E. Hopcroft, *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

[2] A. Aiken, *Compaction-Based Parallelization*, PhD thesis, Cornell University, Ithaca, New York, June 1988. TR 88-922.

[3] A. Aiken and A. Nicolau, *A development environment for horizontal microcode*, IEEE Transactions on Software Engineering, 14 (1988), pp. 584–594.

[4] ——, *Optimal loop parallelization*, in Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation (Atlanta, Georgia), New York, New York, June 1988, ACM, pp. 308–317.

[5] ——, *Perfect pipelining: a new loop parallelization technique*, in ESOP '88: 2nd European Symposium on Programming (Nancy, France), New York, New York, June 1988, Springer-Verlag, pp. 221–235. Lecture notes in Computer Science No. 300.

[6] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, *An overview of the PTRAN analysis system for multiprocessing*, Journal of Parallel and Distributed Computing, 5 (1988), pp. 617–640.

[7] R. Allen and K. Kennedy, *Automatic translation of Fortran programs to vector form*, ACM Transactions on Programming Languages and Systems, 9 (1987), pp. 491–542.

[8] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, *The IBM System/360 model 91: Machine philosophy and instruction handling*, IBM Journal of Research

and Development, 11 (1967), pp. 8–24.

[9] A. E. Charlesworth, *An approach to scientific array processing: The architectural design of the AP–120B/FPS-164 family*, IEEE Computer, 14 (1981), pp. 18–27.

[10] E. G. Coffman, *Computer and Job-shop Scheduling Theory*, John Wiley and Sons, New York, New York, 1976.

[11] E. G. Coffman and R. L. Graham, *Optimal scheduling for two processor systems*, Acta Informatica, 1 (1972), pp. 200–213.

[12] R. P. Colwell, R. P. Nix, J. J. O'Donnel, D. B. Papworth, and P. K. Rodman, *A VLIW architecture for a trace scheduling compiler*, IEEE Transactions on Computers, C-37 (1988), pp. 967–979.

[13] R. Cytron, *Doacross: Beyond vectorization for multiprocessors*, in Proceedings of the 1985 International Conference on Parallel Processing (Penn State University, Pennsylvania), Silver Spring, Maryland, Aug. 1986, IEEE and ACM, pp. 836–844.

[14] K. Ebcioğlu, *A compilation technique for software pipelining of loops with conditional jumps*, in 20th Annual Microprogramming Workshop (Colorado Spring, Colorado), New York, New York, Dec. 1987, IEEE and ACM, pp. 69–79.

[15] ——, *Some design ideas for a VLIW architecture for sequential-natured software*, in Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing (Pisa, Italy), Amsterdam, the Netherlands, Apr. 1988, North-Holland, pp. 1–21.

[16] K. Ebcioğlu and T. Nakatani, *A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture*, Languages and Compilers for Parallel Computing. Research Monographs in Parallel and Distributed Computing, MIT Press, 1989, ch. 12.

[17] K. Ebcioğlu and A. Nicolau, *A global resource-constrained parallelization technique*, in International Conference on Supercomputing (Crete, Greece), New York, New York, June 1989, ACM, pp. 154–163.

[18] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, ACM doctoral dissertation awards; 1985, MIT Press, Boston, Massachussets, 1986.

[19] J. Ferrante, K. J. Ottenstein, and J. D. Warren, *The program dependence graph and its use in optimization*, ACM Transactions on Programming Languages and Systems, 9 (1987), pp. 319–349.

[20] J. A. Fisher, *Trace scheduling: A technique for global microcode compaction*, IEEE Transactions on Computers, C-30 (1981), pp. 478–490.

[21] ——, *Instruction level parallelism.* NYU Computer Science Colloquium, Apr. 1990.

[22] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau, *Parallel processing: A smart compiler and a dumb machine*, in Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction (Seattle, Washington), New York, New York, June 1984, ACM, pp. 37–47.

[23] M. R. Garey and D. S. Johnson, *Computers and Intractability - A Guide to the Theory of NP-Completeness*, Freeman, New York, New York, 1979.

[24] R. L. Graham, , D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, Addison Wesley, 1989.

[25] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, *Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey*, vol. 5 of Annals of Discrete Mathematics, North Holland Publishing Company, 1979, pp. 287–326.

[26] D. H. Greene and D. E. Knuth, *Mathematics for the Analysis of Algorithms*, Birkhäuser, 1982.

[27] N. S. Grigor'yeva, I. S. Latypov, and I. V. Romanovskii, *Cyclic problems of scheduling theory*, Tekhnicheskaya Kibernetika, (1988), pp. 3–11. English translation.

[28] R. Gupta, *A reconfigurable LIW architecture and its compiler*, PhD thesis, University of Pittsburgh, Pittsburg, Pennsylvania, 1987. Technical Report 87-3.

[29] R. Gupta and M. L. Soffa, *Region scheduling: An approach for detecting and redistributing parallelism*, IEEE Transactions on Software Engineering, 16 (1990), pp. 421–431.

[30] IBM Corporation, *IBM RISC System/6000 Technology*, IBM Corporation, 1990.

[31] R. M. Karp, *A Characterization of the Minimum Cycle Mean in a Digraph*, vol. 23 of Discrete Mathematics, North Holland Publishing Company, 1978, pp. 309–311.

[32] R. M. Karp and R. E. Miller, *Properties of a model for parallel computations: Determinacy, termination, queueing*, SIAM Journal of Applied Mathematics, 14 (1966), pp. 1390–1411.

