DISSERTATION

INSTRUCTION SCHEDULING USING GENETIC ALGORITHMS

Submitted by

Steven John Beaty

Department of Mechanical Engineering

In partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 1991

COLORADO STATE UNIVERSITY

October 29, 1991

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY STEVEN J. BEATY ENTITLED INSTRUCTION SCHEDULING USING GENETIC ALGORITHMS BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

Adviser

Department Head

ABSTRACT OF DISSERTATION

INSTRUCTION SCHEDULING USING GENETIC ALGORITHMS

Genetic algorithms are a robust adaptive optimization technique based on a biological paradigm. They perform efficient search on poorly-defined spaces by maintaining an ordered pool of strings representing regions in the search space. New strings are produced from existing strings using the genetic-based operators of recombination and mutation. Combining these operators with natural selection results in the efficient use of hyperplane information found in the problem to guide the search. The searches are not greatly influenced by local optima or non-continuous functions. Genetic algorithms have been successfully used in problems such as the traveling salesperson and scheduling job shops. Instruction scheduling are also modeled as these types of problems, which motivated the application of genetic algorithms to this domain.

Steven J. Beaty
Department of Mechanical Engineering
Colorado State University
Fort Collins, Colorado 80523
Fall 1991

ACKNOWLEDGEMENTS

As with all efforts of this magnitude, rarely does one work without a large amount of support.

My deepest thanks go to Phil Sweany, my compiler compadre, for his work on the ROCKET milieu, for his deep understanding of the problems involved in the translation of one language to another, and for his friendship that made all of this bearable.

I would like to thank my committee for their flexibility and interest in this work. Thanks go to Dr. Darrell Whitley for his expertise with genetic algorithms. My special thanks to Dr. Gearold Johnson, for recognizing Engineering when he sees it and for taking special care of me.

I would like to thank two members of the Usenet community: Thanasis Tsantilas (`thanasis@cs.columbia.edu`) for pointing me in the direction of Peter Winkler (`pw@bellcore.com`.) Peter provided me with an early copy of the work on counting linear extensions.

My family played a part, each contributing, both historically and recently, a part of themselves that has always helped me reach my goals.

A large part of my support for this effort came from my wife, Paula. She was there when I needed intellect, compassion, and someone to support me. She understood what I was going through, as she herself has walked the same path. If mine is the brain behind this work, hers is the soul.

This work was self-funded.

DEDICATION

Dedicated to everyone who has bet on a dark horse that came in.

In memory of Susan Gayle Kennedy.

CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

As the desire for computation speed is increasing at an apparently unbounded rate, different methods have been applied to increase the amount of computation accomplished in a given amount of time. One current approach is advanced by the Reduced Instruction Set Computer method [Kat85, PS81]. The basic tenet is to speed up the processor clock as much as possible. For this to occur, the processor must be simplified so deleterious physical effects, such as gate delay and the speed of light in the media, are reduced. With this simplification, each instruction cannot manipulate much information. The overall speed of the processor is increased by raising the basic clock rate to ever greater heights. These processors appear to work well in the general consumer computer field.

Another approach to increase the amount of computing work done per unit time is to increase the amount of information processed during each processor clock tick. Proponents of this approach purport that the inherent fine-grained parallelism found in virtually all computer programs will map better, and therefore run more rapidly on machines supporting some form of low-level parallelism. Fine-grained parallelism describes architectures implementing parallelism at an instruction or smaller level, thereby executing more than one operation in a single instruction cycle. Historically, machines exploiting this type of architecture have been application specific, realizing the inherent performance benefit while not being required to support general purpose computing. This type of system includes high-end graphics processors, language specific computers, and embedded systems. More general purpose processors are now becoming available that take advantage of this type of parallelism [KA91]. These machines may be contrasted to those with course-grained parallelism that rely on the parallelism present at the function/subroutine level for speed enhancement.

Of course these two opposing methods should not be taken to represent the entire spectrum of compute engine possibilities. The RISC approach has been termed "vertical" because of the way the code appears, with many short instructions. CISC has been termed "horizontal," with fewer long instructions. Most machines are somewhere on the "diagonal" continuum between the two. It might be clearer to picture a pyramid with RISC at the "top" and fine-grained parallel machines at the "bottom."

Terminology used to describe architectures continually evolves, making any description somewhat transient. This work will use the term "MILC" (Machine with Instruction-Level Concurrency), because no other term exists that describes all the types of machines instruction scheduling pertains to. At a minimum, a MILC has the ability to initiate more than one operation per instruction cycle, and therefore benefits from instruction scheduling. MILCs include at least the following as subsets:

1. Complex Instruction Set Computers (CISCs). This type of machine has "high-level" instructions that programming languages are (supposedly) easy to translate to. The VAX 11/780 [Pat89, Dig81] is an instance of a CISC; e.g. it includes an instruction operating on arbitrary polynomials.

2. Superscalar computers. These machines use hardware to determine whether operations can be executed simultaneously. The order instructions are presented to the execution unit can have a profound effect on the speedups possible, necessitating intelligent instruction ordering. Recent examples of this type of machine include the Intergraph Clipper [Pat90], the IBM RS6000 [IBM90], the MIPS chips [Kan87], and the Intel i860 [Int90].

3. Microcoded machines. Used as a basis for the instruction scheduling methods in this paper, they are discussed in Section 1.1.

4. (Very) Long Instruction Word ((V)LIW) architectures. These have replicated processing units with lengthy (currently up to 1000 bits) instructions containing the data and control information for each unit. Fisher [FERN84], Ellis [Ell86] and Colwell et al. [CNO$^+$88] discuss VLIW architecture and translation methods suited for them. The Multiflow TRACE series of computers were based on VLIW techniques. VLIW architectures have both local and shared processor memories, an issue not addressed here.

1.1 Microcode

This work leverages previous work done in microarchitectures. Much of today's MILC heritage lies in yesterday's microcoded machines. The visibility of underlying machine features is the most pronounced similarity. To forestall any confusion regarding the use of the term "microcode," some clarification is necessary. The microcode world has been divided into four distinct groups. Tredennick [Tre82] enumerates these various "cultures" of microprogramming:

- *The commercial machine culture* uses microprogramming to implement a single instruction set on a variety of differing machines. The IBM System/360 [AST67] are good examples of machines with various internal hardware microprogrammed to accept the same "high-level" assembler language.

- *The bit-slice culture* uses generic bit-slice components programmed to perform one specific task. These machines are generally programmed by their designers so high performance is achieved. However, there currently is interest in having the end user program these machines to better match the machine and its intended application. The AMD2900 is an example of this type of architecture [Adv85].

- *The microprogrammable machine culture* builds machines intended to execute native code for various architectures by modifying (at run time) the microcode control store. The Nanodata QM1 is an example of this type of machine [KD81, Nan74].

- *The single chip culture* has a control store (usually a Read Only Memory) on the chip which is microprogrammed. A processor of this type is microcoded as an implementation technique and the microcode usually manages the low-level register–to–register and register–to–functional-unit transfers. The Motorola 68000 series chips are an example of this type of "microprogramming" [HL85].

This work is targeted to, and MILCs are related to, the bit-slice culture. The other three cultures are based on implementing a specific instruction set in the target machine. Implementing instruction generally requires short sequences of microcode independent from one another. Compilers are less useful for this type of code because both the amount of program information and the amount of global optimization possible are less.

Traditionally, fine-grain architectures have been programmed by hand, and in some sense "close" to the actual hardware. There have been few programming abstractions used in order to fully utilize the available resources. Compilers become useful when complex control and data structures are desired to increase programmer productivity. Compilers also allow persons not intimately familiar with an architecture to use it effectively. If a compiler can use the resources present in a machine, e.g., as effectively as hand-generated code, there is no reason to hand code. It has been shown that, for the most part, programmer output is invariant upon the language chosen. This speaks to the usefulness of high-level languages when creating, modifying and maintaining any code.

The use of microcode has advantages beyond the basic one of speed. In [Veg82] four more advantages are listed:

- *Flexibility.* Many design decisions can be delayed until later in the design cycle.
- *Extensibility.* Once an architecture is on the market, it can be extended with additional microcode, perhaps to tailor the machine to a specific application.
- *Cost.* The number and type of components can be reduced using a microcoded control store; the information density is higher in such a store compared to hardwired logic.
- *Simplicity.* Complex instructions such as string compare and table translation, both of which are simple algorithmically, are simple to realize in microcode.

1.2 Compaction

Because this work owes much to previous work done in microcode compaction techniques, a review is in order. A thorough survey of local compaction techniques is found in [LDSM80]; much of this section's terminology and content relies on that paper. Microprogrammable processors allow the simultaneous control of multiple hardware resources. This is achieved by not greatly encoding the control word of the machines. Compaction involves choosing, from the large solution set of possible concurrent instructions, one packing that hopefully reduces both the execution time and the space of the program.

A microprogram is composed of *microinstructions (MIs)*. During execution, the MI is the control word for the machine. Each separate machine operation within an MI is called a *microoperation (MO)*. A *field* within the instruction word of a machine controls a primitive machine activity. An MO may use one or more fields in order to execute its functionality. Expanding upon the earlier definition, if only one or a very few MO's fit within a machine's MI, the machine is said to be *vertical*. Conversely, if many MOs fit in an MI, the machine is said to be *horizontal*. See figure 1.1 for a fictional MI with 64 bits. This particular MI has fields to control the behavior of

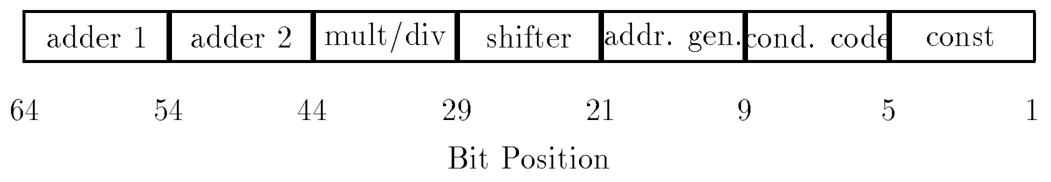

Figure 1.1: A hypothetical machine word

two adders, a pipelined multiply/divide unit, a barrel shifter, an address generator, a condition code register, and has room for program constants. Each operation is independent of all others, allowing a large amount of instruction-level parallelism.

A *straight line microcode sequence (SLM)* is a set of MOs with a single entry at the top and a single exit at the bottom. These are also known as *basic blocks.* When a function is performed on one SLM, the function is termed *local.* If a function is performed on multiple SLMs, the function is called *global. Data dependency analysis* provides the partial ordering on the MOs, by representing in some form (usually a dependence graph) the semantic-preserving flow of data of the original program. Semantics are said to be preserved if, after a transformation, a program produces an identical set of outputs when given an identical set of inputs. *Conflict analysis* determines whether or not a set of MOs can fit within a given MI without violating the constraints of the processor. Compaction has been erroneously termed horizontal optimization; horizontal improvement is more to the point as it cannot guarantee optimality without exponential time.

The problem compaction tries to solve then, is placing a group of microoperations necessary for correct execution of the algorithm into as few microinstructions as possible. An optimal answer has been shown by DeWitt [DeW76] to be NP-complete, i.e. computationally intractable. There are many problems related to compaction such as the Traveling Salesperson Problem and processor scheduling problem but, Landskov et al. [LDSM80] states compaction is an example of job-shop scheduling, giving a direction of attack.

1.2.1 Previous Methods

Astopas and Plukas [AP71] are credited with developing the first method of microcode compaction. Their method examines all combinations of MOs that do not violate data dependencies, and chooses the combination, containing no resource conflicts, producing the shortest MI sequence. This is an exhaustive method and cannot be bounded by reasonable execution time. Landskov et al. [LDSM80] elucidate four methods of non-exhaustive microcode compaction that follow.

Linear Analysis (LA), developed by Dasgupta and Tartar [DT76], works on an SLM, starting with an empty list of MIs. It attempts to add MOs in the order they appear in the SLM into an existing MI. If LA cannot find an MI that can accept all the constraints on the MO, it creates a new MI and places the MO into it. The word "linear" describes the method of examining MOs in the SLM, not the execution complexity. To search for a place for an MO within a list of existing MIs, LA starts at the bottom of the list and proceeds upwards until a data dependency stops the process. This finds the earliest MI the MO under consideration may be placed. This location is termed the *rise limit*. Now the MIs are searched for resource conflicts that disallow packing of the MO. If a rise limit was found, this search proceeds back down the list of MIs. If no MI below the rise limit can accept the MO, a new MI is placed at the rise limit to allow other MOs data dependent upon the current one to be placed as high as possible. If no rise limit was found and no MI can accept the current MO, a new MI is placed above the top of the list and the current MO is placed into it. This, as before, reduces data dependence as much as possible. LA is a first come, first served algorithm that places MOs as high as possible in the MI list.

The *Critical Path* (CP) method, used by Ramamoorthy and Tsuchiya [RT74], determines how few MIs can be used for the given set of MOs, and first places the MOs that must occur at a certain time for the schedule to be optimal. The minimum length for a list of MIs is simply the depth (length of the longest path) of the dependence graph. First, an *early partition* of MOs is formed. This schedules MOs into "frames" based only upon their data dependence relationship. All nodes in the graph at the same level are "scheduled" to occur at the same time, disregarding any resource conflicts. After the early partition is formed, a *late partition* is produced in an analogous method. The late partition is a "bottom up" early partition. In fact, if the dependence graph is represented by an adjacency matrix, the late partition may be computed by inverting the matrix and applying the early partition algorithm. With these two sets generated, the *critical partition* calculation is simple: any MO having an identical early and late partition is in the critical partition. For an optimal schedule, an MO in the critical schedule must be placed within an MI on its level in the critical partition. Because resource and encoding conflicts have not yet been considered by CP, this is not always possible. A *revised critical path* is formed by taking these conditions into account and placing critical path MOs in MIs. The next step is to place all remaining non-critical MOs within the critical path MIs. Resource and data dependent conflicts also arise during this process, possibly necessitating the addition of more MIs to the list.

Branch and bound (BAB), promulgated by Yau et al. [YST74], creates a tree whose nodes represent microinstructions. A path from the root to a leaf corresponds to one possible ordering of microinstructions. The tree branches whenever there is more than one possible MI that could be placed at a point in the MI list. If the tree is complete, it represents all possible MI orderings and is known as BAB exhaustive. It runs in exponential time and space based on the size of the MO list. Heuristics are usually used to prune apparently extraneous branches from the tree. This can make the time polynomial but is not guaranteed to produce an optimal answer. If the algorithm finds a path the length of the longest path through the dependence graph, it has found an optimal answer as noted above and may stop the search. The branching may also stop if the shortest path through the tree so far is remembered and compared to all new paths. At every node, BAB attempts to form as complete an MI as possible. BAB examines the *data ready set (DRS)* for each MI it processes. The DRS is the set of MOs not data dependent upon any unplaced MOs. Multiple MIs can be formed from the DRS, by permuting its order, and because it is unknown which will produce a shorter schedule, branches are made for each.

List Scheduling (LS) is BAB with only one branch followed at each node. LS chooses the branch that appears to contribute the most in producing a short, valid schedule. List scheduling starts with an initially empty MI list, and MOs are placed within this list when they are:

1. data ready (that is, when all the resources they depend upon have the necessary values), and

2. highest on the data ready list as judged by a weighting heuristic.

This weighting function has great influence upon the final schedule and therefore must be chosen with care.

1.2.2 Heuristics

All methods thus far rely on heuristics to remove the examination of parts of the search space that appear fruitless. Linear analysis tries to place MOs in the highest possible MI, reducing interference with other MOs needing placement. It also uses a first come, first served order on the MOs. Critical path places MOs not on the critical path heuristically. Branch and bound reduces the building and searching of the trees by guessing which branches will not lead to more compact code. The packing of MOs into MIs is ranked according to some metric. List scheduling has a similar approach in that MO placement is based on a set of weights assigned to each MO. The MO placed next in the MI list is the one on the data ready list with the highest weight.

Using heuristics can be difficult when attempting to arrive at an efficient yet efficacious compactor. This difficulty is compounded by several factors. The heuristics generally must be regenerated for each machine targeted. The heuristics themselves are not in a form easily understood by humans, thus making it difficult for humans to correctly guess and modify a compactor's behavior. It is also possible the heuristics do not address an issue having great import on the final code. Heuristics that work well for one ordering of MOs may not work well for another.

As noted before, the entire compaction problem is NP-complete. There are currently no methods guaranteed to produce optimal MI schedules without requiring either exponential time or space, or both. Because all previous methods rely on heuristics to reduce the amount of time used to obtain a solution, they are intimately bound to the heuristics chosen. These heuristics are picked before the execution of the compaction routine and remain static throughout. They have no ability to learn from previous runs or to take advantage of anomalous situations existing in specific compaction situations that lead to shorter code sequences. It would be desirable to use a search technique that does not involve heuristics yet provided a robust and efficient examination of the problem space. If such a method is available, and its execution time can be easily bounded, its application to the microcode compaction problem would be attractive.

Recent efforts have focused on inter-block compaction as well. Trace scheduling and percolation scheduling are two methods. Trace scheduling extends local compaction methods by allowing them to operate on more than one block at a time [Fis81, HMS87]. A trace is a loop-free section of code containing multiple basic blocks. When an entire trace is scheduled, the possibility for creating incorrect code, by moving operations past branches, arises. A bookkeeping phase makes copies of operations along other traces to correct the semantics. Traces are chosen in order of most-likely-to-execute first. Other traces can then be compacted. This emphasizes the frequently traversed paths at the possible expense of the infrequently traversed ones. This method relies upon the availability of good quality local compactors.

Percolation scheduling, as described by Nicolau [Nic85] and Aiken and Nicolau [AN88], has a small set of operations allowed between MIs in a program graph. These allow deletion of empty MIs, non-flow-control MO motion, movement of a conditional jump, and removal of redundant MOs. The operations are defined so program semantics are always preserved. Inverses of these functions are also allowed. The application of the operators is guided by higher-level heuristics.

Vegdahl [Veg82] states that more emphasis needs to be given to local compaction. Fisher et al. [FLS81] state that list scheduling may not be general enough for advanced compaction techniques. Wijaya and Allan [WA89] conclude "Local compaction methods other than list scheduling should be considered." This work will explore local compaction only, with the assumption that producing quality local schedules can improve intra-block methods.

1.3 From Compaction to Instruction Scheduling

General instruction scheduling is concerned with many of the same issues as microcode compaction. As noted before, there is increasing visibility of underlying hardware features so that intelligent exploitation of these features result in increased run-time performance. These features remain hidden from the casual programmer in order to speed the development of useful end-user code. Historically, when a particular routine needed enhanced performance, a human would hand code a *vertical migration* (to lower levels of machine abstraction) in order to increase speed. With the advent of language translation tools that take advantage of difficult-to-program architectural features, all programs generated benefit from a "virtual vertical migration."

While this work is based upon microcode compaction, different terms will be used in order that compaction will be easily differentiable from instruction scheduling. The term *operation* will replace microoperation, and *instruction* will likewise replace microinstruction. This minor change in naming is important as it communicates visibility. Terms prefaced with micro are invisible to the casual user of an architecture. These types of features must be addressed by someone intimate with the entire design of the machine.

1.4 Challenges

With all the advantages, why aren't there more horizontal processors? A typical MILC may be characterized by wide-word instructions, heavily pipelined functional and execution units, Single Instruction/Multiple Data (SIMD) processing, and complex timing between operations. These all add to the complexity for generating efficient software for this type of processor. Because such architectures are designed for efficient exploitation of fine-grain (low-level) parallelism, hand programming at the machine level is a slow, tedious, and exacting process. The use of higher-level tools has met resistance because it was felt these tools would remove the advantages these architectures have by not properly exploiting the machine's resources. If all the resources are not kept busy, the machine will be executing in a RISC-like fashion, albeit at a slower clock rate. When a large investment has been made in the development of hardware, the software must wring out all possible performance.

These architectures have multiple, and often replicated, functional units. This allows for many arithmetic, logical and addressing functions to be performed per clock cycle. Each of the functional units can access different types of data contained in different types of memories such as immediate fields, CPU registers, and off-chip memories. A benefit realized by MILCs is the lack of the communication bottleneck found in MIMD computers, because resource usage can be determined before the program is run. Each instruction is broken into many fields, each field controlling a different aspect of the architecture. There is complexity inherent in filling each field in each instruction, and this may be complicated by the presence of pipelined functional units. These units compute complex results, such as floating point multiplication or division, requiring more than one cycle of the clock. Operands are placed in the pipe and the results are available after some specified number of clock cycles. Properly filling and emptying all pipes is crucial in taking the fullest advantage of the machine.

Another difficulty in programming these machines is the nature of the code to be run on them. The trend, escalated by rapidly dropping memory prices, is to place more and more code into lower levels of machine abstraction to receive the speed benefit associated with this migration. The lifetime of processors in general is also decreasing, requiring that code be generated more frequently. All of this speaks to a need for tools to automate part of the difficult process of MILC code generation. Tools would allow programmers removed from the design process to use the architectures in their fullest potential.

1.5 Requirements

What should tools do to increase the usefulness of MILC machines? There are several principles that would increase their acceptance.

The first is machine abstraction. The use of a high level language would increase the number of potential end users by decreasing the learning curve required to program a machine. The rapid spread of the C/UNIX (tm) environment to many platforms displays the benefits of having a machine abstraction. If the end users are already familiar with the high level language chosen for the tool, the speed they can write useful programs is, of course, notably increased.

The second issue is retargetability. The lifecycle for all architectures is decreasing, necessitating the migration of code from one generation of a processor to another. Tools, such as high level languages that directly specify desired algorithms, ease this burden by providing the capability of rapid transference of working code from one machine to another. If the description of the machine is not static within the tools, only the machine description needs to change between differing architectures. The descriptions of the algorithms and the tools using the machine description remain unchanged between platforms. This would also remove the need for end user familiarity with the minutiae of the machine, and free the designers from the task of providing continuing support. The designer could provide a description for the tools and return to other projects.

The third issue is optimization. This involves reducing both the (usually opposing) time and space requirements for a program. For tools to be accepted by the programming community, they must produce quality code. There is no absolute measure for the quality of produced (correct) code, but the margin quoted by some is two to three times as long in time and space complexity as a human who is very proficient on a given architecture [BDM$^+$88]. This is a tall order for automated tools to provide, but not an unreasonable one given the current state of the art. As has been mentioned before, the tools must utilize the available resources well so they do not turn a MILC into a slow, expensive, RISC. Also, because this programming is at the lowest level of a system, its efficiency directly influences the efficiency of the rest of the system.

The advantages to have automated code generation tools for horizontal machines are manifold. Code can be generated for a machine before it is committed to silicon. This can point out bottlenecks in the architecture design that would confound both humans and tools alike. This interaction could close the design loop between hardware and its associated software, benefiting the total performance of the system. Most hardware in the past has been designed in a vacuum; input from software engineers was usually neither encouraged nor accepted. With increasing performance competition from many areas, this sort of designing becomes more difficult to tolerate as it negatively influences the overall system performance.

There should also be more confidence that the system will achieve its goals. This is especially true of migrating code from one platform to another. If an algorithm description produces working code for one, there should be reason to believe that it will work for another. The transference of code to higher performance versions of an architecture should be particularly straightforward. Tools would provide the capability to "hit the ground running" with a new machine. The software could be written at the same time the hardware is being designed/debugged/produced.

To realize all these benefits, multiple tools are required. The first is a compiler to take some form of high level algorithm description and produce well optimized code as output. An assembler usually receives this code and produces machine executable code. This allows the flexibility of human created assembly code should some critical section require intervention. This intervention should not be necessary with a well written compiler/optimizer, and a better response to this impulse would be to redirect the effort required to hand tune the code into improving the compiler's ability to generate quality code. A retargetable simulator is needed for checking, before and after the machine is in silicon, the conformity and efficiency of the code produced by the compiler. All the tools should be based upon one coherent, complete machine description. This allows for rapid retargeting.

This work is only concerned with a compiler's generation of quality code and assumes a supportive environment is available.

Chapter 2

THE SCHEDULING PROBLEM

Much work has been undertaken in methods that schedule event occurrences. This has been, in part, motivated by the impact that quality scheduling imparts to many diversified areas. In Coffman [Cof76], a thorough treatment is given to many different kinds of scheduling problems. The *general scheduling problem* is there stated as

> is there a schedule for the set of n tasks $T_1, T_2, \ldots, T_n$ with precedence constraint $\prec$ and execution times $\tau_1, \tau_2, \ldots, \tau_n$, on m processors with finishing time ω or less?

This problem has been shown to be NP-complete. Coffman [Cof76] further states that even by restricting the scheduling problem various ways (i.e. an empty precedence constraint and only two processors) the problem is still NP-complete. DeWitt [DeW76] shows the Local Compaction Problem is NP-complete by simplifying it to the Unit Execution Time Scheduling Problem, another known NP-complete problem. With the assumption that P $\neq$ NP, exhaustive search for an optimal answer is usually out of the question.

2.1 A Formal Definition of the Instruction Scheduling Problem

Gasperoni [Gas89] gives the formal definition of the *Local Compaction Problem* that follows. Let $\mathbf{N}$ be the set of naturals and $\mathbf{Z}$ be the set of integers. Given:

1. a machine $\mathcal{M}$, a set of resources $\mathcal{R} = \{r_1, \ldots, r_m\}$ that the machine possesses,

2. a resource configuration vector $\vec{\mathcal{R}}_{\mathcal{M}}$ of $\mathbf{N}^m$, where the kth entry of $\vec{\mathcal{R}}_{\mathcal{M}}$ (denoted $\vec{\mathcal{R}}_{\mathcal{M}}(k)$) gives the number of units of resource r_k available in the machine configuration [1],

3. a set of l operations $\mathcal{O} = \{op_1, \ldots, op_j, \ldots, op_l\}$,

4. a duration function $d : \mathcal{O} \rightarrow \mathbf{N}$, where $d(op_j)$ if the number of machine cycles op_j takes to execute,

5. a resource usage function $\vec{\mathcal{R}_{\mathcal{O}}} : \mathcal{O} \times \mathbf{Z} \rightarrow \mathbf{N}^m$ ($\vec{0}$ is the null vector):

$$\vec{\mathcal{R}_{\mathcal{O}}}(op_j, X) = \begin{cases} \text{the vector of the } x\text{th step of } op_j & \text{if } 0 \leq x < d(op_j) \\ \vec{0} & \text{otherwise} \end{cases}$$

   the kth entry of $\vec{\mathcal{R}_{\mathcal{O}}}(op_j, X)$ (denoted $\vec{\mathcal{R}_{\mathcal{O}}}(op_j, X)(k)$) gives the number of units of resource r_k needed in the xth time step of operation op_j,

6. a data dependence dag DDD $= (\mathcal{O}, \mathcal{E})$ imposing a partial ordering on $\mathcal{O}$ (DDD embodies the data dependences of $\mathcal{O}$'s operations),

7. a delay function $\delta : E \rightarrow \mathbf{N}$, defined on the edges of DDD, where for $e = (op_{j_1}, op_{j_2}), \delta(e)$ is the delay that has to be respected before scheduling op_{j_2}, once op_{j_1} has been scheduled.

The goal of the Local Compaction Problem is to find a schedule $\sigma : \mathcal{O} \rightarrow \mathbf{N}$ such that:

1. *minimality:* σ is of minimum length. The length of a schedule $\sigma, len(\sigma) = \max_{op \in \mathcal{O}}(\sigma(op) + d(op))$

2. *dependence constraints:* $\forall e = (op_{j_1}, op_{j_2}) \in E, \sigma(op_{j_2}) - \sigma(op_{j_1}) \geq \delta(e)$

[1] ROCKET allows easy expression of a replicated storage and functional resources.

3. *resource constraints:* define vector addition the usual way, and vector comparison $\preceq$ to be: $\vec{v_1} \preceq \vec{v_2} \leftrightarrow \forall k\{k \mid 0 \leq k \leq m\}\ \vec{v_1}(k) \leq \vec{v_2}(k)$, then the resource constraints are:

$$\forall t\{t \mid 0 \leq t \leq len(\sigma)\} \sum_{j=1}^{l} \vec{R_{\mathcal{O}}}(op_j, t - \sigma(op_j)) \prec \vec{R_{\mathcal{M}}}$$

2.1.1 Adding Encoding Conflicts

An issue the above definition does not address is that of *encoding conflicts.* These can occur within an instruction if more than one operation uses the same field. A way of adding this complication, similar in nature to resource conflicts, to the Local Compaction Problem follows. Given:

1. a field configuration vector $\vec{\mathcal{F}_{\mathcal{M}}}$ of $\mathbf{N}^m$, where the kth entry of $\vec{\mathcal{F}_{\mathcal{M}}}$ (denoted $\vec{\mathcal{R}_{\mathcal{M}}}(k)$) gives the number of units of field f_k available in the machine configuration [2],

2. a field usage function $\vec{\mathcal{F}_{\mathcal{O}}} : \mathcal{O} \times \mathbf{Z} \rightarrow \mathbf{N}^m$ ($\vec{0}$ is the null vector):

$$\vec{\mathcal{F}_{\mathcal{O}}}(op_j, X) = \{ \begin{array}{ll} \text{the field use of the } x\text{th step of } op_j & \text{if } 0 \leq x < d(op_j) \\ \vec{0} & \text{otherwise} \end{array}$$

the kth entry of $\vec{\mathcal{F}_{\mathcal{O}}}(op_j, X)$ (denoted $\vec{\mathcal{F}_{\mathcal{O}}}(op_j, X)(k)$) gives the number of units of field f_k needed in the xth time step of operation op_j,

The schedule σ must be consistent with:

$$\forall t\{t \mid 0 \leq t \leq len(\sigma) \sum_{j=1}^{l} \vec{F_{\mathcal{O}}}(op_j, t - \sigma(op_j)) \prec \vec{F_{\mathcal{M}}}.$$

[2] ROCKET allows easy expression of a replicated fields.

2.1.2 Types of Failures

With this definition, three types of failures are possible [SDX87].

- *Timing failure:* at least two paths do not allow a correct partitioning due to incompatible timing.
- *Resource failure:* there does not exist enough of a certain resource to allow a correct partitioning in the target machine.
- *Scheduling failure:* the algorithm used does not find a correct partition when one is available in the DDD.

The first two failures are not produced in any architecture that supports all the semantics of the source. It is the last item addressed in this work.

2.1.3 Similar Problems

As mentioned before, the Local Compaction Problem is said, in Landskov et al. [LDSM80], to be similar to the Job-Shop Scheduling Problem. Coffman [Cof76] demonstrated that Job-Shop Scheduling is an NP–complete problem. Coffman also proposes several polynomial methods for attacking the problem, one of which is *list scheduling*. Chapter 4 goes into detail about the list scheduling algorithm. With the similarity to Job-Shop Scheduling, an investigation into methods producing good results for that problem should be explored for use in instruction scheduling.

2.2 Representation

The choice of how to represent a problem in a data structure often plays a major role in how the problem is solved. This allows the benefit of using existing algorithms along with the possible deleterious effects of using existing algorithms (i.e. using methods ill-suited for the current application.) A natural representation of dependencies between program operations has been that of a directed acyclic graph (dag). Operations are only dependent upon operations preceding them, resulting in the acyclic nature of the graph. Nodes in a dag represent the operations that must be executed in order to perform the semantics specified in the source. Usually additional information is included in each node to identify both the fields needed and the resources used and defined by the operation. This provides all the information needed by the scheduler at each node.

2.2.1 Dags and Orders

Edges provide a *partial order* on the nodes such that an edge between nodes specifies when nodes can execute relative to each other. [Knu73] defines a partial order on a set S is a relation between the objects of S, denoted with "$\preceq$," satisfying the following properties for any objects x, y, and z (not necessarily distinct) in S:

1. If $x \preceq y$ and $y \preceq z$, then $x \preceq z$. (Transitivity.)

2. If $x \preceq y$ and $y \preceq x$, then $x = y$. (Antisymmetry.)

3. If $x \preceq x$. (Reflexivity.)

If $x \preceq y$ and $x \neq y$ then we write $x \prec y$. $\preceq$ is termed "precedes or is equal to", and $\prec$ is termed "precedes." For the relation $\prec$, transitivity is defined as are the two additional properties:

1. If $x \prec y$, then $y \not\prec x$. (Asymmetry.)

2. $x \not\prec x$ (Irreflexivity.)

Either way of defining partial order may be used where appropriate.

Given a partially ordered dag, a question that arises is determining the *total orders* consistent with the partial order. That is, to embed the partial order in a linear order, i.e., to arrange the objects into a linear sequence $a_1, a_2, \ldots, a_n$ such that whenever $a_j \prec a_k$ we have $j < k$ [Knu73]. A method for producing this result

```
topo_sort ($dag$)
{
        set $Q = dag(N)\{\forall a, N \mid a \prec N = \emptyset\}$;
        while ($Q \neq \emptyset$)
        {
                $x = z\{z \mid z \in Q\}$;
                print $x$;
                $dag - x$;
                $Q - x$;
                $Q \cup y\{\forall a, y \mid a \prec y = \emptyset\}$;
        }
}
```

Figure 2.1: Topological sort

is called a *topological sort*. Figure 2.1 gives an algorithm for topologically sorting a dag from Hecht [Hec77].

A desirable property the topological sort algorithm has is the operation is *possible* for every partial ordering. This means it will always produce a total order, of the possibly many available, given a partial order. Topological sorting is one method of creating total orders from partial orders, but certainly not the only method.

2.2.2 Enumerating Orders

The number of edges in a dag are a concern because they limit the number of different possible total orders. A lower and upper bound can be calculated to demonstrate this. In a completely inter-connected dag, the number of different possible orderings is

$$D(N, E) \prod_{l=0}^{l<=levels} (\text{number of nodes in } l)!$$

The *level* of a node in a dag is defined to be length of the longest path from the roots to the node. This formula can be derived by observing that all nodes at level l must be chosen before any nodes in level $l+1$. The number of different orderings at any level is the number of permutations for the nodes at that level. This results in a lower bound for a dag. The upper bound may be calculated by using a completely unconnected graph. The function is then simply

$$D(N, E)\ N!$$

This is simply the number of permutations of all the nodes. This is the upper bound for a dag and represents the ultimate in flexibility. As the number of orderings increase, the number of different final schedules increase, allowing a scheduler more opportunities to create good schedules. A difficulty with reducing the number of edges is the resultant increase in the size of the search space. To make use of the increased flexibility, a powerful search technique must be used.

Recent work by Brightwell and Winkler [BW90] has shown that determining the actual number of total orders in a dag, given a partial ordering, is #P–complete. That is, the problem is at least as hard as finding all the Hamiltonian circuits existing in a graph [GJ79]. #P–complete enumeration problems are thought to be "harder" than their corresponding NP–complete existence problems. For example, if P=NP, and it could be shown in polynomial time that an arbitrary graph *contains* a Hamiltonian circuit, it is not apparent that this would provide a polynomial time method of knowing *how many* Hamiltonian circuits exist. This has a large impact on the existing methods of instruction scheduling.

2.3 Dags to Represent Data Dependence

Data dependence between operations in a program is usually represented in a data dependence dag (DDD). Data dependence concepts and standard terminology are widely discussed in the literature [BSKT79, PKL80, Veg82, PW86, Ban88]. The three basic types of data dependence are

- *Flow Dependence* — sometimes called true dependence or data dependence. An operation m_2 is flow dependent on operation m_1 if m_1 executes before m_2 and m_1 writes to some memory location read by m_2. (This is read-after-write dependency.)

- *Anti-Dependence* — sometimes called false dependence. An operation m_2 is anti-dependent on operation m_1 if m_1 executes before m_2 and m_2 writes to some memory location read by m_1, thereby destroying the value read by m_1. (This is write-after-read dependency.)

- *Output Dependence* An operation m_2 is output dependent on operation m_1 if m_1 executes before m_2 and m_2 and m_1 both write to the same location. (This is write-after-write dependency.)

For completeness (with two operations: read and write, taken two ways), it may be noted that read-after-read usually does not create a dependency and therefore does not require an edge in the graph. Reading a value multiple times does not destroy it, except in the presence of volatile resources. If, for example, reading from a port removes a value from a queue and the next value from the queue is placed in the port, an edge is required so the reading occurs in the order specified in the source. This type of edge can be termed an *input dependence* edge.

A code fragment is illustrative

```
    ...
first_char = readchar (port1);  /* 1 */
second_char = readchar (port1); /* 2 */
if (first_char == 'a')
    ...
```

Here, there is no data dependence between statements (1) and (2) intimating their order of operation is unimportant. However the order of operation must be preserved in the resultant code in order to retain program semantics. An input dependency edge must be added to reflect this. Output dependencies can result from writing similar volatile machine resources.

2.3.1 Necessary Edges

Of the four types of edges, only true dependency edges are usually required for program correctness. These result from the expression of semantics in the source. In the presence of volatile resources, input and output dependencies can be added to express the lexical ordering of operations [3].

Without volatile resources, read-after-read does not destroy information and therefore requires no edge in the DDD. Also, without volatile resources the other two are remnants from decisions made by the translation from the source. This is a reflection of modern architectures and languages. In architectures, a write to a resource destroys information that might be needed later. In non-single-assignment languages, variables may hold different values during the course of program execution. A variable's value is well defined by language rules such as lexical ordering and control flow constructs.

A write-after-write dependency reflects dead code, either in the source program or in operations generated by the compiler. A dead code removal phase should remove these dependencies before instruction scheduling occurs. Write-after-read dependencies are created by the desire to reuse a resource before its value has been used for the last time. The removal of this type of dependency has a large impact upon the resulting schedule. Resource binding should occur as late as possible to ameliorate this impact. Section 3.4 and Sweany and Beaty [SB90] discuss a method of delaying resource binding so fewer anti-dependencies will exist in a DDD. When there are not enough resources of the necessary type to hold all values live at a point in the program, anti-dependencies may have to be introduced.

Edges in a DDD describe the order the operations must appear in the *final schedule*. It *does not* restrict the order operations may be placed *during* scheduling. The reason for this emphasis will be apparent.

[3] Volatile resources are easy to model in ROCKET.

2.4 Timing on Edges

In the previous definition of the Local Compaction Problem, a function $\delta(e)$ specified the duration of an operation. This is the amount of time after an operation start its value is ready for use. If $\delta(e) = 0$, then a $\preceq$ relation is defined by the edge. If $\delta(e) > 0$ then a $\prec$ relation is defined by the edge. The $\delta(e)$ allows for expression of multi-cycle operations. It does not allow expression of operations having either transient results or results latched into a resource upon completion. These types are prevalent in architectures that, for example, contain visible pipe stages.

In order to model these two types of resources, a new function $\Delta(e)$ is defined. $\Delta(e)_{min,max}$ specifies the *range* the output from an operation is valid. That is, if *op* is scheduled at time T, the resultant value may be used any time from $T + min$ to $T + max$. For operations that latch their output, $t = \infty$. Edges in the DDD are then labeled with their corresponding $\Delta(e)$ values. This allows for great flexibility in describing architectural features to the scheduler.

With the addition of $\Delta(e)$, more machine features can be described, allowing the scheduler to remove some of the hardware requirements in the processor. If the scheduler can operate directly on a DDD with $\Delta(e)$ specifying an operation's duration range, a larger variety of architectural features can be exploited. For example, synchronous pipes may be directly modeled and scheduled as easily as other operations. The need for explicit pipe advances is not needed (although they can be modeled as well, if needed.)

Pipes doing arithmetic on integer and floating point types usually have varying completion times. For example, an add can take two cycles, a multiply 11, and a division 19. Some pipes have transient outputs, requiring $\Delta(e)_{min,max}\{min, max \mid min \leq max < \infty\}$. If the pipe operation includes a destination resource, the use of the value may occur any time after the operation's completion: $\Delta(e)_{min,max}\{min, max \mid min \leq max = \infty\}$. Individual pipe stages can be modeled with chains of operations. For example, a multi-stage pipe operation of length

L can be modeled with $\Delta(e)_{min,max}\{min, max \mid min = max = L\}$. Another approach, possibly exposing more of the actual hardware implementation, is to model each stage of the pipe individually, each separated by $\Delta(e\star)_{min,max} = \{min, max \mid min = max = 1\}$. In this case, $e\star$ represents a stage of the entire pipelined operation. The entire pipe is then modeled by the chain

$$\sum_{1}^{L} \Delta(e\star)$$

where L is the length of the pipe.

When this approach of modeling each stage in a pipe is used, it simulates a pipe reservation station at code generation time. This reduces the amount of hardware resources needed in the pipe by guaranteeing that operation semantics are preserved. Hennessy et al. [HJP$^+$82] [4] give several reasons for shifting complexity from the hardware to the software:

1. the complexity is paid for only once (during compilation) and therefore the architectural overhead required otherwise is not present during the execution of all programs, and

2. allows the concentration of effort on software instead of a complex hardware engine.

Software is not necessarily easier to design but can pay more dividends on time spent. If a function can be handled in software, more chip space will be left for functionality that cannot either be derived in software or will benefit greatly from its on-chip inclusion.

[4] There they use "MIPS" to denote Microprocessor without Interlocked Pipeline Stages.

2.5 Timing on Nodes

If only $\delta(e)$ is used, only the for an operation minimum time $\theta(op)$ can be calculated. The calculation is simply the longest path from the roots of the DDD to *op*. This produces the earliest instruction ι that *op* can be scheduled. ι is used during scheduling as the basis for placement within the instruction list. $\theta(op)$ is sufficient for machines where $\delta(e)$ is sufficient to describe the architecture. The assumption is the maximum timing for $\delta(e) = \infty$.

2.5.1 Absolute Timing

When an architecture description requires the use of non-infinite maximum timing by $\Delta(e)$, the simple node timing algorithm must also be updated. With the definition of $\Delta(e)$ on edges, timing can be assigned to the nodes (operations) as well. Using $\Delta(e)$, a range of values the operation can be scheduled in can be calculated. This range will be termed $\Theta(op)_{min,max}$ meaning *op* can be scheduled in instruction $\iota\{\iota \mid min \leq \iota \leq max\}$. This is termed the *absolute timing* for *op*. Figure 2.2 displays the algorithm for calculating the absolute timing for the nodes in a DDD given in [AM88].

The addition and subtraction operations above must take into account the possibility of ∞ values.

The idea behind the algorithm is to keep the operations as early as possible but as late as necessary. Keeping them as early as possible allows for better scheduled results by creating more flexible DDDs. Because operations must remain in semantic order, increasing their start times may be necessary. Note that either successors or predecessors of a node can change the range where its operation is valid.