[33] K. Karplus and A. Nicolau, *Efficient hardware for multi-way jumps and prefetches*, in 18th Annual Microprogramming Workshop (Pacific Grove, California), Washington, DC, Dec. 1985, IEEE and ACM, pp. 11–18.

[34] ——, *A compiler-driven supercomputer*, Applied Mathematics and Computations, 20 (1986), pp. 95–110.

[35] P. Kogge, *The microprogramming of pipelined processors*, in Proceedings of the 4th Annual Symposium on Computer Architecture, Silver Spring, Maryland, 1977, IEEE, pp. 63–69.

[36] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, *Dependence graphs and compiler optimizations*, in 8th ACM Symposium on Principles of Programming Languages, ACM, New York, New York, 1981, pp. 207–218.

[37] ——, *The structure of an advanced retargetable vectorizer*, in Tutorial on Supercomputers: Designs and Applications, K. Hwang, ed., IEEE Press, New York, New York, 1984, pp. 163–178.

[38] M. Lam, *Software pipelining: an effective scheduling technique for VLIW machines*, in Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation (Atlanta, Georgia), ACM, June 1988, pp. 318–328.

[39] S. Lam and R. Sethi, *Worst case analysis of two scheduling algorithms*, SIAM Journal of Computing, 6 (1977), pp. 518–536.

[40] E. L. Lawler, *Combinatorial Optimization*, Holt, Rinehart and Winston, 1976.

[41] J. K. Lenstra and A. H. G. Rinnooy Kan, *Complexity of scheduling under precedence constraints*, Operations Research, 26 (1978), pp. 22–35.

[42] A. Munshi and B. Simons, *Scheduling loops on processors: Algorithms and complexity*, SIAM Journal of Computing, 19 (1990), pp. 728–741.

[43] T. Nakatani and K. Ebcioğlu, *Using a lookahead window in a compaction-based parallelizing compiler*, in 23rd Annual Microprogramming Workshop, Nov. 1990, pp. 57–68.

[44] A. Nicolau, *Parallelism, Memory Anti–aliasing and Correctness Issues for a Trace Scheduling Compiler*, PhD thesis, Yale University, New Haven, Connecticut, June 1984.

[45] ——, *Uniform parallelism exploitation in ordinary programs*, in Proceedings of the 1985 International Conference on Parallel Processing (Penn State University), Silver Spring, Maryland, Aug. 1985, IEEE and ACM, pp. 614–618.

[46] A. Nicolau and J. A. Fisher, *Measuring the parallelism available for very long instruction word architectures*, IEEE Transactions on Computers, C-33 (1984), pp. 968–976.

[47] R. Paige, *Transformational programming – applications to algorithms and systems*, in Proceedings of the 10th ACM Symposium on Principles of Programming Languages, ACM, Jan. 1983, pp. 73–87.

[48] P. W. Purdom and C. A. Brown, *The Analysis of Algorithms*, Holt, Rinehart and Winston, 1985.

[49] B. R. Rau, C. D. Glaeser, and R. L. Picard, *Efficient code generation for horizontal architectures: Compiler techniques and architectural support*, in 9th Annual Symposium of Computer Architecture, Silver Spring, Maryland, Apr. 1982, IEEE and ACM, pp. 131–139.

[50] R. Reiter, *Scheduling parallel computations*, Journal of the ACM, 15 (1968), pp. 590–599.

[51] E. M. Riseman and C. C. Foster, *The inhibition of potential parallelism by conditional jumps*, IEEE Transactions on Computers, C-21 (1972), pp. 1405–1411.

[52] I. V. Romanovskii, *Optimization of stationary control of a discrete deterministic process*, Kibernetika, 3 (1967), pp. 66–78. English translation.

[53] U. Schwiegelshohn, F. Gasperoni, and K. Ebcioğlu, *On optimal parallelization of arbitrary loops*, Journal of Parallel and Distributed Computing, 11 (1991), pp. 130–134.

[54] M. Sharir, *A strong connectivity algorithm and its application in data flow analysis*, Computers and Mathematics with Applications, 7 (1981), pp. 67–72.

[55] B. Su, S. Ding, and J. Xia, *URPR – An extension of URCR for software pipelining*, in 19th Annual Microprogramming Workshop (Washington, DC), New York, New York, Oct. 1986, IEEE and ACM, pp. 94–103.

[56] R. E. Tarjan, *Depth first search and linear graph algorithms*, SIAM Journal of Computing, 1 (1972), pp. 146–160.

[57] J. E. Thornton, *Design of a Computer – The Control Data 6600*, Scott, Foresman and Co., Glenview, Illinois, 1970.

[58] R. F. Touzeau, *A Fortran compiler for the FPS scientific computer*, in Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction (Montreal, Canada), New York, New York, June 1984, ACM, pp. 48–57.

[59] A. K. Uht, *Requirements for optimal execution of loops with tests*, in International Conference on Supercomputing (St. Malo, France), New York, New York, July 1988, ACM. Also available as Technical Report CS88-116, University of California, San Diego.

[60] D. W. Wall, *Limits of instruction-level parallelism*, in Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Apr. 1991, pp. 176–188.

[61] A. Zaky and P. Sadayappan, *Optimal static scheduling of sequential loops on multiprocessors*, in International Conference on Parallel Processing, IEEE, Aug. 1989, pp. III–(130–137).