A complication not addressed in [AM88] is the possibility of legal loops in the timing. In Figure 2.3, if the absolute timing algorithm is run, the following occurs:

1. Starting with $\Theta(2) = (10, \infty)$: $\Theta(1) = (11, \infty)$, $\Theta(3) = (11, \infty)$.

```
set_time ($node$)
{
    for each successor, $succ$, of $node$
    {
        if update_time ($succ$, $node$, SUCC) alters timing for succ
        {
            set_time ($succ$);
        }
    }
    for each predecessor, $pred$, of $node$
    {
        if update_time ($pred$, $node$, PRED) alters timing for pred
        {
            set_time ($pred$);
        }
    }
}
update_time ($this\_op, prev\_op, direction$)
{
    if ($direction ==$ PRED)
    {
        $\Theta(this\_op)_{min} = \max(\Theta(this\_op)_{min},$
            $\Theta(prev\_op)_{min} - \Delta(prev\_op \rightarrow this\_op)_{max});$
        $\Theta(this\_op)_{max} = \min(\Theta(this\_op)_{max},$
            $\Theta(prev\_op)_{max} - \Delta(prev\_op \rightarrow this\_op)_{min});$
    }
    else // direction == SUCC
    {
        $\Theta(this\_op)_{min} = \max(\Theta(this\_op)_{min},$
            $\Theta(prev\_op)_{min} + \Delta(prev\_op \rightarrow this\_op)_{min});$
        $\Theta(this\_op)_{max} = \min(\Theta(this\_op)_{max},$
            $\Theta(prev\_op)_{max} + \Delta(prev\_op \rightarrow this\_op)_{max});$
    }
}
```

Figure 2.2: Original absolute timing algorithm

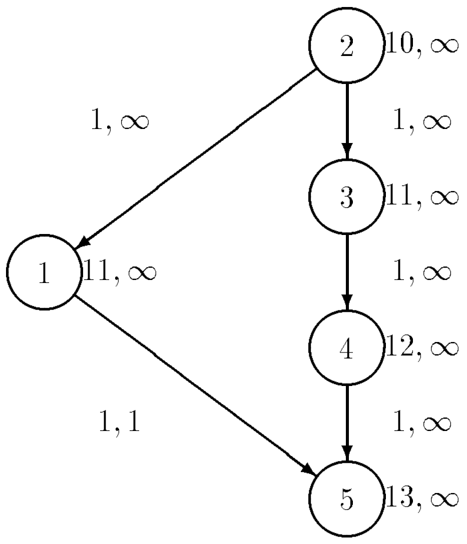

Figure 2.3: A graph where the absolute timing algorithm can fail

2. Recurse to $\Theta(3)$: $\Theta(4) = (12, \infty)$.

3. To $\Theta(4)$. $\Theta(5) = (13, \infty)$.

4. At $\Theta(5)$, change predecessors: $\Theta(1) = (12, \infty)$, $\Theta(4)$ unchanged.

5. Recurse to $\Theta(1)$, change predecessors: $\Theta(2) = (11, \infty)$, goto 1.

The difficulty occurs because the routine follows a (n, ∞) edge to a previous node, changing its timing. An important observation is

> there is no reason to follow (n, ∞) edges to predecessors. The absolute timing algorithm is only interested in making the timing on nodes *later*, and with ∞ maximum timing on an edge, no reason exists to make a predecessor node later. The operation may occur as early as possible and the value will remain valid forever.

With this observation, a revised algorithm may be created. The routine in Figure 2.4 defines how successor node's timings are updated.

The assertions check for timing errors. For example, a timing error is created when, for a scheduled node, its time must be updated in order to fulfill the constraints of the surrounding nodes. This is an obvious impossibility as a scheduled node has its timing completely restricted. Also, for example, if a node's earliest time becomes later than its latest time, a timing error with respect to its neighbors is present. The assertions provide a means for detecting timing errors in the DDD when the order of packing being attempted is erroneous. This has the same result as *extended timings* [SDX87] have, although not necessarily as early in the scheduling process.

To correctly set predecessor times, the routine in Figure 2.5 may be used. The assertions for this routine are similar, they also check to see whether the timing is still valid.

```
update_succ_time (prev_op, this_op)
{
        changed = FALSE;
        if ($\Theta(this\_op)_{min} < \Theta(prev\_op)_{min} + \Delta(prev\_op \rightarrow this\_op)_{min}$)
        {
                assert (!scheduled);
                $\Theta(this\_op)_{min} = \Theta(prev\_op)_{min} + \Delta(prev\_op \rightarrow this\_op)_{min}$
                changed = TRUE;
        }
        if ($\Theta(this\_op)_{max} > \Theta(prev\_op)_{max} + \Delta(prev\_op \rightarrow this\_op)_{max}$)
        {
                assert (!scheduled);
                $\Theta(this\_op)_{max} = \Theta(prev\_op)_{max} + \Delta(prev\_op \rightarrow this\_op)_{max}$
                changed = TRUE;
        }
        assert ($\Theta(this\_op)_{min} \geq \Theta(prev\_op)_{min} + \Delta(prev\_op \rightarrow this\_op)_{min}$);
        assert ($\Theta(this\_op)_{min} \leq \Theta(prev\_op)_{max} + \Delta(prev\_op \rightarrow this\_op)_{max}$);
        assert ($\Theta(this\_op)_{max} \geq \Theta(prev\_op)_{min} + \Delta(prev\_op \rightarrow this\_op)_{min}$);
        assert ($\Theta(this\_op)_{max} \leq \Theta(prev\_op)_{max} + \Delta(prev\_op \rightarrow this\_op)_{max}$);
        assert ($\Theta(this\_op)_{min} < \infty$);
        assert ($\Theta(this\_op)_{min} > 0$);
        assert ($\Theta(this\_op)_{min} \leq \Theta(this\_op)_{max}$);
        return changed;
}
```

Figure 2.4: Update successor timing

```
update_pred_time (succ_op, this_op)
{
        if ($\Delta(prev\_op \rightarrow this\_op)_{max} = \infty$)
        {
                return FALSE;
        }
        changed = FALSE;
        if ($\Theta(this\_op)_{min} < \Theta(succ\_op)_{min} - \Delta(succ\_op \rightarrow this\_op)_{max}$);
        {
                assert (!scheduled);
                $\Theta(this\_op)_{min} = \Theta(succ\_op)_{min} - \Delta(succ\_op \rightarrow this\_op)_{max}$;
                changed = TRUE;
        }
        if ($\Theta(this\_op)_{max} > \Theta(prev\_op)_{max} - \Delta(prev\_op \rightarrow this\_op)_{min}$);
        {
                assert (!scheduled);
                $\Theta(this\_op)_{max} = \Theta(prev\_op)_{max} - \Delta(prev\_op \rightarrow this\_op)_{min}$;
                changed = TRUE;
        }
        assert ($\Theta(this\_op)_{min} \geq \Theta(prev\_op)_{min} - \Delta(prev\_op \rightarrow this\_op)_{max}$);
        assert ($\Theta(this\_op)_{min} \leq \Theta(prev\_op)_{max} - \Delta(prev\_op \rightarrow this\_op)_{min}$);
        assert ($\Theta(this\_op)_{max} \geq \Theta(prev\_op)_{min} - \Delta(prev\_op \rightarrow this\_op)_{max}$);
        assert ($\Theta(this\_op)_{max} \leq \Theta(prev\_op)_{max} - \Delta(prev\_op \rightarrow this\_op)_{min}$);
        assert ($\Theta(this\_op)_{min} < \infty$);
        assert ($\Theta(this\_op)_{min} > 0$);
        assert ($\Theta(this\_op)_{min} \leq \Theta(this\_op)_{max}$);
        return changed;
}
```

Figure 2.5: Update predecessor timing

Nota bene: there are two possibilities worth consideration when computing $\Theta(op)_{min}$ for a predecessor node:

1. $\Theta(this\_op)_{min} = \Theta(succ\_op)_{min} - \Delta(succ\_op \rightarrow this\_op)_{max}$.

2. $\Theta(this\_op)_{min} = \Theta(succ\_op)_{min} - \Delta(succ\_op \rightarrow this\_op)_{min}$.

Both produce valid $\Theta(op)_{min}$. The first produces more liberal $\Theta(op)_{min}$ allowing more flexibility during scheduling and is therefore chosen. The same logic may be used in calculating $\Theta(op)_{max}$ for a predecessor node.

These two routines guarantee $\Theta(op)$ is within the permissible range. If it already is, they do not change it. Notice it is an error to change the timing to be earlier than it currently is. The node is as early as it can be in relation to some node; if another requires it earlier a timing error is present.

As before, a recursive routine may be used to calculate the correct absolute timing for an entire DDD (Figure 2.6 .) The recursive method has the impediment of possibly visiting a node multiple times to correctly set its absolute timing. A better method would only visit a successor (predecessor) node once all its predecessors (successors) had been visited. This is possible by setting the absolute timing in successor nodes in a reverse postfix depth-first order, and similarly the predecessor nodes in a postfix depth-first order. Whenever an entire traversal in either direction is made without any absolute timing changing, the method may stop. In this way, nodes are visited as few times as possible.

The subgraph in Figure 2.7 demonstrates the need to iterate an undetermined number of times. Consider the following flow of events:

1. Originally, nodes 1 and 2 are set to $1, \infty$ and node 3 and 4 are set to $2, \infty$.

2. Because of timing (not visible in the subgraph), node 2's timing changes to $15, \infty$.

```
update_time (node)
{
        for each succ_node in node.successors
        {
                if (update_succ_time (node, succ_node))
                {
                        update_time (succ_node);
                }
        }
        for each pred_node in node.predecessors
        {
                if (update_succ_time (node, pred_node))
                {
                        update_time (pred_node);
                }
        }
}
```

Figure 2.6: Update timing

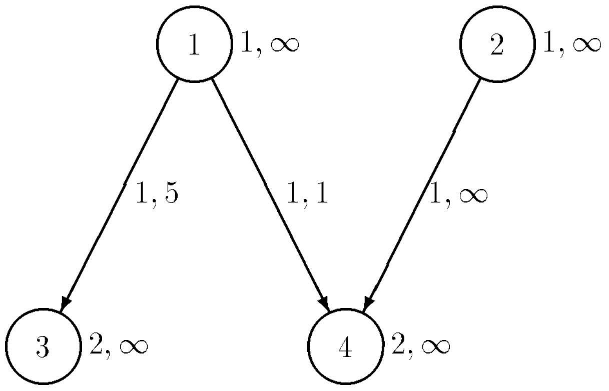

Figure 2.7: Example for the need to iterate

3. A pass down through the graph notices that node 2's timing has changed, necessitating the possible updating of its successors and predecessors.

4. Node 4's time is updated to $16, \infty$. Node 1 and 3 are unaffected.

5. On a pass back up, node 4's timing changed in the previous path, and its neighbors must be updated. Node 1's timing is changed to $15, \infty$ due to the restricted edge from node 4.

6. This leaves node 3 in need of updating, and another pass must be made to reflect this.

2.6 Summary

This chapter has shown how difficult the general scheduling problem, and a specific instance: the Local Compaction Problem, are. Scheduling problems are usually represented in a directed acyclic graph, giving a partial order to the operations. Enumerating all total orders consistent with a partial order in a dag is shown to be #P–complete. Dags are used to represent the data dependence information in the Local Compaction Problem using extensions to model both architecture-specific and translation-related information.

Chapter 3

COMPILER TECHNOLOGY

There are many different phases involved in producing quality code from a high level language algorithm description. Many are familiar to compiler writers and can be found in Aho et al. [ASU86]. These include lexical analysis, parsing, dead code removal, loop invariant code hoisting, induction variable simplification, operator strength reduction, constant folding, copy propagation and common subexpression elimination. These code improvement routines are more difficult to implement for MILCs because it is not known whether an apparent improvement actually decreases the time or space in the resulting program.

3.1 MILC Complexities

As an example of how MILCs can complicate traditional optimizations, consider hoisting a loop invariant expression out of a loop and into its header. If the expression does not add to the length of the scheduled code within the loop boundary but adds instructions when placed above, because of resource or encoding conflicts, then the "optimization" is not an optimization at all. Leaving the expression in the loop, and recomputing it each time the loop executes, costs nothing in time and space. With this in mind it can be seen that simple minded code improvements need good heuristics or proper feedback to do their job well.

Phases not usually found in compilers generating code for traditional machines are also present. In MILCs, more options exist in choosing resource bindings. An integer for example may reside in an integer register, a floating point register, an address register, an external memory, a graphics vector or a volatile latch within the machine. The choice for where to place an arbitrary integer then becomes one of great difficulty, also having a large impact on the resulting code. The information needed to make a good choice is usually available too late in the compilation process to do any good. Only after instruction scheduling has completed is information known about unused machine resources, and instruction scheduling cannot complete without all values being bound to a machine resource. In simpler machines, there are fewer choices about the placement of values into resources.

The scheduling of computations to functional units is another difficult process in MILCs. With replicated functional units or units with redundant capabilities (such as floating point units that can also do integer arithmetic), choosing where a computation is performed is difficult. Work done in scheduling problems in operating systems might be applicable here but finding an optimal solution is certainly computationally intractable. With simpler machines, there exist very few functional units to do calculations and so the choice is trivial. The difficulty in the ordering of translation processes has been termed the *phase coupling* problem, and Allan [AM87] discusses different approaches attempting to reduce its impact.

Many of the code improvement routines require a large amount of analysis to perform effectively. These must preserve the semantics of the original code while producing as flexible a representation as possible to operate upon. These analyses include:

- *Dataflow analysis.* Local dataflow analysis computes the uses and definitions for all variables within a basic block. Global dataflow analysis uses this information to form live-in and live-out sets for each basic block. This allows complete dataflow information to be found for a function, allowing the use of semantic-preserving transformations and correct register save and restore operations. If the dataflow procedures look beyond the function boundaries to encompass the entire program, the information is more accurate and less conservative transformations may take place. The intelligent passing of parameters is a particularly rich area for these transformations.

- *Dominator analysis.* Dominator relationship provides a partial ordering of basic blocks within a function. A block A is said to dominate a block B if all paths to B pass through A. This reveals the loop structures within a function, for example.

- *Symbolic evaluation, origin, and cover analysis.* These analyses provide, for each expression, a completely expanded expression based upon previous expressions and initial values, and are discussed by Reif and Tarjan in [RT81]. They provide information as to the origin of each expression, useful in delimiting the amount of code motion allowable while maintaining program semantics. They are also used to remove global common subexpressions and to propagate both copies and constants globally.

- *Memory reference disambiguation.* This procedure attempts to identify program variables that are aliases of one another. For example, this can occur in C with pointers and parameters. Memory reference disambiguation also attempts to resolve which array references are actually the same memory location [Nic84, Ell86].

With the information provided by these analyses, a compiler produces a representation semantically equivalent to the original program, but in a form much more amenable to code improvement. That this process must take place, i.e. a process generalizing the algorithm which the human entered, speaks to the fact that more abstract descriptions of the desired results would benefit the production of good quality code. That it would benefit the person who wants a problem solved should be beyond doubt.

The problem of *code selection*, present in all machines, is exacerbated in MILCs by the plethora of instructions available to perform a specified task. The expression: A * 8 may be calculated in a variety of ways using various different functional units and resources. For example, it could be performed by a multiply, a shift by three bits, or even a repeated add. Choosing the best method depends at least upon the surrounding machine resource and functional unit usage. If the expression is part of a larger address calculation whose result is used to reference a memory, a multiplicity of addressing modes increases the difficulty in determining a good sequence of instructions to compute the expression.

When all of the previous phases have completed, instruction scheduling is performed. Instruction scheduling is not explicitly present in non-MILC architectures, but is implicit when the code selector produces actual machine code as its product. Because only one operation may occur per cycle, no instruction issuance issues exist. In compilers producing scheduled code, the code selector instead produces an intermediate representation (usually a graph) denoting which operations must occur, in which order, in the final code. The scheduler takes the graph and attempts to reduce it into as few instructions as possible. Operations may be combined into a single instruction when there are no data dependencies, encoding conflicts, or resource conflicts among them. This is the phase this work addresses.

3.2 The ROCKET Compiler

This work is supported by the ROCKET compiler developed to research many areas of MILC architecture compilation. ROCKET is an offshoot of the Horizon compiler, described by Mueller et al. [MDSW88], also developed at Colorado State University. Like Horizon, ROCKET focuses on machine resource usage as the primary issue in both retargetability and production of highly optimized code. ROCKET targets to a wide variety of architectures which are assumed to have a single control store and to operate synchronously (such machines correspond to what Dasgupta and others [Das84, DDMS86] have called "clocked microarchitectures.") Beyond that, they may have arbitrarily wide control words; polyphase or monophase execution; pipelined fetch/execute; pipelined functional execution; permanent or transient storage elements with arbitrary (discrete) setup and hold times; machine operations with side effects; and branches with arbitrary (discrete) branch delays.

3.2.1 Code Generation Phases

To translate C into highly optimized code for MILC architectures, ROCKET first produces an abstract representation of an input C program and then performs *global optimization* which modifies the intermediate representation to improve expected program speed; *code selection*, which replaces abstract representations of C statements with collections of machine operations; *parallelization*, which determines resource dependencies and timing; *instruction scheduling*, which assigns machine operations to (a hopefully minimum number of) instructions satisfying data dependency and machine resource constraints; and *register assignment*, which assigns variables to machine resources.

ROCKET's global optimization includes common subexpression elimination, copy propagation, constant folding, constant propagation, algebraic simplification, induction variable simplification, and reduction in strength. Aho, et al. [ASU86]. describe these "traditional" compiler optimizations. Unfortunately, although many traditional optimizations produce improved code efficiency on vertical architectures, they do not always do so on horizontal machines. This can be attributed mainly to the need to take advantage of the available concurrency in a MILC architecture during instruction scheduling. Often, traditional optimizations perturb code in a manner that inhibits a scheduler's ability to minimize the number of instructions required to execute a program. Rather than disregard these traditional optimizations, ROCKET includes modified versions that consider parameters in the target machine description to evaluate when an optimization is likely to provide improved code. Beaty et al. discusses a similar optimization scheme in [BDM$^+$88].

After global optimization and code selection a *serial-parallel coupler/decoupler*, builds the data dependency dag (DDD) for each basic block that is an abstract intermediate representation of the source program. The result forms the input to the instruction scheduler. The coupler/decoupler, originally implemented in the Horizon compiler [MDSW88, MS86], creates and manipulates DDDs, performing ROCKET's parallelization phase in the process.

The ROCKET instruction scheduler receives generated DDDs and places the represented operations into instructions. These instructions, in a list form, are then passed to a routine that produces an output file based on information present in the machine dependent configuration database. This usage of the configuration database for this procedure allows the compiler to be "targeted" to a variety of assemblers. Information is available on the syntax the assembler expects, and few preconceptions about the final output form are made.

3.2.2 Machine Description

The ROCKET machine description is patterned after the Horizon machine description [MDSW88]. The ad hoc machine description language indicates how data passes among storage resources, and how data is transformed by functional resources, as well as specifying resource timing data, and field attributes of the instruction word.

A ROCKET target machine description includes:

- **Machine Resources** — Each machine resource has an associated setup and hold time. ROCKET views this timing abstractly in terms of instruction cycles. Resources can include replication, as with registers, such that the resource allocation process must be extended to consider the placement of values in each. Replicated resources may also be divided into reserved regions and regions under the resource allocator's control.

- **Machine Instruction Word Fields** — The compiler assumes a number of distinct encoding fields comprise the instruction word. A field's encoding values are symbolically represented in the specifications of target machine operations. ROCKET allows any number of distinct instruction formats; the scheduler correctly chooses the appropriate one. This allows the usage of pseudofields to describe machine idiosyncrasies as described by Linn and Ardoin in [LA89].

- **Machine Operations** — Each machine operation has a distinct identifier as well as a collection of instruction fields that invoke it during execution.

- **Machine Data Paths** — The code selector must extract information about how data can be moved among storage resources, how functional operations are invoked, and how instruction sequencing is invoked. This information forms the bulk of ROCKET'S machine description.

- **Activation Description** — Three different types of activation records can be described: default, non-recursive, and leaf procedures. This allows the compiler to use procedure-specific information when creating an activation record so the special cases may use faster methods. Each description contains information on how to pass and return each basic type, to call and return from a procedure, and the conventions used in ordering locals and parameters.

- **Assembler Syntax** — This includes the
  - format of labels,
  - delimiters of programs, functions, blocks, instructions, and operations,
  - predefined constants, and
  - constant formats.

3.3 Reduction of Anti-dependencies

Vegdahl [Veg82] discusses how data anti-dependencies affect the quality of code produced by compilers, and suggests methods to reduce anti-dependencies (including a constant *unfolding* technique.) At least three traditional optimization techniques may increase the execution time required by a scheduled program by adding anti-dependent edges to the DDD.

Common subexpression elimination is a process replacing replicated computations with a reference to the resource holding the value of the first computation. This reduces the amount of computation a program performs, but increases the demand placed on the resource holding the value of the operation. The operations are not limited to programmer-supplied operations, but may be present in code generated during the translation process (such as addressing expressions.) Anti-dependencies are formed, for the life of the computed value, with any other value that could use that resource.

Copy propagation replaces references to known resource aliases with a reference to one resource. This can occur with such statements as `x = y`, where all later references to `y` are replaced with references to `x`, thus allowing the copy statement to be removed. In non-MILC architectures, this can increase program execution speed. The difficulty produced is the lengthening of the live range of `x`. This range now includes the entire range of `y`, whereas it originally did not, possibly adding to the anti-dependencies present. If the copy operation can be performed at low cost, and lengthening the live range produces undue pressure on the resource holding the value, the optimization is unadvisable. Both of these conditions can be present in a variety of architectures.

Constant folding replaces references to expressions known at translation time to be constant with the computed value of the expression. If there is an immediate field in the instruction word capable of containing the constant and no

> contention exists for that field in all instructions requiring the constant, this optimization reduces the execution time of the program. Vegdahl states that immediate fields are expensive in terms of instruction bits, and limit the choice of instruction formats available for an operation. He suggests the method of constant unfolding that attempts to replace translation-time constants with simple expressions requiring fewer machine resources.

These methods also suffer from the phase coupling problem; the information needed to make correct decisions about each is unavailable until instruction scheduling has completed.

The removal of superfluous data anti-dependencies has a beneficial impact upon the final, scheduled, code. Several circumstances contribute to the production of unnecessary anti-dependent edges.

The first occurs when two references to memory cannot be differentiated, that is, are not known to be differing locations. To be conservative in insuring program semantics, undifferentiable references to the same memory must have an anti-dependent edge between them. Single assignment languages, such as Sisal [Can90], disallow the reuse of memory locations, removing the source of the problem. ROCKET currently compiles C and therefore must deal with the problem in a different manner. As mentioned before, memory reference disambiguation is used in an attempt to differentiate memory references. This can notably increase parallelism in the resulting code and is explored by Ellis [Ell86] and Nicolau [Nic84]. ROCKET includes memory reference disambiguation.

The second occurs when a given register is assigned more values than necessary, resulting in its unavailability for reuse. This situation occurs because most modern compilers attempt to use as few registers as possible. The goal of minimal register usage is a noble one that can easily increase execution speed by reducing references to off-chip memories. Competing with this goal is the effective usage of all on-chip registers. Consider the following code fragment:

```
r1 = 300
r2 = r1 + 4
r1 = 7
r5 = r1 + 5
```

If, instead of re-using `r1` for the second computation, `r3` was used, the two computations could be performed in parallel. An anti-dependency (sometimes termed a "false" dependency [Wal91]) exists between the first use of `r1` and the second definition of `r1`, limiting parallelism. The difficulty in choosing which registers to use to hold which values is again one of phase coupling, as register usage for a particular area of code is not known until scheduling is complete.

3.4 Register Assignment

Various methods have been tried to reduce the impact of register anti-dependencies. A hardware approach is *register renaming* that imposes a level of indirection between the register number referenced in the code and the actual machine resource. When an instruction sets a register value, the hardware maps that value to a resource (possibly distinct from the register number present in the instruction) for the duration of the value. This, in effect, performs register allocation at run time, possibly producing better resource usage than a compiler's static analysis could. Wall [Wal91] discusses the impact of register renaming and reports an additional benefit: the machine can contain more registers than the instruction format is able to describe. Register renaming can increase the length of the instruction pipeline, increasing the branch delay penalty. It also adds more circuitry on the die, a process that inherently slows the processor's clock rate. Wall concludes that register renaming is an important method for increasing performance if the compiler is unable to cope with the parallelism available in the machine. The question of where the effort of intelligent register assignment is best performed is an important one dealt with in the following sections.

3.4.1 Terminology

In order to avoid misconceptions about terms used in the remainder of this chapter, some definitions must be introduced.

Variable. A non-traditional definition is chosen for this term. Historically, a variable is a program entity that a human declares, defines, and uses. This definition is too narrow. For example, a compiler often generates its own variables to produce expressions that can be understood by the target architecture (e.g. temporaries.) Another possibility often overlooked is program variables that do not vary. This may be due to a programmer mistake or a compiler optimization removing the variability of a value. A better term for a variable may be a *changeable value* which better reflects the usage of the value, however variable is the accepted nomenclature and will be used for the rest of this chapter.

Symbolic or Pseudo Register. These represent a variable's current value in the compilation process until register assignment is executed. They may or may not directly reflect programmer-declared variables. In ROCKET, they do not. ROCKET's symbolic registers represent the livetrack of a *value*, albeit a changeable value, which may differ greatly from the programmer's view of a variable. ROCKET's register assignment phase maps all symbolic registers to actual machine resources, be they registers or other location capable of storing variables.

Register Unification. ROCKET initially gives the destination of each assignment operation a different symbolic register. In the absence of control flow, this produces correct code. With control flow, variables are propagated through differing paths, and those symbolic registers referencing the same value must be rectified. ROCKET contains a process, register unification, that performs this rectification. Register unification uses the control flow and dataflow information produced earlier to assure all references to a value reference the same symbolic register.

3.4.2 Graph Coloring Register Assignment

The graph coloring method proposed by Chaitan [CAC+81, Cha82] performs register allocation and assignment at the same time. Allocation decides which variables will reside in physical (hard) registers at run time; assignment places those variables into their respective hard registers. The aim is to place as many program variables into hard registers as possible. This increases the program's execution speed and reduces the code size, both desirable effects. This process is limited by the fact that variables live at the same time cannot share a hard register. The graph coloring method uses graph nodes to represent variables, and places edges between nodes whose variables are live simultaneously. The solution then involves finding an n-coloring of the graph, where n represents the number of the target machine's available hard registers. A graph is considered correctly colored if each node's color differs from all its neighbor's. As an architectural paradigm, this implies each variable is assigned a register different from all other variables live during the same execution cycles.

It is well known that given a graph G and a natural number $n > 2$, the problem of determining whether G is n-colorable is NP-complete [HS80]. That is, there exist graphs that have a search space that grows exponentially based upon the number of nodes. An exhaustive method could randomly choose a color, from a pool of n, and assign it to a node. This process could continue until the graph is colored or adjacent nodes received the same color. If adjacent nodes received the same color, the process would backtrack in search of a different, correct, solution. This can be a lengthly process, with no assurance of ever finding a solution.

Criteria can be added to this process to speed the search for a solution. Each node is given a metric of urgency, based on that variable's perceived importance, and the order of assignment is based on the nodes' urgency. This speeds the search in graphs that are, in some sense, easy to color (non-pathological graphs.) Unfortunately, solutions based on urgency criteria will still create exponential search spaces in the worst case. There is no way to easily predict the onset of exponential search, leaving this method with obvious flaws.

Another approach is possible with a simple observation about coloring graphs. If a node is removed from the graph that has degree less than n, no matter how its neighbors are colored, there will be at least one color left over for it. For example, in a graph where a four-coloring is being attempted, if a node of degree three is removed, each of its neighbors may be assigned any three colors leaving at least one color for the removed node. Nodes are removed in this manner until the graph is empty or no remaining nodes have degree less than n. This deterministic method consumes non-exponential time and space. It is not guaranteed to find n-coloring if one exists; however, it does produce excellent results in practice. Once a register interference graph has been shown by this method to be n-colorable, nodes are replaced in inverse order of removal, and colored by choosing a color not present in their neighbors. The order of replacement is significant; it is known that *at the time of removal* the node can be given a color differing from all its neighbor's. Different orders can produce a lowered register usage, but cannot guarantee colorability. A hard register (color) not used by any of a node's neighbors is chosen for it. Two methods for choosing which hard register to assign next are

- pick the lowest numbered register not used by neighbors, or
- pick registers in a round-robin fashion. In this method, a record is kept of the last hard register allocated and allocates the next higher numbered register (mod the number of colorable registers, of course) not already allocated to a neighbor in the interference graph.

The first method reduces the overall number of hard registers used, possibly reducing the number of save and restores around call sites. The second method increases the number of hard registers used, thereby possibly reducing the number of anti-dependencies required, and possibly creating a more advantageous environment for instruction scheduling.

It is possible for this method to produce erroneous solutions to certain graphs, i.e. deem a graph uncolorable when it is not. Figure 3.1 shows an example where the graph is 2-colorable, but has no nodes with order less than 2. Briggs et al. [BCKT89] give a refinement of the coloring process. They suggest removing the node with the lowest degree instead of those with degree less than n. Then replace them, as before, in inverse order of removal, giving each the lowest color not used by its neighbors. This increases the likelihood of finding a coloring because it is possible multiple neighbors will have the same color, leaving more colors available to a node. In the previous method, spilling would have already occurred. When a node is unable to be colored, it is ignored (left uncolored), and the replacement process continues. This gives an uncolored node's neighbors more colors to choose from. When the assignment procedure is complete, uncolored nodes are reassigned to another machine resource and a new interference graph is built. This method has the difficulty of not using usage cost to determine which node to reassign. Briggs et al. go on to make another improvement to the register assignment process. Nodes with degree less than n are removed in arbitrary order. Any nodes remaining in the graph are ordered for removal by their associated spill cost. All other parts of the algorithm remain invariant. Briggs et al. report good results with this procedure; ROCKET also implements this method.

When a graph is not found n-colorable, some program variables must be placed in a non-register resource. This resource is usually an off-chip read/write memory. This creates a speed penalty so these variables must be chosen carefully. Code must be generated to "spill" the variable out after each definition and to "spill in" before

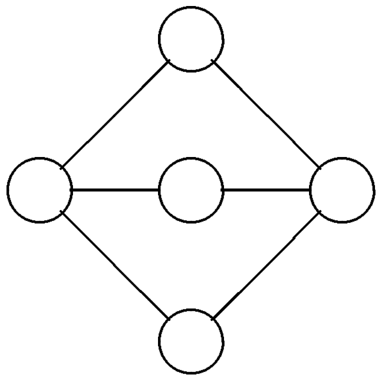

Figure 3.1: A Pathological Graph

each use [ASU86]. This reduces the pressure on the internal register bank by reducing the length of any spilled variables' live tracks. This reduces the interference caused by spilled variables, hopefully allowing the graph to be completely colored. Because the interference may not be reduced enough and hard registers are usually still temporarily needed for spilled variables, a new graph is built and the coloring/spilling cycle repeated as necessary. Variables are chosen to be spilled based on perceived cost. For example, those within nested loops or with a high number of uses will be spilled last.

When a program executes a call statement, values residing in hard registers may be destroyed by the called routine. A convention may be adopted specifying which hard registers may be overwritten by a subroutine, but this practice removes resources that could otherwise be well used. An alternative convention saves the values needed before they are overwritten and restores them when control is returned to the calling routine. Using this procedure, two choices exist: 1) the caller can save the active values before transfer of execution and restore them upon return or, 2) the callee can save upon entrance and restore before exit. The caller-save method saves all those hard registers live before and after the call; the callee-save method saves any register that will be destroyed during subroutine execution. The callee-save method has the benefit that the save/restore code is present only once whereas the caller-save has to place code around each call. A refinement to the callee-save process saves hard registers only before they are modified within the subroutine and restores only those, after the last use. Control flow constructs within the subroutine may cause certain registers not to be destroyed every time it is called. The caller-save will save fewer registers if there are fewer registers live before and after the call in the calling routine than exist within the subroutine. The inverse is true for save by callee. ROCKET uses callee-save for its code space reduction and its implementation simplicity.

3.4.3 Interference

To perform graph coloring register assignment, a register interference graph needs to be built. Fortunately, ROCKET's method works equally well with either a program's intermediate representation or final form as input. Whether to perform register assignment before or after scheduling depends exclusively on factors other than the register assignment procedure.

Before describing further how ROCKET builds the register interference graph, a brief digression is required to discuss *execution points*. Intuitively, an execution point can be thought of as a distinct step in the program's execution. Thus, the definition of an execution point differs slightly depending on the program representation being considered. In C source code, each ";" might be considered an execution point. Program analysis considers each intermediate statement a distinct execution point. After scheduling, each instruction is an execution point. The lifetime of a variable is certainly different in each form, with only the scheduled form correctly representing the actual lifetime.

Given this malleable definition, it can be determined which variables have intersecting lifetimes using two dataflow sets that the ROCKET compiler maintains for each different type of execution point.

Defined: the set of variables redefined during the execution of this point.

Live: the set of variables which "contain" values needed at this or some later execution point.

The interference graph is built using the following observation:

> A variable, R, interferes only with those variables which are live at the execution point(s) where R is defined.

Stating this notion in algorithmic form, the following pseudo-code outlines how the interference graph is built:

```
foreach execution point, P, in the program:
{
        foreach member, defR, of P->defined:
        {
                foreach member, liveR, of P->live:
                {
                        add an edge between defR and liveR
                }
        }
}
```

To build the register interference graph the defined and live sets for each program execution point must be calculated and the ability to traverse the execution points must be maintained. Since ROCKET maintains the defined and live sets at each execution point in both the intermediate and scheduled representations of the program, it is equally easy to build the interference graph either before scheduling (using intermediate statements), or after scheduling (using instructions).

3.4.4 When?

The important question remaining is when, during compilation, should register assignment take place? At least five places are possible, three are shown in Figure 3.2. Most traditional compilers perform register assignment on intermediate code, and ROCKET has this ability. For traditional architectures, this is a benevolent environment for register assignment as the final code closely mirrors the intermediate representation. This is not true for MILCs. A previous incarnation (Horizon) performed register assignment on the DDDs produced by the code selector. This representation is closer to the final form with more parallelism exposed and Beaty [Bea87] found this to be an effective method. The third place for register assignment is after instruction scheduling, where all the information is available about the final form of the code. A fourth place is during the process of code selection, long used in traditional compilers [ASU86]. Bradlee et al. [BEH91] integrate register allocation with instruction scheduling, moving the process closer to the final form of the code. These last two methods were found challenging to make completely machine-independent, and were not implemented in ROCKET.

It is desirable that register assignment be done very late in the compilation process. The myth of unlimited register resources can be maintained until after optimizations, such as common subexpression elimination, copy propagation, and dead code removal, are completed. Delaying register assignment provides several benefits. If an optimization calls for creation of a new register, (or expansion of a register variable's lifetime), the optimization can be performed given the assurance that a later register assignment process will "make everything right" with respect to the register values needed. Similarly, if an optimization (e.g. dead code removal) removes the need for a register or shortens a register variable's lifetime, the lowered register interference will be undoubtably noticed at register assignment time. Thus, the basic rationale for performing register assignment late is the preference to assign values to hard registers only after any optimizations that may change either the number of register values needed or those values' lifetimes. If registers are assigned before one or more of these optimizations, assignment and spilling decisions are based on poor estimates of the register usage in the compiler's final product.

By delaying register assignment, some difficulties are encountered. The fact cannot be ignored that no target architecture has an infinite number of registers.

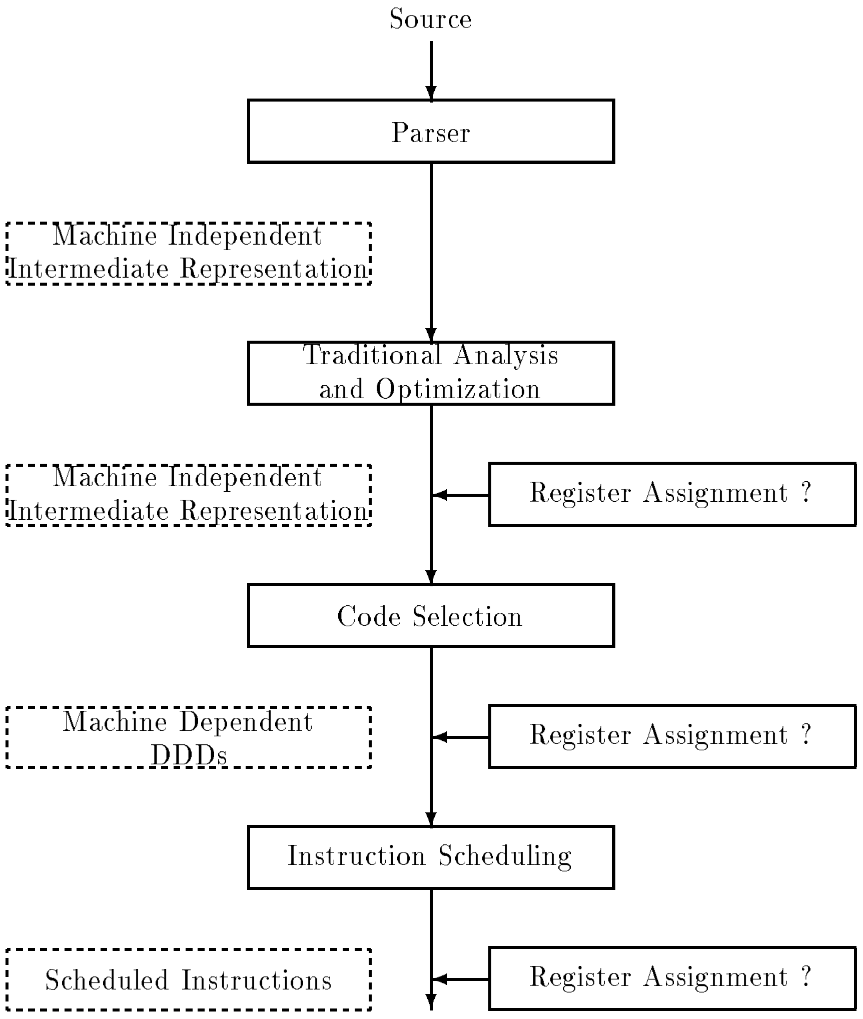

Figure 3.2: Possible placements of register assignment

Common optimizations must consider the consequences of adding to a program's register interference. Indeed, if to eliminate a common subexpression, register interference is increased to the point that register assignment spills, the cost of evaluating an expression multiple times must be great to outweigh the added spill cost. Thus, a phase coupling problem is present where intermediate optimizations and register assignment depend on one another. A common solution, discussed in Beaty et al. [BDM$^+$88] and used in ROCKET, performs register assignment after common optimizations such as common subexpression elimination but includes a "register interference" parameter in the target machine description. This parameter estimates the probability that creation of a new register value would lead to register spilling, and is larger in a machine with fewer registers.

So far, timing of graph coloring register assignment has been discussed only with respect to traditional optimizations. How does inclusion of a scheduling phase affect the optimal placement of register assignment? While scheduling itself will not create or destroy register values, it will certainly alter the lifetimes of register values by changing the relative order of operations from the intermediate code. Therefore, to use the most accurate dataflow information, register assignment is delayed until after the compiler's scheduling phase. All known compilers perform register assignment before instruction scheduling. It is assumed this is due to implementation difficulties encountered. Specifically

- in those cases requiring spilling, it is not clear how to add spill code to already scheduled instructions, and

- code for saving and restoring registers must be scheduled, causing a phase coupling problem, because until register assignment is complete it cannot be determined how many registers need to be saved/restored.

Having shown why it is useful to delay register assignment until after scheduling, the implementation issues mentioned previously must be investigated, specifically the difficulty in spilling registers in scheduled instructions and the uncertainties in saving and restoring registers around procedure calls.

One of the nice consequences of performing register assignment based upon intermediate statements is the ease with which spill code can be inserted into the intermediate code. All that is needed is to

1. construct new intermediate statements to represent the actions of storing a register's value to memory and later loading a register from that memory location, and

2. insert the new statements into the list of intermediate statements for the program being compiled.

Sadly, such insertion will not generally work with scheduled code. The scheduled code probably will contain machine operations packed into different instructions but bound by restrictive timing. Arbitrarily inserting new instructions into the scheduled code could violate the constraints under which the code was originally scheduled.

Because inserting spill code into already-scheduled instructions is undesirable, it might be desirable to add spill code to the DDD provided by the code selector. While possible, this would be extremely messy. Rather than add spill code to either the scheduled instructions or the DDD, the scheduled instructions are used to choose candidates for spilling (providing good information about spill cost), but spill code is added to the intermediate statements. With this approach spill code can be easily inserted, and spill decisions are based on register usage in scheduled code. This method's major disadvantage is the possible reduction of compilation speed. Register assignment is a loop where the process assign, spill, assign, spill, ..., continues until the assignment phase can allocate all the (remaining) variables to hard registers. By assigning registers after scheduling but spilling in the intermediates, this loop has been lengthened. Now, in addition to adding spill code for each iteration of the assign/spill loop, DDDs must also be built from the intermediate and scheduled for each iteration. A method to reduce the number of iterations register assignment requires is to spill more variables than appear needed. This will reduce the pressure on the registers being assigned, but care must be taken so that a good estimate on the number of additional spills needed is found.

Delaying register assignment until after scheduling may also lead to implementation problems in saving and restoring registers around each procedure call. It cannot be determined how many registers must be saved/restored until after register assignment. But the save/restore code also needs to be scheduled. Thus, save/restore code must precede scheduling but follow register assignment, playing havoc with the decision to schedule before assigning registers. Luckily, the decision to place the save/restore code in the callee rather than caller subprogram provides an easy solution. A crucial observation in reaching a solution is: save and restore code does not reference variables, only the hard registers that must be saved. The values residing in the hard registers are meaningless to the subroutine, and must simply be saved for the calling routine's benefit. Therefore, register assignment can ignore the register save/restore code. To insert register save/restore code in a function, ROCKET inserts two dummy basic blocks, ENTRY and EXIT, into the function's control flow graph. As the names imply, these blocks will be the first and last executed. Through most of the compilation process these blocks contain no code. After scheduling and register assignment, ROCKET inserts register save code into the ENTRY block and register restore code into the EXIT block, saving/restoring those hard registers referenced within the function. Finally, the ENTRY and EXIT blocks are scheduled.

3.4.5 Results

Much of the foregoing is summarized in Sweany and Beaty [SB90]. A hypothetical machine, a 68020-based engine with an off-chip floating point adder and an off-chip floating point multiplier, was devised to study the performance of the delayed register assignment process. Code size improvements of 20% where found in some of the longer Livermore Loops [McM86]. Shorter code segments showed less improvement. The reduction of anti-dependencies was responsible for most of the improvements. Register usage was increased due to the increased parallelism. The number of registers used by all examples did not exceed 32 integer or 32 floating point registers. With the current trend of having a larger number of registers present on-chip, this method will pay large dividends with little risk of compilation-time detriments.

Bradlee et al. [BEH91] give a schedule cost function, based upon the number of registers, of

$$schedcost_b(x) = c + \frac{d}{x^2}$$

where b is a particular basic block and x is the number of registers available. This function is for general purpose code and relates well to results found by ROCKET. For numerical code, the performance increase can be expected to be greater.

The delaying of register assignment until after instruction scheduling has several obvious advantages.

1. It reduces anti-dependencies by performing the parallelization process without consideration to resource usage. This exposes more parallelism to the scheduler by using the assumption that all scalar variables are independent. This remains true if no spilling occurs. Fewer edges are present in the graph, for no register reuse is present, increasing the scheduler's ability and flexibility in the production of quality code.

2. It directly reflects the scheduled code, not the vagaries of the source program, any intermediate form, or the target machine. If register assignment occurs any time earlier, resource usage must be guessed at, and compilers guess conservatively in order to reduce the number of registers used. The reduction of

register usage is a worthwhile goal for it can save time at call sites; if done at the expense of run-time performance, this goal is vacuous.

3. It incurs no hardware penalty. Techniques should be performed where the cost/benefit ratio is smallest. With the ease and efficacy of post-scheduling register assignment, a cogent argument can be made that resource allocation decisions are better made at compilation time rather than run time, because of the possibility of incurring the associated run-time penalties. The method also works for any machine, reducing the design cost of all new machines.

3.5 Summary

This chapter has explored the issues surrounding the production of good quality code for MILCs. Specific difficulties, such as non-improving code improvements, were addressed and methods were given for response. Motivation for performing good analyses was given. A high level description of the current ROCKET compiler was discussed as it relates to MILC compilation issues. The necessity of reducing the number of false dependency DDD edges was addressed. Several methods, including an important change in the placement of register assignment, were included.

Chapter 4

LIST SCHEDULING

Of all the methods in Section 1.2.1, list scheduling is used the most in existing schedulers. There are several reasons for this. There exists a known bound on the time it takes to execute, and it is a polynomial bound based on the size of the DDD. It has been shown by Davidson et al. [DLSM81] to produce good results in the presence of good heuristics. List scheduling is also relatively easy to implement.

4.1 Methods

In Coffman [Cof76], an outline for general list scheduling is given. An ordered list of tasks is constructed. The list is called a *priority list* because the tasks are ranked with those with the highest priority first. When a processor becomes free, the list is scanned for the first unexecuted task ready to be executed. A task can be executed given all its predecessors have been executed and there exist enough resources for the task. If not enough resources exist, the list is scanned further until a task is found that can be executed or none are ready, in which case the processor remains idle until a task on the list is ready. When the list is empty, the schedule is complete.

4.1.1 Specifics

Algorithmically, [Gas89] gives the method in Figure 4.1 as a method of performing list scheduling. This method runs in time based on the size of the DDD. No backtracking is performed in the event of a schedule failure. The only method

```
list_schedule ($DRS$)
{
    while ($DRS \neq \emptyset$)
    {
        choose $op \in DRS$ with highest priority;
        for ($\iota = \Theta(op)_{min}; \iota \leq \Theta(op)_{max}; \iota++$)
        {
            if (pack $op$ in $\iota$)
            {
                $DRS - op$;
                $DRS \cup N\{\forall a, N \in op.succ \mid a \prec N = \emptyset\}$;
                list_schedule ($DRS$);
            }
            else
            {
                continue;
            }
        }
    }
}
⋮
list_schedule (DDD($N$)$\{\forall a, N \mid a \prec N = \emptyset\}$);
```

Figure 4.1: List scheduling

for not creating scheduling failures (of any of the three types mentions in 2.1.2) is avoidance. Sections 4.4 and section 4.5 discuss two methods of avoidance.

Another approach would be to generate all the possible schedules by trying all the possible operations when scanning the list instead of using the first one found and using the shortest found. if list scheduling can produce the optimal schedule, this would guarantee optimality because all possible combinations of forming instructions from operations would be attempted. Algorithmically, see Figure 4.2.

Note this is a recursive routine, in order to simplify its expression. What this routine does is:

1. Places each node in turn in the data ready set into the first available instruction that can hold it.

```
list_schedule ($DRS$)
{
        foreach ($node \in DRS$)
        {
                pack $node$ in $\iota$, the first available instruction;
                $DRS - node$;
                $add\_to\_DRS = N\{\forall a \in DDD, N \in node.succ \mid a \prec N = \emptyset\}$;
                $DRS \cup add\_to\_DRS$;
                list_schedule ($DRS$);
                $DRS - add\_to\_DRS$;
                $DRS \cup node$;
                remove $node$ from $\iota$;
        }
}
⋮
list_schedule ($N\{\forall a, N \in DDD \mid a \prec N = \emptyset\}$);
```

Figure 4.2: Exhaustive list scheduling algorithm

2. Update the data ready set to reflect the packing of the operation.

3. Calls itself recursively with the new data ready set.

4. Places the node back on the data ready set so that it is used while iterating on all the other nodes at this level.

5. Removes the operation from the instruction it was placed in so its resources and encodings are freed.

The routine is called with the sources of the DDD; the original data ready set. If it is unable to place an operation in an instruction (on the basis of a timing conflict), it does not fail. The routine simply continues to check other possible schedules. If no schedule is found, no list scheduling routine could schedule the DDD. This method does not backtrack, it does exhaustive enumeration and searches all the valid final schedules for the shortest.

It is interesting to note this method does not produce all possible schedules. There is a heuristic even here: place a node in the first available instruction. In the presence of (n, ∞) timing on any edge, it is possible for an infinite number of schedules to be generated. While this heuristic seems reasonable, optimal schedules might not be found if delaying an operation would reduce the length of the resultant schedule. All *orderings* of the list are attempted, not all *placements* of the orders.

Simple modifications are needed to change the exhaustive method into a non-exhaustive method (See Figure 4.3.) This routine attempts to schedule the node with the highest priority. If it fails, it tries the next highest node until it finds an operation it can pack. If it fails to pack any of the nodes from the list, the order used created one of the failures from 2.1.2. No backtracking is attempted and other possible schedules are not tried. The routine is a recasting of the while loop version from Figure 4.1; it is recursive for clarity and comparison to Figure 4.2.

Comparing Figure 4.2 with Figure 4.3, few differences exist. Instead of all orders being attempted, only one is tried. This eradicates the need to change the information in the data ready set after a node is packed. It also eliminates the need to remove any operations from their instructions as the first placement is the final placement. If list scheduling is performed by these recursive methods, choosing between them is easily possible during scheduling.

The non-exhaustive recursive method and the while-loop method execute equivalently. Both choose the node with the highest priority to schedule next. They are differentiated by the fact that the recursive version saves each DRS on the stack as it executes. If backtracking or exhaustive methods are used in concert with the list scheduling routine, this is an advantage. If not, the space on the run-time stack of the scheduler is wasted.

4.2 Direction

```
list_schedule ($DRS$)
{
    choose $op\{op, n \in DRS \mid n < op)$
    {
        if (pack $op$ in $\iota$, the first available instruction)
        {
            $DRS - op$;
            $DRS \cup N\{\forall a \in DRS, N \in op.succ \mid a \prec N = \emptyset\}$;
            list_schedule ($DRS$);
        }
        else
        {
            $n = op$;
            continue;
        }
    }
}
⋮
list_schedule ($DDD(N)\{\forall a, N \in DDD \mid a \prec N = \emptyset\}$);
```

Figure 4.3: Non-exhaustive list scheduling

The direction a scheduler traverses a DDD can have a large impact upon its efficacy. Thus far, forward traversals have been discussed. To schedule in the backwards direction, no changes to the algorithms thus far enumerated are required. The change occurs exclusively in the data structure representing the DDD. Here, all the sources become sinks and vice versa, all the predecessors edges become successor edges and vice versa, and all the operations are placed in instructions in a backwards fashion. Upon completion of scheduling, the instruction list is reversed to reproduce the original semantic ordering of the source.

Allan and Mueller [AM88] note the direction of scheduling has a large impact on a per-architecture basis. One direction will succeed in producing a valid schedule in a given architecture much more often than the other. Bias is predicted on the presence of restricted timing within the DDD. Reasons for this bias include:

1. Presence of restricted branch delays. If branches in an architecture have (n, m) (or more likely (n, n)) timing to the end of the DDD, there exist few (or one) instructions in which they may be placed. Reverse traversal will tend to place this type of operation in the correct instruction early in the scheduling process, increasing the chance for a valid schedule. Forward traversal will tend to schedule branch operations late in the process, when much less flexibility is available in the DDD because of the amount of already completely scheduled instructions.

2. Presence of restricted pipe stages. As above with branch delays, (n, m) pipe stages can cause failures when a pipe operation is not instantiated at the proper time. Traversal direction is dependent upon whether

   (a) the inputs of the pipe are latched,

   (b) the outputs of the pipe are latched,

   (c) both are, or

(d) neither are.

3. Use of transient condition code registers within the DDD.

It is also possible for the direction to have an effect on the length of the final schedule without considering the impact of restricted edges. The reason is simple: direction has an impact upon the order nodes are chosen to be placed. This is because the formation of the data ready sets differs between the two directions. The *definition* of data ready is intransigent, the *calculation* is applied to a different set of nodes. In this way, for a given DDD, differing directions of traversal may produce two valid schedules with greatly different lengths. It may be worthwhile to attempt both and choose the shorter. If one direction fails to produce a valid schedule, the other direction certainly should be tried.

4.3 Complexity

List scheduling has a complexity of $O(n^2)$ [LDSM80, Gas89]. This is because it operates on a precedence graph; general precedence graphs have $O(n^2)$ edges [Cof76]. Landskov et al. give another technique for viewing list scheduling's complexity: consider the worst-case DDD, one where no data dependencies exist between nodes and all nodes conflict with each other. Then

- each node is checked $\frac{(n-1)n}{2}$ times to decide which to schedule next, and
- each node is then checked for conflict $\frac{(n-1)n}{2}$ times against the nodes already placed.

This bounds the problem by $n^2 - n$ or $O(n^2)$ (note either bounds the problem by $\frac{n^2}{2}$ or $O(n^2)$.)

Given this complexity bound, can an optimal schedule be found in polynomial time? If list scheduling could generate all possible schedules in polynomial time, the shortest could certainly be chosen in polynomial time. The question is therefore transformed into: can all schedules be generated in polynomial time? The answer is no.

Consider the generation of the data ready set list scheduling chooses operations to be scheduled. To add a member to the DRS, all of the node's predecessors must already have been scheduled (and in some fashion, removed from the graph.) If a node has any unscheduled predecessors, it cannot be added. This operation of finding which nodes to add to the DRS is an example of producing a topological sort in a precedence graph, which has complexity of $O(n^2)$ (topological sorting has $O(\max(nodes, edges))$, and precedence graphs can have $O(n^2)$ edges.) As this is also the complexity of the entire list scheduling technique, list scheduling must be as hard as producing a topological sort of a general precedence graph. As above, calculating conflicts adds to the complexity, but does not change the order. If both resource and encoding conflicts must be checked, complexity can become as great as $3n^2$, still $O(n^2)$. If topological sorting was not required to properly schedule a graph, a method with less complexity might be possible. In Section 2.2.2, generating the number of total orders consistent with a partial order is discussed. An upper bound of $O(n!)$ is given and any process to enumerate the total orderings is said to be #P–complete.

The impact of recognizing list scheduling is topological sorting has several results:

1. Shows that it cannot generate a known optimal schedule in polynomial time.

2. Gives a method for viewing list scheduling, i.e. seeing it as topological sorting.

3. Produces a method for analyzing the algorithm.

4. Demonstrates that for valid input, valid output is *possible*.

Another interesting point is observed: list scheduling's building a data ready set, and thereby performing a topological sort, is a heuristic used to create valid schedules.

The implicit heuristic is: scheduling nodes with no predecessors results in valid orderings more often than scheduling nodes with predecessors.

As noted before, the edges of a DDD do not constrain the order nodes are scheduled, only the order they appear in the final schedule. A topological sort of the DDD reduces the search space of the scheduling problem. This produces several results:

1. It increases the chances of finding a valid schedule. This benefit should not be underestimated.

2. It reduces the chances for finding an optimal or near-optimal schedule. This is because order of placement is highly constrained by the DRS by reducing the number of operations available to be placed in the schedule.

If methods can be found that produce valid schedules without the price of performing a topological sort, scheduling might be simpler. Another advantage could result if a method could be found that explores the search space of total orderings well; better schedules might be produced. In list scheduling, a plethora of heuristics are used to reduce the search space.

4.4 Heuristics

Because list scheduling uses heuristics to prune areas of the search space that appear uninteresting, the heuristics must be chosen with great care so unsearched spaces are truly uninteresting. The choice of giving one operation higher priority than another can have great influence on the final schedule. This is particularly true in the presence of multi-cycle operations. Placing an operation at the wrong time, so its output is delayed, will tend to serialize the code. In architectures where more of the hardware features are visible in order to achieve greater performance, this is counter-productive.

A heuristic often cited [All86, LDSM80, SDX87, Woo78] as one necessary for efficacious list scheduling is that of *critical path.* A critical path in a dag is defined to be a longest path from any of the roots to any of the leaves [Gib85]. It is easy to find a critical path in a dag if all the heights of the nodes are known. Computing the height of any node is simple

- if the node is a leaf, its height is zero,
- else its height is the largest height of its successors, plus one.

To find a critical path, find a largest root node, and follow a highest successor node until reaching a leaf. The path followed will be a critical path.

This definition is correct for unweighted dags, that is, those whose edges are of unit length. With the introduction of weights on the edges of the DDD (denoted as $\delta(e)$ for simple timing or $\Delta(e)$ when more complex timing is present), the definition must be slightly modified. A *schedule critical path* is one with the greatest sum of the edge weights from all the roots to all the leaves. Defining the *schedule height* of a node to be

- if the node is a leaf, its schedule height is zero,
- else its schedule height is the largest schedule height of its successors, plus the length of the edge (either $\delta(e)$ or $\Delta(e)_{min}$) to that successor,

allows the usage of the same routine that finds a critical path to find the schedule critical path. In Figure 4.4, node 1 would have a height of 1 and a schedule height of 39. Node 2 would have a height of 3 and a schedule height of 3. Nodes 2, 3, and 4 would be on a critical path, Node 1 would be on the schedule critical path. Node 1 should, in all likelihood, be scheduled before any of the others, otherwise its operation would happen in serial with the others. If scheduled before, parallel execution is possible. Basing the choice of operations upon schedule critical path instead of critical path would accomplish this. Nota bene: critical path and schedule critical different only if $\delta(e) \neq 1$ or $\Delta(e)_{min} \neq 1$.

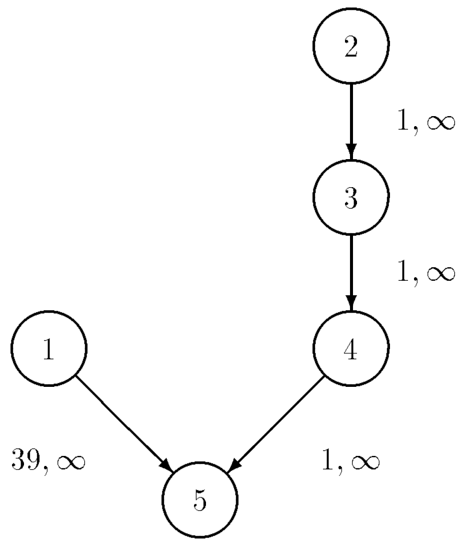

Figure 4.4: Critical path comparison

4.4.1 Specifics

Certainly, a vast number of heuristics are available to reduce the search space of list scheduling. The search space is condensed by choosing one node in the DRS that appears most promising for generating a short, valid schedule. In order to get a short schedule, the schedule critical path is the most important priority heuristic. This is because it defines the lower bound for the length of the schedule (the critical path technique of Section 1.2.1 emphasizes this point well.) All other nodes might or might not have an impact on the final length; those on a critical path will [1].

An example where critical path does not produce a valid schedule is shown in Figure 4.5. Nodes 1, 3, 4, 5, and 6 form a critical path for this DDD. If the resource usage for node 2 conflicts with all the nodes on a critical path, this DDD will not be properly scheduled. Nodes 1, 3, 4, and 5 will be placed first. Then node 2 will attempt to be placed; no instruction is available with enough of the correct type of resource and the instruction list is limited in length by the restricted timing from node 5 to node 6. If node 2 had been placed before node 3, the DDD could be scheduled.

Another heuristic often used to produce valid schedules is that of raising the priority of nodes having restricted timing on successor edges. The logic behind this is to first place nodes more "difficult" to schedule. Nodes with unrestricted timing only depend upon being data ready for placement [2].

In Figure 4.6, the critical path and the restricted successor heuristics form competing, erroneous heuristics. Assume node 2's resources conflict with those of nodes 3, 4, and 5. If the critical path heuristic is given the most weight, followed by

[1]Note that there may be multiple critical paths, i.e. more than one longest path from the sources to the sinks.

[2]As mentioned before, list scheduling uses the data ready condition as its foremost priority heuristic.

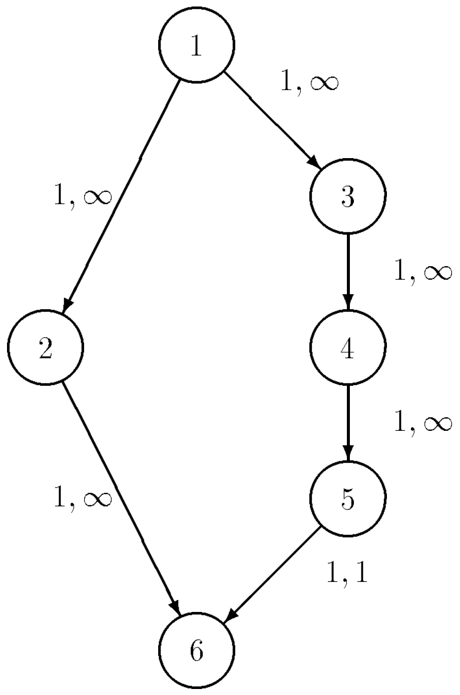

Figure 4.5: A DDD where the critical path heuristic fails

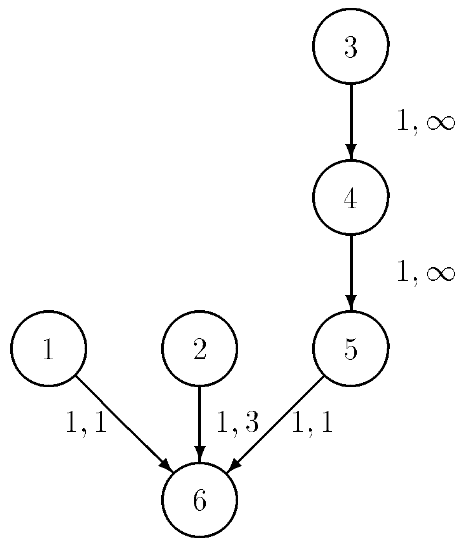

Figure 4.6: A DDD where the heuristics compete.

the restricted successor, nodes will be scheduled in the following order: 3, 4, 5, and 1. Node 2 will not be able to be placed due to its conflicts with node 3, 4, and 5. If the heuristic importance is switched, nodes will be scheduled in the following order: 1, 3, 4, and 5. Again node 2 is unable to be scheduled. Only if node 2 is scheduled somewhere between nodes 3, 4, 5, and 6 will a valid schedule be achieved.

As can be seen, developing a set of heuristics that will guarantee a valid schedule for any valid DDD can be challenging. Further, the most useful heuristics for assuring validity during list scheduling vary from architecture to architecture. This results from architecture-dependent difficulties: one may have a restrictive branch delay while another may have a synchronous pipe that does not latch its output. Differing machine features make the generation and reuse of heuristics difficult when a machine-independent scheduler is desired.

4.4.2 Enumeration

In ROCKET, many heuristics were developed in order to achieve valid schedules for a variety of target architectures. The following list exists at the current time:

1. `height` – distance from the leaves of the DDD.

2. `on_critical_path` – node is on critical path.

3. `on_compaction_critical_path` – node is on schedule critical path.

4. `lexical_order` – ordering of nodes from source [3]. Fisher [Fis79] shows that program lexical order is not a good metric for list scheduling priority, our experience agrees with this.

5. `branch_node` – the node is a branch node, especially useful in the presence of delayed, restricted branching mechanisms.

6. `schedule_range` – the range of $\Theta(op)$, a measure of the flexibility of placement.

7. `resource_usage_of_this_type_in_dag` – the amount of use of this node's resource in this DDD. The more contention for resources, the earlier a node should be placed in order to free the resource as soon as possible for reuse.

8. `number_of_used_and_defined_resources` – as above, nodes that use more resources than others should be scheduled so they do not interfere with others needing those resources.

9. `least_recently_used_resource` – a method of forming round-robin reference to resources.

10. `field_usage_of_this_type_in_dag` – as with resources, try to minimize field conflicts.

[3]Easy in ROCKET because node numbers are generated in lexical order.

11. `number_of_fields_used` – as supra.

12. `least_recently_used_field` – as supra.

13. `number_of_successors` – the more successors a node has, the earlier it should be scheduled, allowing its successors to become data ready as early as possible. This exposes more parallelism to the scheduler.

14. `number_of_restricted_successors` – the more restricted successors a node has, the earlier it should be scheduled so timing is more flexible within the DDD. Once timing becomes increasingly limited, restricted successors become harder to place.

15. `total_of_restricted_successors` – total of $\Delta(e)$ for all restricted successors.

16. `shortest_restricted_successor` – restricted successors having a smaller $\Delta(e)$ reduce flexibility, and therefore the possibility for valid scheduling, diminishes.

17. `distance_from_succs` – a measure of how restricted the edges to the successors are.

18. `average_restricted_successor` – the average of $\Delta(e)$ for all the restricted successors.

19. `number_of_predecessors` – as with successors, increase priority of those nodes with many predecessors.

20. `number_of_restricted_predecessors` – as supra.

21. `total_of_restricted_predecessors` – as supra.

22. `shortest_restricted_predecessor` – as supra.

23. `distance_from_preds` – as supra.

24. `average_restricted_predecessor` – as supra.

All of these have proven useful in different circumstances in ROCKET for a given DDD.

ROCKET's scheduler has the ability to tune the weight given to each heuristic using *Discriminating Polynomial Selection* as discussed in Allan and Mueller [AM88]. Each heuristic value listed above, for a given DDD, is multiplied by a constant specified in a polynomial form. This weighting information can be contained in the architecture description file so it may be easily varied on a per-machine basis.

4.4.3 Update Interval

An issue exists as to when to update the priority weightings on the node in a DDD. At least two possibilities exist:

1. calculating the weights once, before the list scheduling algorithm begins (denoted *static* weighting), and

2. calculating after each node is placed in an instruction (denoted *dynamic* weighting).

Certainly, the first method requires the least amount of computational time. It also gives a good estimate of the overall priorities present in the DDD. Its difficulty is that a DDD does not remain static throughout the scheduling process. As operations are placed into instructions, they are removed from the DDD, changing the shape and makeup. This is not reflected in the priorities if they are calculated only once.

One important heuristic that can change during scheduling is critical path. If nodes are favored from a critical path, chances are favorable that that path will become shorter than another remaining in the DDD. A simple example is shown in Figure 4.7. Originally, nodes 1, 2, 3, and 4 are on a critical path. After nodes 1 and

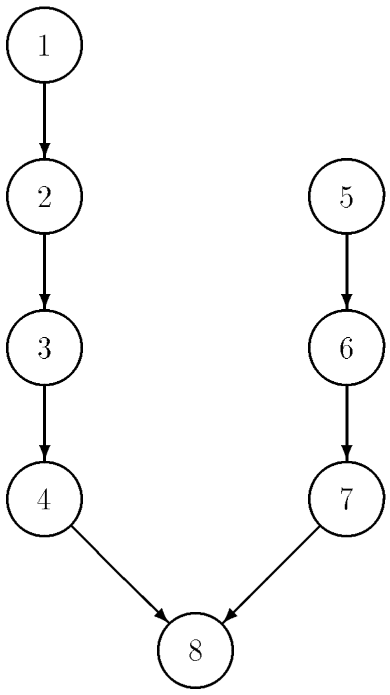

Figure 4.7: Critical path competition

2 are scheduled, nodes 5, 6, and 7 constitute a critical path. Another particularly important heuristic is that of schedule range ($\Theta(op)$). Using the absolute timing algorithm from Section 2.5.1 and Figure 4.8, node 3 will originally have $\Theta(3) = 1, \infty$. After the placement of node 2, node 3 will have $\Theta(3) = n, n$ where n is one greater than the scheduled value of $\Theta(2)$. This is a much tighter bound on the range of node 3 and should be reflected in its priority.

The decision as to when to generate the priorities on the nodes is one that must be considered carefully when producing a list scheduler [4]. Empirical results usually drive the decision; if the static method works well, no reason exists to use the dynamic method.

[4]The ROCKET list scheduler can easily choose between either of the two methods mentioned.

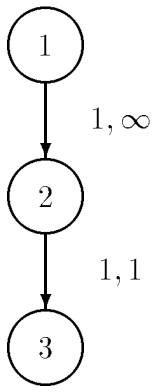

Figure 4.8: Need for dynamic schedule range calculation

4.5 Foresight

A powerful method to increase the likelihood of generating a valid schedule called *foresight* is introduced in Wijaya and Allan [WA89]. The procedure checks to see whether, after placing an operation in an instruction, all (direct and indirect) successors with restricted timing can be "easily" placed in their respective instructions. If so, the operation under consideration is placed. If not, the operation is moved to its next valid instruction and foresight is repeated. If no valid instruction can be found, the schedule generated thus far is deemed invalid. This is not a backtracking algorithm; on the contrary, it looks forward checking for validity of placing a node before the final decision is made. A routine, $foresight\_ok()$ (adapted from [WA89]), that checks all nodes restricted by a node's placement is shown in Figure 4.9. An outline of a routine to schedule a single operation, based upon $foresight\_ok()$, is found in Figure 4.10.

Several routines are assumed to exist in the algorithm presented

- $can\_pack(op, \iota)$ checks the resource and encoding conflict to assure that op can be placed in ι, and

```
foresight_ok ($constrained$)
{
          $\forall op \in constrained$
          {
                    $\forall \iota \in \Theta(op)$
                    {
                              if (can_pack ($op, \iota$)
                              {
                                        ok = TRUE;
                                        break;
                              }
                    }
                    if (!ok)
                    {
                              break;
                    }
          }
          return ok;
}
```

Figure 4.9: Foresight_ok() routine

- *update_absolute_timing*() assures that all the absolute times on all nodes are not violated.

Note that both direct and indirect successors are checked with these routines. The absolute timing algorithm from Section 2.5.1 assures that all nodes affected by the placement decision are updated. Note also that routines to both add and remove an operation in an instruction are assumed to exist. This can be complicated by the machine having multiple instruction formats; care must be taken to assure that as operations are added and removed from instructions, the instruction format changes to reflect the changing requirements.

4.5.1 Hindsight

Because foresight was developed for a list scheduling system, there was no reason to check for conflicts with predecessor node timings. Checking a predecessor does

```
schedule_op ($op$)
{
    $\forall \iota \in \Theta(op)$
    {
        if (can_pack ($op, \iota$))
        {
            $op \in \iota$;
        }
        else
        {
            continue;
        }

        update_absolute_timing ();
        if (foresight_ok ($\forall a \{ a \in DDD(N) \mid \Theta(a)_{max} \neq \infty \}$))
        {
            break;
        }
        else
        {
            $\iota - op$;
        }
    }
}
```

Figure 4.10: Schedule an operation using foresight

not make sense in a list scheduler: there must be no predecessors of a node under consideration for if there are, the node is not data ready. In different schedulers, this condition can be removed, requiring the addition of *hindsight* [5]. Hindsight checks whether, after scheduling an operation, all of its restricted predecessors can be easily placed into their respective instructions. The basic method does not change appreciably; it is only an extension to foresight to add hindsight. A simple recursive foresight and hindsight routine is shown in Figure 4.11 [6].

4.5.2 Incremental

Because a substantial amount of information generated during each pass of the foresight routine, Wijaya and Allan add the ability to keep data from one pass to another, resulting in *incremental* foresight. The schedule ranges for operations have a form of temporal locality, i.e. once they are constrained, they remain constrained. The constrained set does not vary greatly between iterations. Rules may be formed specifying when schedule ranges are affected by placement of operations and must be updated. When incremental foresight cannot reuse information from a previous pass, non-incremental foresight is used. In this way speedup is achieved and incremental foresight fails only when foresight would.

Foresight certainly helps in the generation of valid schedules by checking the validity of operation placement before committing to it. An assumption made by foresight is successor nodes will be placed in either the instructions foresight checks or other instructions can be found to hold them. This assumption can be invalidated. If the successors cannot be placed in the instruction examined, foresight breaks down. For example, if instead of the successor nodes being scheduled into the instructions examined, nodes from another part of the DDD are scheduled into

[5]Giving the scheduler 20/20 vision?

[6]ROCKET's scheduler implements foresight/hindsight.

```
foresight_ok ($node$)
{
        $\forall op\{op \in node.succ \mid \Theta(op)_{max} \neq \infty\}$
        {
                $\forall \iota \in \Theta(op)$
                {
                        if (can_pack ($op, \iota$)
                        {
                                ok = foresight_ok ($op$);
                                if (ok)
                                {
                                        break;
                                }
                        }
                }
                if (!ok)
                {
                        return FALSE;
                }
        }
        $\forall op\{op \in node.pred \mid \Theta(op)_{max} \neq \infty\}$
        {
                $\forall \iota \in \Theta(op)$
                {
                        if (can_pack ($op, \iota$)
                        {
                                ok = foresight_ok ($op$);
                                if (ok)
                                {
                                        break;
                                }
                        }
                }
                if (!ok)
                {
                        return FALSE;
                }
        }
        return TRUE;
}
```

Figure 4.11: Recursive foresight

those instructions (either due to higher priority or the successor nodes not being data ready), the validity of the examination is obviated.

4.6 Lookahead

Foresight performs a lot of work checking the validity of placing a node. This work can be wasted in the event that the schedule checked does not become the actual schedule. A method termed *lookahead* was developed to enhance the benefits of foresight.

When the scheduler does not place the constrained nodes in the instructions that foresight determined will produce a valid schedule, work is lost. Why not perform the scheduling of those nodes immediately? Within the framework of list scheduling, the reason is simple: those nodes are not data ready. If the data ready criterion is removed, what is the impact upon forming a valid schedule? None.

As noted before, the edges in a DDD only limit the ordering in the final schedule. So long as this order is preserved, the method of placing the nodes is irrelevant. The absolute timing algorithm assures that nodes are placed such that their range is valid in the final schedule. The value $\Theta(op)$ for a node, calculated by the absolute timing routine, specifies the range in the final schedule where an operation can be placed. Because the foresight routine examines instructions in this range for node placement, foresight finding a valid schedule guarantees validity of that placement in the final schedule. An alternative view is that not only a node *can* be placed where foresight predicts, it *should* be placed there. If it is not, scheduling with foresight can fail on a placement it previously judged valid.

A decision must be made as to whether to pack only the surrounding nodes with $\Delta(e) = (n, n)$ (equivalently $\Theta(op) = (a, a)$), or additionally to pack the surrounding nodes with $\Delta(e) = (n, m)\{n, m \mid n < m < \infty\}$ ($\Theta(op) = (a, b)\{a, b \mid a < b < \infty\}$.) In the first case, no choice exists as to when to pack the nodes, they must be placed in instruction $\iota + n$. The second case contains more flexibility and requires the analysis of a tradeoff. Having lookahead place them will result in a larger chance of generating a valid schedule, similar to the improvement that lookahead has to foresight. If lookahead does not place the $\Theta(op) = (a, b)\{a, b \mid a < b < \infty\}$ nodes, the scheduler may be able to produce a more compact final sequence. This tradeoff varies with the amount of flexibility in the operation's schedule range, in the current DDD, and in the architecture, making it difficult to analyze the tradeoff universally.

Lookahead also increases the speed of list scheduling by removing the number of nodes that can be data ready in the graph. There is no need to change the definitions or implementations of any routines within the list scheduler. The definition for data ready remains the same. The only change applies to the foresight routine shown in Figure 4.11: after the recursive call to foresight_ok(), a call to pack(op, ι) is made to affect the placement. This assures that all restricted successor operations can be placed in instructions. A further extension to lookahead, similar to hindsight, is *lookbehind* [7]. Because lookahead places nodes that are not data ready, it is possible for a node under consideration to have restricted predecessors. With the same logic that produced the packing of restricted successors, there is no reason to delay the placement of restricted predecessors. Any failure to place either a restricted successor or predecessor would result in a failure later in the scheduling process, reducing the time spent on an infeasible schedule.

It is important to understand that lookahead is still an avoidance technique, albeit a more powerful one than foresight, itself more powerful than nothing at all. There still is the probability that valid DDDs exist that cannot be scheduled due to poor choices made by the node priority algorithm. This is inherent both in the use of avoidance techniques and in only searching a small subspace of the possible solutions [8].

[7]Producing *LOOKOUT!* scheduling?

[8]ROCKET's scheduler contains both lookahead and lookbehind scheduling.

4.7 Summary

Several equivalent methods of list scheduling have been presented and compared. The direction of traversal of the graph has a definite impact upon both the length and validity of the schedule. The complexity of list scheduling is bounded (at least) by the complexity of producing a topological sort of an unconnected graph, i.e., $O(n^2)$. The presence of other constraints on the nodes will increase the complexity. Foresight (with the possible addition of hindsight) increases the probability of producing a valid schedule by checking the impact that placing a node with restricted successors (predecessors) has on the final schedule. Lookahead (lookbehind) greatly increases this probability by actually placing restricted nodes in the final schedule in their schedule range, obviating the chance that another (unrestricted and therefore unchecked) node will preempt their placement. Lookahead also reduces scheduling time by placing nodes that are not data ready and by stopping the production of infeasible schedules.

Chapter 5

GENETIC ALGORITHMS AND INSTRUCTION SCHEDULING

Genetic algorithms (GAs) manipulate populations of strings representing the parameterization of the optimization problem. The strings correspond to chromosomes or genotypes in biological terms. A mapping exists from this representation to the phenotype of the actual solution. GAs use a form of selective pressure to encourage over-achieving and discourage under-achieving strings in the population. A string's chances of reproducing correspond to its performance in the current environment. This is an easily understandable method and it produces robust searches of difficult parameter spaces as demonstrated by Holland and others [Hol75, DeJ86, Gol89].

5.1 Foundations

Parameters are usually encoded into some form of binary representation. This representation is then used for subsequent operations and evaluations. Consider the string: `1100101001110110001`. This could represent an integer, a fixed or floating point real, or any other relevant model of the parameters to be optimized. Multiple parameters are simply appended together. The initial population is usually generated by creating random strings.

To perform recombination, the basis for most genetic adaptation in nature, consider also the string: `xyyxxyxyxyxxxyyyxxy` (with `x` for `0` and `y` for `1`) and some number of break points. The genetic material from one string is then swapped between those break points with the corresponding material from the other. An example with two break points is

```
11001 \/ 01001110110 \/ 001
xyyxx /\ yxyxyxxxyyy /\ xxy
```

resulting in the two children

```
11001yxyxyxxxyyy001
xyyxx01001110110xxy
```

Although a single break point is usually used in discussions of GAs, two have been empirically shown by Booker [Boo87] to produce better results.

Another operation in the reformation of strings is mutation. This is accomplished by randomly toggling some of the bits in the offspring. This creates genetic diversity. It has been found, in the general case, that mutation rates should be kept low (less than 5%) for best exploitation and least disruption of the information present.

In standard GAs, all the strings in the population are reformed during a generation. Parents are crossed on the basis of their performance in comparison to the average fitness of the population and mutation is allowed to occur on the offspring. Selective pressure is provided by the fitness measure; the differential need not be great to achieve good results. Both selective pressure and initial population sizes may be tuned to match the problem space. The type of crossover and rate of mutation needs selection based on the problem type.

To relate the encoding with the sampling of hyperspace, consider a string of length three. This gives the ability to represent a three-dimensional hypercube. The string `011` represents a corner of the hypercube. Edges have one of the bits as a "don't care," i.e., `01*`. Faces have two "don't cares:" i.e., `0**`. The entire space can be expressed by a complete "don't care string:" i.e., `***`. Strings containing a "don't care" in some position are termed schemata. Figure 5.1 demonstrates schema sampling in a three-dimensional hypercube. In general, each binary encoding corresponds to one corner in the hypercube and samples $2^L - 1$ different hyperplanes in the search space where L is the length of the binary encoding. This is the idea

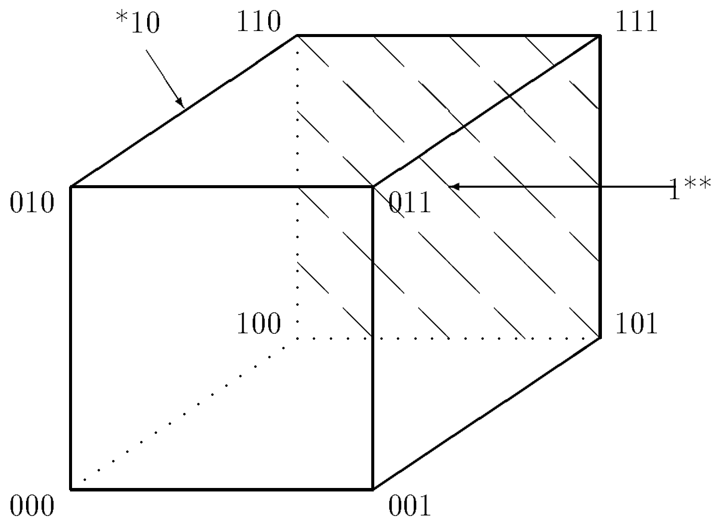

Figure 5.1: Sampling in a three-dimensional hypercube

of *intrinsic parallelism* whereby one string samples the productivity of many hyperplanes [Hol75]. Schema theory indicates that individual hyperplanes will increase or decrease their representation in a population based upon their relative fitness in that population when reproduction and recombination are applied.

The more diverse the original population, the more global the search. The search does not avoid or escape from local minima; it does a global search where local minima are ignored in favor of higher-valued strings. If a local minima is found to be best, it will tend to be competitive with all areas of the space searched. It has been shown that if an area in hyperspace has above average performance and is sampled by a schema in the population, that area's representation will increase within the population. It has been calculated that for the processing of N structures per generation approximately N^3 schemata are sampled (intrinsic parallelism).

The ability to sustain search is dependent upon the genetic diversity in the population. When a population lacks diversity, new areas of the space are not examined. Mutation can be used to drive the search into these unexamined areas.

However, a fixed level of mutation has been shown to disrupt the search early and then fail to provide enough diversity in the later stages. Adaptive mutation increases the mutation rate based on the homogeneity of the population and gives better performance.

5.1.1 GENITOR

The GENITOR GA program, developed by Whitley [WSF89, WSS90], has some differences with "standard" GAs that appear to increase performance. It does not replace the entire population with each generation. Instead it probabilistically chooses two parents to reform into two offspring. Recombination and mutation occur, then one of the offspring is discarded randomly. The remaining offspring is placed in the population according to its fitness in relation to the rest of the strings. The lowest-valued string is discarded. This keeps high-valued strings within the population, directly accumulating high-performance hyperplanes. It also bases the reproductive opportunity upon rank with the population, not upon a string's fitness value in comparison with the average of the population, reducing the impact of selective pressure fluctuation. It also reduces the importance of choosing a proper evaluation function for fitness in that the difference in the fitness function between two adjacent strings is irrelevant.

A recent improvement has been made to GENITOR in the form of a distributed genetic search. This is not simply running subpopulations on different processors but also occasionally swapping the best members from neighboring subpopulations. In this way, subpopulations are somewhat independent while still sharing some information. The search speed is increased over the serial method, allowing the exploration of larger populations in a given amount of time. Larger problems are also approachable with the distributed genetic search. It is also a more robust method, producing better results without as much sensitivity to population size or selective pressure. The improvements are attributed to the maintenance of genetic diversity by the interacting subpopulations.

5.2 Results from Related Work

Realistic scheduling problems are difficult to represent using traditional mathematical techniques. As a result, more traditional optimization methods are difficult to apply. GAs are capable of searching ill-structured spaces and also provide a global method of search.

Genetic algorithms have been recently applied to three areas of interest with good result: the traveling salesperson problem, job-shop scheduling, and flow shop scheduling. The following three subsections discuss the motivation and results reported by Whitley and others [WSF89, WSS90, CS89]. High quality solutions have been found for each of these problems. The results are not based upon heuristics or local optimization information. As is common with GAs, a method of ranking the current population is required. In these problems, this is a simple task of summation to find the length of each member of the population. This task is, if anything, easier for instruction scheduling because the length of scheduled code is trivial to find.

5.2.1 The Traveling Salesperson Problem

This problem involves finding the shortest Hamiltonian path or cycle in a graph where nodes represent cities, and edges represent the paths and distances between two cities. The optimal solution is one having the least distance and yet visits all the nodes (cities). The TSP is an example of an NP-hard problem; all known methods for finding an optimal solution require searching a space that grows exponentially with the number of nodes in the graph.

When using GAs to perform TSP optimization, maintenance of good subtours present in the parents is desirable. This leads to shorter overall tours in the children. A recombination operator that preserves edges will exploit the most amount of information from the parents. In Whitley et al. [WSF89, WSS90] this is achieved by making an edge map and having the recombination operator use this map. It is possible to show that this method changes the sampling rate of hyperplanes in the

N-dimensional hypercube (where N is the length of the encoding of the problem) in favor of high fitness hyperplanes. This method of optimization does not use any information on the distance between cities, only the distance of the overall tour. The use of this type of metric for the evaluation function is important because of its simplicity and applicability to many forms of scheduling and sequencing.

The published results from this method are impressive. The best known solution for a certain 30-city TSP problem is 420. With a serial version of GENITOR, a population of 200 and allowing up to 70,000 recombinations, the GA found the best known solution in 28 of 30 tries. It found a solution of 421 with the other two tries. With the distributed version, it found the best known solution in 30 of 30 attempts. On a 105-city problem, with the distributed version and 10 subpopulations of 1000 each and allowing for 200,000 recombinations in each subpopulation, the best known value of 14,383 was found 15 out of the 30 times. In the remaining 50% of the time, the solutions were within 1% of the best known. More recent enhancements to the algorithm have further improved performance (solutions of 420 on 45 of 45 tries) while reducing search time by more than 50% (a maximum of 30,000 recombinations).

5.2.2 Job-Shop Scheduling

In scheduling machine usage on a job-shop floor, the flexibility is usually found in the sequence of jobs presented to the line. There are fixed setup, idle, and active costs for machines. A strict amount of product needs to be produced in order to meet the demand. Therefore, the approach taken is viewing the optimization problem as one of sequencing the types of jobs presented to the first machine in the line. This can then be viewed as a problem similar to the TSP. The sequence is then evaluated on the basis of total cost and the GA performs the search accordingly.

In Whitley et al. [WSS90] a detailed description of an actual production line in use at Hewlett-Packard in Fort Collins, Colorado is discussed. It contains 6 workcells (groups of machines) in sequence, each performing a specific operation. Each has a single input and a single output queue. Every workcell contains two identical machines operating independently. The machines have costs associated with the various tasks they perform. Twenty different types of products are produced by the line.

Two different approaches were tried:

1. a strict FIFO where the GA controlled the sequence of jobs presented to the line, and

2. a HYBRID where the GA attempted to optimize the initial sequence and a greedy algorithm attempted to reorder jobs in the line for maximal machine usage.

The FIFO task appears more difficult because it is not allowed to reorder jobs within the line. Both models try to keep all machines busy all the time. Surprisingly, the FIFO model produced better sequences, i.e. kept more machines busy more of the time resulting in lower cost. It also produced results faster than the HYBRID method. The results were not greatly different (approximately 3%) but the implications of not having to use a greedy, heuristic-based method (requiring more code and effort) are great. It is thought that FIFO worked better than HYBRID because of its ability to directly control all the global information. What appears good from a local greedy point of view is not always good from a global perspective. The FIFO is also probably a more realistic model of many scheduling tasks.

The implications of using only overall cost for the evaluation function are also significant. This allows for the application of GAs to other scheduling or sequencing problems that are not well-structured enough for traditional optimization methods. The only two areas that vary between problems are:

1. the representation scheme and

2. the evaluation function.

It has been shown empirically that these are easy to vary.

[WSS90] concludes "that one should not attempt to build heuristics into the scheduling system, but rather let the genetic algorithm do the work." By doing this, the implementation is simplified, the computational complexity is reduced and the quality of the results are not damaged.

5.2.3 A Warehouse/Shipping Scheduler

In Starkweather et al. [SMM$^+$91], the development of a system to schedule the production of beer at a large local brewery [1] is discussed. A simulator was developed that mirrored the actual production constraints of the brewery. The objective of the scheduler was the efficient allocation of orders to loading docks based on some fixed production cycle. Production occurs 24 hours a day on multiple lines, each line capable of producing a certain beer type. There exists different labeling and packaging possibilities that complicate scheduling. Data were available on the different flow rates, start and stop times, and product type for each line. A set of orders to be filled was also available. The goal was the reduction of average inventory (costly in many different ways) while filling as many orders as possible. The genetic algorithm manipulated strings representing the sequence beer orders were processed. Each sequence was run on the simulator to produce an evaluation, with the original population generated randomly. Results from simulations demonstrated that, for improvement, preserving relative order was more important than preserving adjacency in the population's strings. This is not surprising as adjacency has little meaning in this problem's context; items adjacent in the sequence were probably unrelated, occurring on separate production lines. The order of tasks scheduled for any particular line determines the final output.

[1] Coors Brewery, Golden Colorado.

Using the efficient search possible with genetic algorithms and a modern workstation, good daily schedules can be generated in five minutes. This is considered real-time in the context of the problem. Line down-times, priority orders, and the like can be adjusted for on the fly.

5.3 Instruction Scheduling using Genetic Algorithms

The encouraging results presented above drove the use of GA's for instruction scheduling. With the increasing complexity that expressing parallelism in both the source code and in the architecture places on scheduling, and the concomitant increase in the importance of producing good code, previous methods were found too weak. The search space is certainly discontinuous, with good schedules adjacent to bad ones, so that methods of local improvement (such as hill-climbing) cannot be relied upon to find globally competitive solutions. Another important feature provided by the removal of local methods is the time reduction realized by not performing the analyses required to drive these methods. In the case of instruction scheduling, the need to analyze the DDD and produce the metrics the heuristics use is obviated. Finally, the time taken by genetic algorithms can be directly controlled. The time taken by other global search methods, such as exhaustive search, cannot be.

A distinction before some specifics. In Chapter 2, the problem of *scheduling* was stated as that of placing tasks with precedence constraints on processors for execution. The precedence constraints must not be violated by the scheduler. Given a method of placing tasks in a schedule that ensures validity, the problem may be changed into one of *sequencing*. Sequencing specifies the *order* tasks are scheduled; another method generates the schedule based on that order.

5.3.1 Methods

To use genetic algorithms for a particular problem domain, only a few ancillary functions must be defined. One of these is an evaluation function that ranks the fitness of a string from the population. Choosing a proper function, i.e., one representing a string's relative worth in the population without inordinate bias, is important. For instruction scheduling, a minimization problem, the result of the evaluation function must reflect the length of the final schedule that a member of the population generates. A difficulty encountered is that not all strings will produce valid final schedules. Failures will occur when a conflict arises (i.e. timing, resource, or field) due to a string's ordering. It is not surprising that certain orders will fail to produce valid schedules for any given DDD; in fact, the impact of ordering on the production of valid schedules is (over)emphasised in previous methods of instruction scheduling.

One possible solution is to give failing strings some predetermined "bad" (large) value. Difficulties with this method include

- all failing strings will be given the same value, no matter how close they got to producing a valid schedule, and
- the evaluation function will produce undesirable bias among the poorly performing strings, i.e., they will all appear equivalent.

This will interfere with the natural selection process of the genetic algorithm because actual performance is not reflected by the evaluation function. After consideration, the method selected performs a "worst-case" evaluation when a string fails to produce a valid schedule. This evaluation is produced by assuming all unscheduled operations have no parallelism available in them, necessitating their serial placement. The calculation of the evaluation function is then trivial; it is the number of instructions containing operations so far, plus the length of the path containing the serial ordering of all the unscheduled operations. This produces a good estimate in the event of schedule failure; those schedules with more operations placed will receive a better evaluation. It also produces an exact evaluation in the presence of a valid schedule.

Six different operators were studied. Recall that in GENITOR, two parents are chosen for an operation, and one child at a time results. A summary generated from Starkweather et al. [SMM+91] of each operator follows.

Order Crossover #1: Developed by Davis [Dav85], this is very similar to "traditional" genetic crossover. The child inherits genes from between two crossover points from one parent. The elements remaining unmentioned in the child that exist in the alternate parent are placed after the final element from the first in the order they occur in the alternate parent. Elements in the alternate parent already present in the child are simply skipped. This method preserves the order, adjacency, and absolute position from the first parent and relative order of the alternate. An example follows.

```
Parent  : a b c d e f g h i j
Cross   :     |       |
Parent  : C F A J H D I G B E
-----------------------------
Child   : I B c d e f g A J H
```

Order Crossover #2: Developed by Syswerda [Sys90], this method is similar to Order Crossover #1. Instead of choosing two crossover points, however, several crossover positions are chosen randomly, and the order of these elements in one parent is preserved in the offspring. All other elements are mapped directly from the other parent.

```
Parent  : a b c d e f g h i j
Cross   :     | |     |   |
Parent  : C F A J H D I G B E
-----------------------------
Child   : a j c d e f g h i b
```

Partially Mapped Crossover (PMX): Goldberg and Lingle [GL85] detail this operator. Again, two crossover sites are selected randomly and the information between the sites in one parent is copied directly to the child. The genes in the alternate parent between the crossover points are mapped to the child by placing them in the spot corresponding to the genes placed by the first parent. This maps genes between the crossover points in the alternate parent to locations outside of the crossover points. Any remaining genes, i.e., those not contained between the crossover points in either parent are inherited by the child directly from the alternate parent. As with the first order operator, the crossed section's genes retain order, adjacency, and position from the first parent.

```
Parent  : a b c d e f g h i j
Cross   :     |     |
Parent  : D I J H A G C E B F
-----------------------------
Child   : H I c d e f J A B G
```

Cycle Crossover: Developed by Oliver et al. [OSH87], this operator preserves absolute position from the parent sequence. A starting position is chosen and that element is placed into the child at the same position. The gene at this place in the alternate parent cannot be placed in this position, so the gene is found in the first parent and mapped to the child at the position it is found. This cycle is repeated until the initial item from the first parent is encountered in the alternate. Remaining genes are inherited from the alternate. Cycle crossover maintains the order from one or the other parent without disruption.

```
Parent  : a b c d e f g h i j
Cross   :       |
Parent  : C F A J H D I G B E
-----------------------------
Child   : C b A d e f g h i j
```

Position-based Crossover: Also developed by Syswerda [Sys90], it is also similar to Order Crossover #1. Some number of locations are chosen at random, and those elements from the first parent are copied directly to the child. The remaining elements are inherited in the order they appear in the alternate, ignoring elements already placed. This maintains some relative order from both parents.

```
Parent  : a b c d e f g h i j
Cross   :   | |     |     |
Parent  : C F A J H D I G B E
----------------------------
Child   : A b c J H f D G i E
```

Edge Recombination: Whitley et al. [WSS90] developed the edge recombination operator for use in problems where adjacency information is paramount, as is the case in the Traveling Salesperson Problem. An edge table is built representing the paths into and out of a city. A child is formed by choosing elements from the edge table and following the edges from each element. This assures both a valid tour of the cities and the preservation of adjacency information.

Starkweather et al. demonstrate that each operator will perform differently for each given problem domain. The performance difference can be measured in the speed of convergence to a good solution. For example, edge recombination finds good solutions more rapidly on the TSP while performing more poorly than the others on scheduling problems.

The conditions for stopping search must be examined. Halting is certainly predicated on finding one valid schedule, a possibly non-trivial requirement. Because this requirement is present in all forms of instruction scheduling, it is not viewed as a detriment to a GA-based approach. Another condition for stopping can be the closeness to the theoretical best for the DDD. On “easy” DDDs (with simple timings and dependencies), the theoretical best can be achieved with regularity. Therefore, this condition should be checked to determine the credibility of continuing the search. Another bound easily placed is the number of recombinations performed. Time constraints certainly can be used to generate a value for this bound, unfortunately, a direct relationship does not exist between time spent and the quality of the resulting schedule; an optimal schedule may be found in the first generation, or not found after innumerable generations. Such is the case with all non-deterministic optimizers; unless a solution matches a known theoretical best, a stopping condition cannot be reliably specified [2]. Genetic algorithms will generate solutions competitive with all areas of the solution space searched. Increasing the number of recombinations, and to a lesser degree the population size, will tend to produce “more optimal” solutions. This provides motivation to perform as many recombinations as possible.

The number of generations should therefore be related to the relative difficulty of producing an optimal schedule for a given DDD. DDDs with few simple operations do not require as many generations to find good schedules as do those with many complex operations. Basing the number of generations on a low-order polynomial has proven effective. The size of the population is based on a different lower-order polynomial. The homogeneity of the population is an indirect indicator of convergence. If all solutions are similar, either a competitive solution has been found or unvisited areas of the search space contain better answers. In the first case, the search may be halted. In the second case, mutation should be used to expand the area of the search space covered. It is difficult to differentiate these two conditions.

5.3.2 Approaches

Two genetic approaches to the instruction scheduling problem are outlined in the following sections. Both are based on the manipulation of strings of non-repeating integers. This representation is consistent with those used in the TSP and shop scheduling problems previously mentioned. All populations are randomly

[2]Remember that theoretical best and optimal are not the same. The former specifies the best without the presence of any additional constraints, the latter considers those extra constraints.

initialized. For each member of the initial population, the evaluation function must be executed in order to create a sorted gene pool. It is possible that a member of the initial pool will generate the theoretical-best solution, removing the need to perform any recombinations. This situation occurs most frequently in DDDs easy to schedule. If information about the presence of possibly good-performing strings is known (from another scheduling method), those strings can be added as seeds in the population. The evaluation function is then unnecessary for the performance of the string is known. One of the possible difficulties generated by seeding is the corresponding reduction in the amount of space searched if the population is constrained by too many similar solutions. In the work presented here, no attempt was made to seed the initial population.

A method for improving the overall performance of a genetic algorithm repeatedly restarts the process from the beginning. This essentially “randomizes” the entire procedure by re-initializing the gene pool, and re-choosing the crossover points. This effectively infuses the population with new, possibly more robust, genetic information. Multiple runs with different populations may result in a more efficient search than simply increasing the number of generations. This restart can be viewed as a large-scale mutation operation (only those equivalent between populations are free of disruption) and can therefore be used when the homogeneity of a population becomes too great.

With List Scheduling

The first approach used a genetic algorithm in concert with a list scheduler. Several immediate benefits accrued due to this method:

1. the use of existing list scheduling technology (requiring no additional scheduling code),

2. the use of an already-understood scheduling method, and

3. the use of a method similar to both Syswerda's job-shop scheduling and Starkweather et al.'s brewery scheduling approaches.

These benefits allowed comparisons to be made against existing results, both to verify the validity and efficacy of the resultant schedules.

The strings in the population represented the priority ordering of the nodes in the DDD. As nodes in ROCKET's DDDs are numbered consecutively from $1 \ldots n$, where n is the number of nodes, a node number at a given place in the string has priority over all appearing later. This information is used to pick which node, of all those possible in the data ready list, will be placed next. This removes all heuristic judgements based on node attributes.

Most of the schedules produced were valid. This is attributed to the power of the implicit heuristic of list scheduling, placing only data ready nodes. In the present implementation, integrating the GENITOR routines took less time than tuning the list scheduling heuristics in order to achieve a high degree of reliability in the production of valid schedules. Each machine targeted requires the re-tuning of these heuristics, replicating this time spent. Using a genetic algorithm, nothing needs modification to target to another architecture. This can greatly increase the flexibility of an instruction scheduler, useful both for pre-production performance studies where architectural features may be changed and their impact studied, and where the production of code for many disparate machines is desired.

This combined list scheduling/genetic algorithm performed well. In simple DDDs, it easily found solutions as good as list scheduling alone. These were instances where timing was flexible, and placement order was basically irrelevant. The combination also found some new best-known solutions to difficult DDDs. While this may appear surprising, consider the great lengths required to generate a set of heuristics that would usually produce valid schedules. The heuristic with most impact on final schedule length, critical path, has been shown to produce erroneous results in simple cases. Other heuristics, such as counting the number of restricted edges a node has, must be emphasised in an attempt to assure schedule validity. The emphasis therefore shifts (correctly) from the possibility of generating shorter schedules to the probability of producing valid schedules. The use of GAs tends to place nodes in an order producing shorter results, and with an evaluation function reflecting failure as a longer result, it will place nodes in an order that also produces valid results.

A detraction from these encouraging results is the time complexity of the combined algorithm. As shown before, list scheduling is at least $O(n^2)$, and repeatedly performing this operation to evaluate a given string increases the time complexity. If the number of generations is linear with respect to the number of nodes in the DDD, the complexity becomes $O(n^3)$. Increasing the population size also results in similar complexity increases.

Without List Scheduling

The time complexity of the GA-based list scheduling method motivated an exploration into other forms of scheduling. No existing methods had less complexity and fit within the framework developed. The goal became the placement of operations in less than $O(n^2)$ complexity. It was noticed that, with the application of the absolute timing algorithm, each node already "knew" approximately where in the final schedule it must be placed. If the order of placement could be performed intelligently, there was no reason for a top-down (list scheduling) priority in node placement. Top-down priority had been used as a pseudo-intelligent form of ordering; it cannot adapt to vagaries of individual graphs. With genetic algorithms an intelligent, adaptable method is present. The scheduling mechanism therefore became using the GA to pick the order operations are placed in the schedule. This works because each operation "knows" where it can be placed. Certainly, failures can occur due to choosing an improper order, creating an invalid schedule.

To increase the likelihood of valid schedules, limited lookahead scheduling is employed. Here, "limited" denotes lookahead packing only those nodes with (n, n) absolute timing. That is, only nodes with no choice in their placement, due to the placement of another by the GA, are scheduled. Nodes with $\Theta(op) = (n, m)$ absolute timing are ignored by lookahead. This retains the greatest amount of flexibility and problem knowledge for the genetic algorithm to work with. If the GA places nodes toward the end of the DDD first, lookahead compaction may do most of the work in scheduling the DDD. This is not a detriment to the process as it only properly reflects the affects of the GA choice of node-placement order. As a by-product, this lookahead effort decreases the time spent by the entire scheduling process, as the GA is required to examine fewer nodes for placement. Most of the time spent scheduling then resides in the absolute timing and lookahead algorithms.

A difficulty with this method is the increased potential of generating invalid schedules. As node placement order is critical for success in creating a valid final schedule, the original random generation of orders the genetic algorithm provides tend to fail often. Several methods were used to increase the number of valid schedules generated:

1. increasing the number of generations,
2. increasing the population size, and
3. seeding the initial population with a known good order. This order may be generated with another form of scheduling.

All three of these methods produced an increase of valid schedules for difficult-to-schedule DDDs. The first method provides the genetic operator more diverse material upon which to operate. The second enlarges the area of the search space represented in the initial population. The third refines an initially correct schedule while also searching for better solutions.

The ease of implementing this solution to the instruction scheduling problem came due to all the work mentioned earlier on scheduling DDDs. Without a correct absolute timing algorithm and a working lookahead mechanism, this approach would have proved difficult. As it was, the implementation required fewer lines of code than the combined GA/list scheduler. This is important, the earlier lower-level work supported the new algorithm well; it provides a platform whereby scheduling research can be easily accomplished. Studies are then simple to make.

5.3.3 Studies

Various machines were targeted and studied. Figures 5.2 – 5.7 contain graphs comparing the performance of each of the six different operators on various problems for the IBM RS/6000 architecture. The graphs also show the results from the traditional list scheduler, and the theoretical best, for comparison. Some of the problems were selected from the Lawrence Livermore Loops [McM86], translated to C by Martin Fouts. Others were from the LINPACK [DBMS78] suite translated to C by Stephan V. Schell. Both have been used to compare the performance of different computers on computational tasks. Each of the graphs represent the longest basic block in each particular problem.

These results mirror the ranking of operators in the brewery production line, indicating that operators that perform well in other scheduling tasks perform well with instruction scheduling. This order of performance is reversed on problems, such as the TSP, that rely on adjacency information. This is important in that it provides more evidence in classifying the instruction scheduling problem and directing efforts in future approaches to the problem.

Not all of the blocks are presented; some produced uninteresting results. Interesting results were considered those that had multiple correct answers, i.e., places where the search capability of genetic algorithms played a part. There exist DDDs with little flexibility in the placement of operations. This can arise either from the

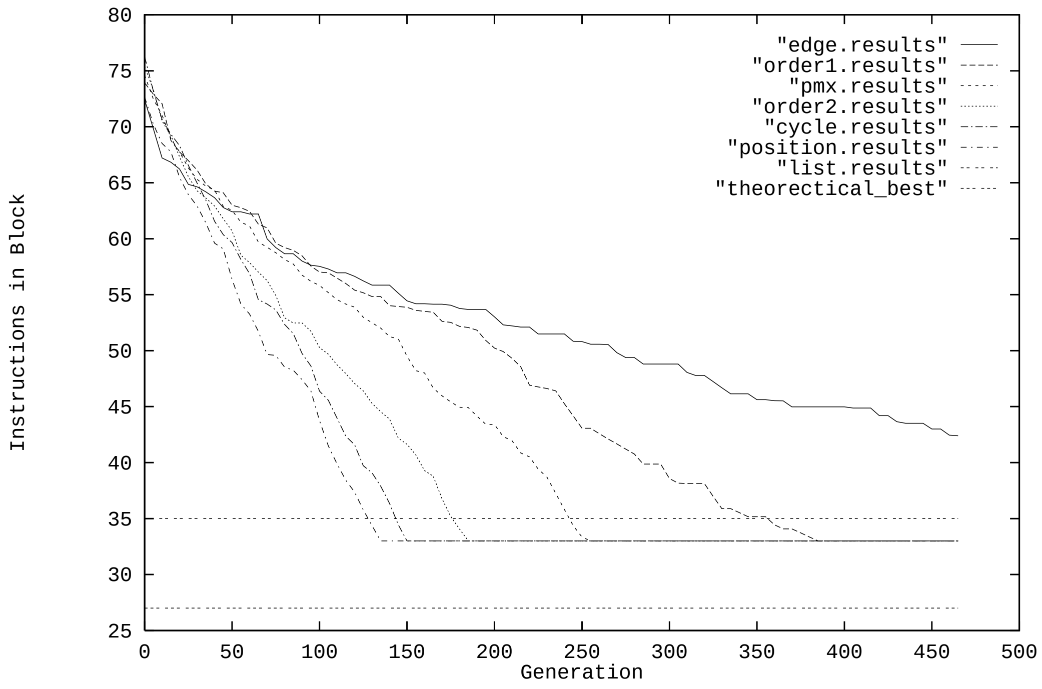

Figure 5.2: Livermore Kernel Number 1

nature of the targeted machine or the type of operations in the block. Machines with little concurrency offer little opportunity, or need, to schedule instructions. Basic blocks with either few, or highly constrained operations also afford for little optimization. Entry, exit, loop headers and trailers are examples of this type of block. Most operations are load/store or comparison/branch, removing some ability to optimize due to their sequential nature.

While all of the operators solved each problem, the rate of convergence to better solutions varied. Those operators emphasizing order over adjacency in a string perform better. The position operator most often produced the fastest convergence. The edge recombination, best at the TSP, was always to slowest to converge. Note in Figure 5.4 two competitive solutions exist, giving rise to two lines of convergence. Figure 5.6 is the largest block presented, with 288 operations. An excellent result of 135 instructions was found, the theoretical best being 133 instructions. This suggests that using a genetic scheduling method benefit larger blocks more than

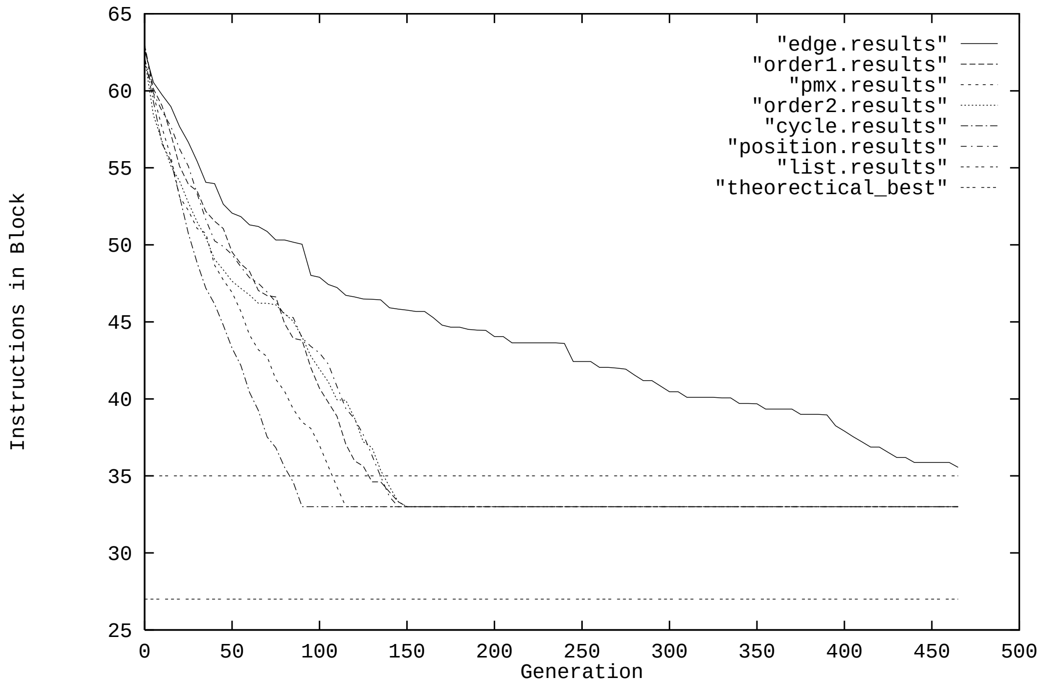

Figure 5.3: Livermore Kernel Number 2

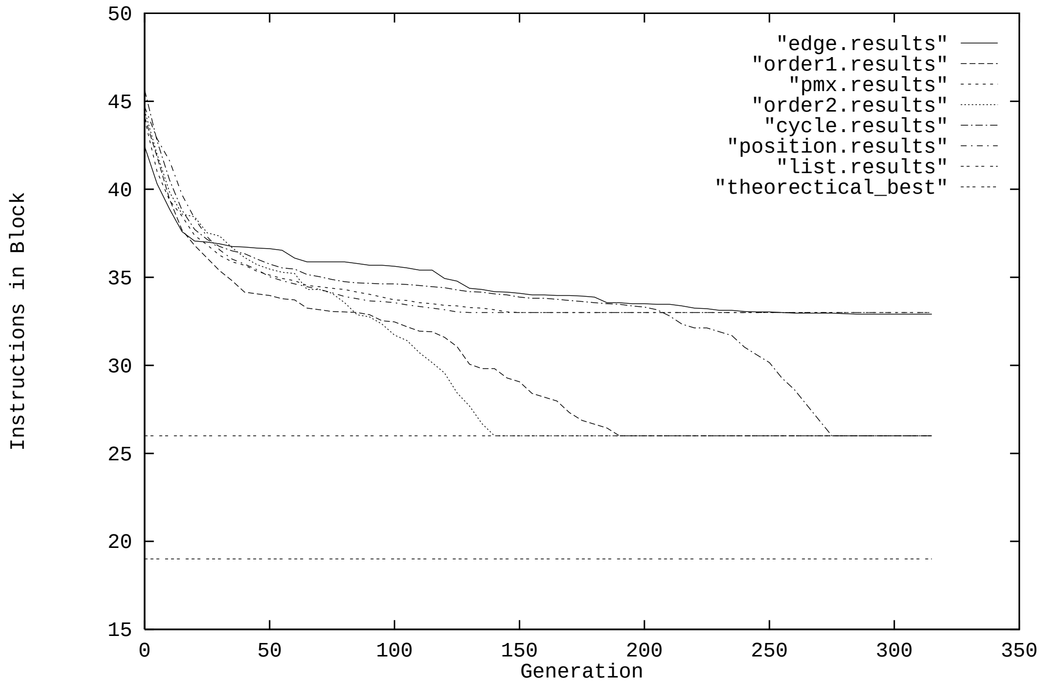

Figure 5.4: Livermore Kernel Number 4

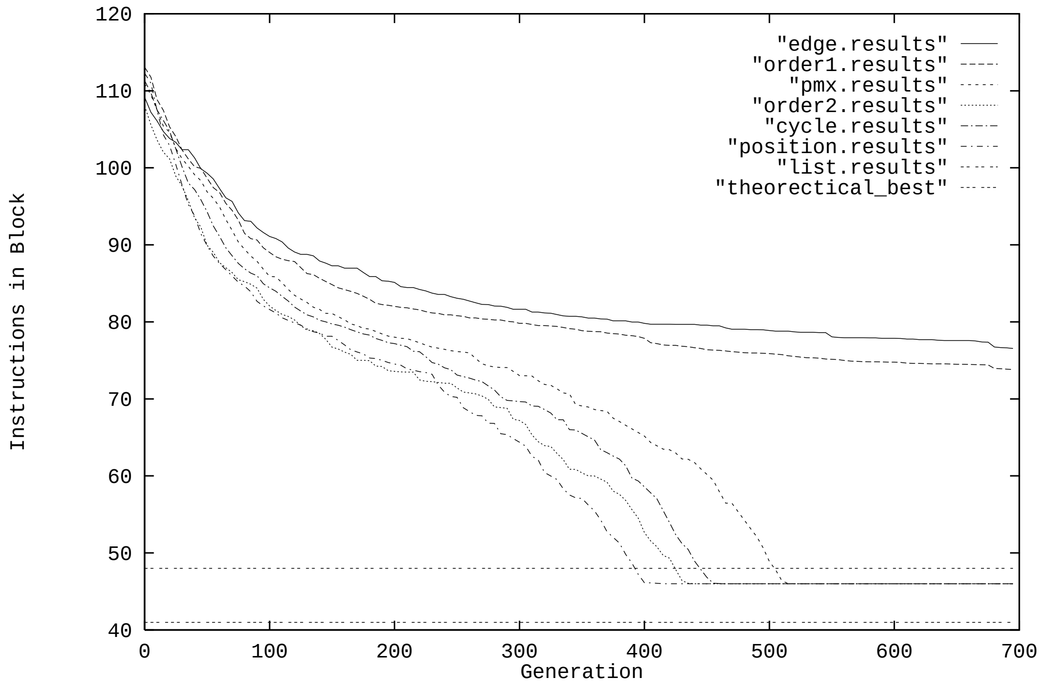

Figure 5.5: Livermore Kernel Number 8

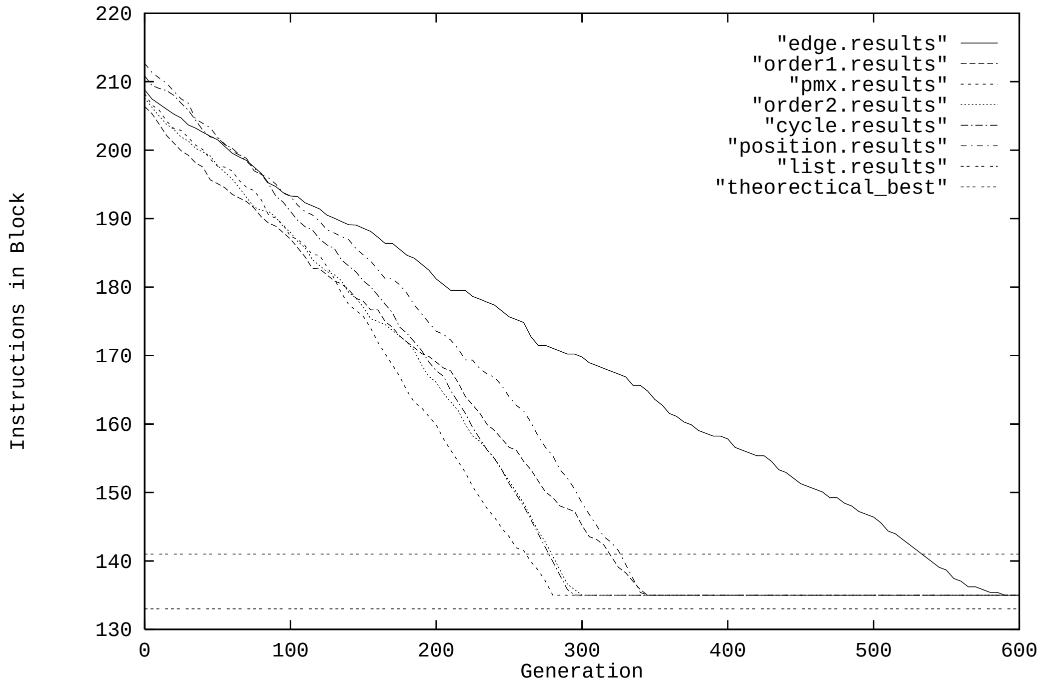

Figure 5.6: Livermore Kernel Number 10

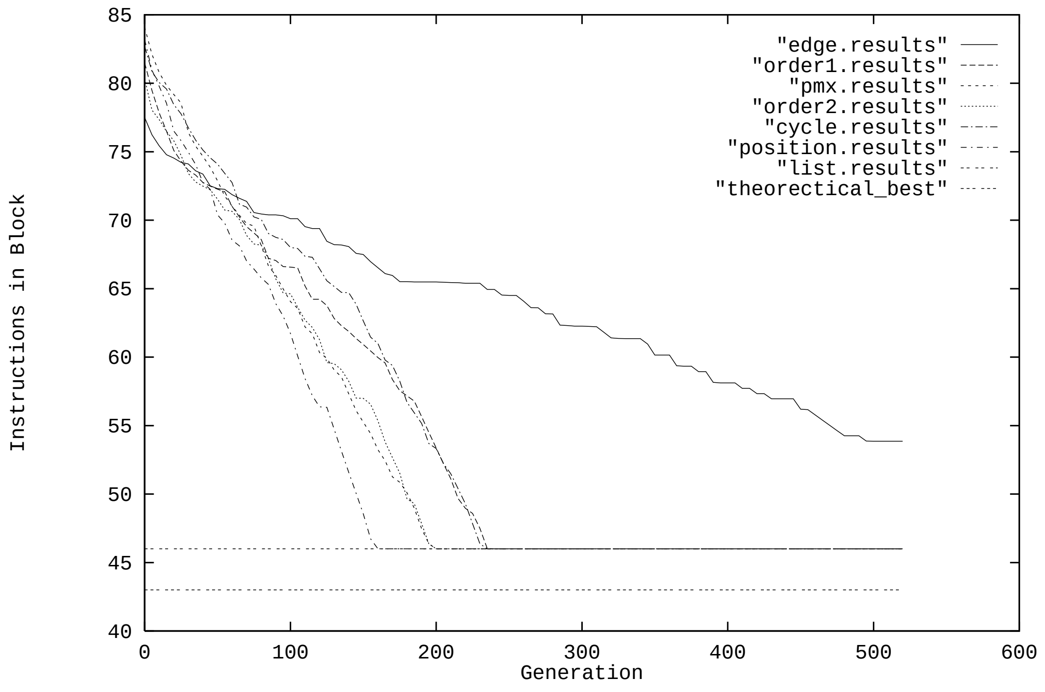

Figure 5.7: Linpack cgefa.c

smaller; both the additional flexibility and the desire for tighter code in long blocks motivate this.

The GA-based method always performed as well as the list scheduling, and often better. This came as no surprise, they have more opportunity and ability to search for good solutions. They also take more time in doing so. A GA-based approach is not a be-all or end-all method. Under at least two conditions they are attractive:

1. the desire for a machine-independent method, to use for rapidly changing targets, and

2. the desire for highly optimized code.

When code is needed as rapidly as possible, an architecture is mature, and an investment to implement an exceptional heuristic-based scheduler can be made, a GA-based approach may be displaced.

It is possible to vary various parameters of a genetic algorithm to increase its effectiveness on a particular problem although no such effort was made in these studies. The selection bias was 1.5. There was no mutation, adaptive or otherwise. Each population was initialized with different (random) values.

5.4 Summary

As the graphs bear out, order is more important than adjacency in optimizing the instruction scheduling problem. Operators emphasizing order converged faster than those emphasizing adjacency. This comes as no surprise; all previously effective methods also emphasized order. This evidence does however shed additional light on the nature of of the instruction scheduling process by providing more controlled, empirical evidence.

The ordering of the placement of nodes by the genetic algorithm mirrors the approach used by human coders. The nodes with the greatest impact on final schedule length are placed first, with those having lesser impact placed later. The order of placement that ensures validity is also reflected.

Chapter 6

CONCLUSIONS, CONTRIBUTIONS, DIRECTIONS

In this, the final chapter, goals set forth for this work in the first chapter will be addressed in a concise form. First, a section will review the motivation for, and directions pursued by, this work. Another section will make explicit the contributions to the current state of the art this work provides. A final section will suggest directions for follow-on research that should bear fruit.

6.1 Conclusions

The single most motivating factor that produced this work was the desire to produce an efficacious instruction scheduler for code improvement in a retargetable compiler environment. This goal produced several immediate concerns.

First, whatever representation is chosen to preserve the semantics of the source during instruction scheduling must specify a minimal number of constraints, allowing for a maximal amount of achievable parallelism. With the choice of dags to represent the data dependencies, the removal of spurious edges is paramount. This removal adds to the available parallelism, and was treated in great detail in Section 2.2.2. Recent work has shown that computing the number of total orders consistent with a partial order is #P–complete. There does not currently exist a method to solve this class of problems optimally in polynomial time. Any polynomial-time method to find a solution must therefore involve an incomplete search of the solution space. This drove the exploration of increasing the effectiveness of an existing method, list scheduling. When this process of exploration proved ultimately dissatisfying, different methods of search were explored. Promising work had been done locally using genetic algorithms for scheduling problems. This avenue was then explored with satisfying results.

The desire for as few edges as possible in the DDD predicated several methods of edge reduction. By assigning changing values to register resources after instruction scheduling, no register data anti-dependencies are created to restrict parallelism. If spill code is required, less likely in current register-rich architectures, another iteration of the code selection/instruction scheduling process is required. Empirical evidence has shown that this approach can generate significant speedups. Post-scheduling register assignment has complete information on the final code, allowing better assignment and better calculation of spill cost, should spilling be needed.

The removal of dead code, whether programmer- or compiler-generated, removes all non-essential output dependencies. The only time either an input or output dependent edge is required is in the presence of volatile machine resources that must be so modeled.

A desire to model a wide range of machines, and therefore a wide range of architectural features, necessitated an examination of the data dependence representation. With the increasing exposure of low-level machine features to compiler technology, a rich set of ordering information must be considered. Machine features finding favor in recent processors include

- multistage pipes,
- pipes without interlocked stages,
- synchronous pipes, and
- transient resources (i.e. used for communication between different sections of the processor.)

Simple timing on edges will not suffice to model these types of resources. The representation, and the algorithms operating upon the representation, must be extended to include the information necessary to reflect these resources. The edges and nodes in the DDD are therefore extended to include information about minimum and maximum origination and completion times for the various operations available in a machine. Routines must either be updated or created to deal with this increased richness in expression. Examples include the absolute timing, foresight, and lookahead algorithms.

With the increased flexibility in representation these methods achieve comes an increased burden on the scheduling method. At one end of the difficulty spectrum are machines with no parallelism and no transient resources. For these machines, instruction scheduling is an implicit operation done during code selection. There is no reason to try any one ordering over another; they all result in a schedule that is the same length. The other end of the spectrum contains machines with a large amount of instruction-level parallelism and a large amount of transient resources. These mechanisms should be visible to the translator so that intelligent use of operations result in a speed increase for a given complexity level in a given processor technology. To realize this performance enhancement, and therefore recoop the investment made in the production of the processor, the presence of an intelligent instruction scheduler in the code-generation process is needed.

The possibility of exponential execution times that optimal instruction scheduling takes requires methods to reduce the running time of any algorithm. Heuristic-driven search is a favored method for reducing the running time of many optimization methods. This ostensibly attempts to search only those areas of the solution space that appear to reduce the length of the final schedule. If effective heuristics can be found that usually generate optimal or near-optimal results, it would be desirable to use them. There can be a number of heuristics that will produce good solutions for any given instance of a problem; the difficulty lies in finding heuristics that produce acceptable (or even valid) solutions for all instances of a problem. An attempt was made in this work to enumerate as many heuristics that could have an impact upon the effectiveness of list scheduling. This resulted in finding that heuristics failed to always produce valid schedules, and additionally that the use of multiple heuristics can lead to competing goals.

Non-heuristic methods also increase the chances for generating valid, high quality schedules. Foresight is an example from instruction scheduling that increases the chances for schedule validity by checking the impact a certain placement decision has upon the resulting final schedule. A drawback of this approach is that later placement decisions might not reflect the schedule examined earlier by foresight. This motivated the development of lookahead compaction, a greedy method. Lookahead not only examines the schedule resulting from the placement of an operation, it places all operations that lose flexibility due to that placement. This provides a substantive increase to the probability of generating a valid schedule, but does not guarantee validity. Lookahead uses only local information for its execution; global data that might influence the final schedule are not considered.

None of the methods attempted thus far produced completely satisfying results. Genetic algorithms had shown that, for similar problems, good schedules could be expected. A simple approach was tried wherein a GA prioritized the data ready list for a list scheduler. This proved advantageous by finding better schedules than had been found by the heuristic-driven list scheduler. This should come as no surprise; heuristics producing excellent results for certain code fragments certainly will not produce such results for all code fragments. The GA-based list scheduling algorithm also required more time due to the need to execute the entire list scheduling process to evaluate a single priority list. This motivated the search for a GA-based approach not predicated on list scheduling. An approach was found wherein the genetic algorithm chose the order of placement of operations with the absolute timing algorithm assuming increased responsibility for assuring that any order used would tend to produce a valid schedule. Lookahead compaction also increased the chances of producing a valid schedule by placing restricted operations in their respective locations.

Given these different methods suitable for scheduling dags with timing, it is possible to choose the method used for a specific dag on the fly. For example, if a dag contains "easy" timing (few restricted edges), using the power of an advanced technique is not only wasted, it is wasteful. However, the existence of powerful techniques allows the method of solution to match the difficulty of the problem. The difficulty of scheduling may either be observed by the examination of certain descriptive parameters or by the failure of the less-powerful methods of scheduling.

6.2 Contributions

This section contains a brief outline of the contributions made by this work.

6.2.1 DDD constraints

A study of the meaning of the edges in a DDD was produced. Motivation for the removal of any unnecessary edges, in order to increase the flexibility of scheduling, was given. Several processes to achieve this result were presented. Post-scheduling register assignment was shown to be an important tool for the reduction of anti-dependent edges. The minimal complement of edges required in a DDD to properly reflect program semantics was addressed. Recognition of the motivation and value of input dependent edges was developed.

6.2.2 List scheduling

List scheduling was given thorough study by this work. A number of different heuristics were developed and used in attempts to derive an efficacious scheduler. It was discovered that increasing the number of heuristics does not increase either the ability to create shorter or more likely valid schedules. Heuristics that had previously been deemed important were found wanting. Heuristics were found to compete within a DDD, each trying to emphasize an important characteristic in either the length or validity of the final schedule. The interval of updating the priorities within the data ready list was examined. Justification for either static or dynamic ordering of the priority list was found. Most importantly, the difficulty of using only heuristics for a general-purpose, retargetable instruction scheduler was demonstrated, motivating research into other areas of optimization.

The absolute timing algorithm was studied and improved by removing the possibility of timing loops. The least stringent conditions for the schedule range for an operation were explored and enumerated. The need to iterate when computing the timing on a graph was demonstrated. An observation was made on the number of times a node was visited, leading to a non-recursive speedup method.

Foresight was studied in an attempt to produce valid schedules more often. Conditions for failure of the algorithm were found, motivating the development of lookahead scheduling. Both foresight and lookahead scheduling were extended to include the possibility of examining nodes having predecessors. This allows for the removal of the data ready condition for placing nodes in the schedule, providing both a speedup in schedule generation and an increased likelyhood of generating a valid schedule.

6.2.3 Genetic Algorithms

Genetic algorithms were applied to the instruction scheduling process. Two methods were developed, one that performed the sequencing of the data ready list and one that controlled the order of placement of operations. The second method does not require the time spent to rank the apparent importance of each of the operations. Both were found to produce promising results. Less time was required to produce a working GA-based scheduler than to tune a heuristic-based list scheduler. The GA-based methods were better able to cope with a variety of difficult DDDs. The addition of these methods allows for the choice of scheduling methods based upon a DDD's characteristics.

6.3 Directions

With the framework provided by this work, many interesting directions for future work exist. Three follow.

1. Software pipelining was not considered. Its ability to improve repetitive code structure is impressive. Loops are important structures to optimize as more execution time will be spent in them compared to straight line code. Previous methods have used an unrolling technique combined with local compaction [SDX86, MSDP86]. Because of the power of the scheduling methods developed here, their use on critical portions of code could create enhanced run-time performance for a wide variety of code. Given a choice, "critical sections" of code should be emphasized in the optimization process at the expense of less critical sections of code. The range of scheduling mechanisms presented herein allows this type of flexibility during code generation. For example, another parameter used to decide the type of scheduling performed on a section of code (other than the previously-mentioned parameters to account for dag properties), could be the importance of producing short schedules for a particular part of the code. Exploring the impact of GA-based scheduling with respect to these types of code sections would be interesting.

2. Inter-block scheduling was not considered. The problem of intra-block scheduling needs to be addressed in more depth. Propitiously, methods exist that use intra-block scheduling as a basis of inter-block scheduling. One that has received much attention is *Trace Scheduling* due to Fisher [Fis81]. Work on an earlier academic compiler at Colorado State University produced good results using trace scheduling in combination with an existing local compactor [HMS87, How87]. A difficulty encountered in that work was the computational complexity of the scheduler, resulting in limiting the number of blocks

considered during instruction scheduling. Using methods suggested here, this complexity may be reduced or bounded by a chosen amount of time.

3. The search capabilities of genetic algorithms could be used to generate better weightings on the list scheduling algorithm. As these will vary on a per-architecture basis, a suite of "typical" programs could be compiled while changing the weights in the discriminating polynomial selection function. Those weights found to produce good results for the current machine/program combination could then be used for any program on that machine. This would also provide feedback to the machine designers as to which features have a direct bearing upon performance and which do not. Closing the loop that exists between hardware and software design will increase the speed of both.

6.4 Summary

In summary, previous methods of local instruction scheduling were examined and found lacking. Several new approaches were discovered and developed to address specific deficiencies uncovered. A wholly new method, applying genetic algorithms, was explored and found beneficial. The increasing complexity of machines mandate the strengthing of instruction scheduling; genetic algorithms are a method of achieving this.

REFERENCES

[Adv85] Advanced Micro Devices, Santa Clara, CA. *Am2900 Family Data Book*, 1985.

[All86] V.H. Allan. *A Critical Analysis of the Global Optimization Problem for Horizontal Microcode*. PhD thesis, Computer Science Department, Colorado State University, Fort Collins, Colorado, 1986.

[AM87] V.H. Allan and R.A. Mueller. Phase coupling for horizontal microcode generation. In *Proceedings of the 20th Microprogramming Workshop (MICRO-20)*, Colorado Springs, CO, December 1987.

[AM88] V.H. Allan and R.A. Mueller. Microcode compaction with general synchronous timing. *IEEE Transactions on Software Engineering (Special Section on Microprogramming)*, 14(5):595–599, May 1988.

[AN88] A. Aiken and A Nicolau. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering*, 14(5), May 1988.

[AP71] F. Astopas and K.I. Plukas. Method for Minimizing Computer Microprograms. *Automatic Control*, 5(4):10–16, 1971.

[AST67] D. Anderson, F. Sparacio, and R. Tomasulo. The ibm system/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1), January 1967.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[Ban88] U Banerjee. *Dependence Analysis for Supercomputing*. Kluwer, Boston, Ma, 1988.

[BCKT89] P. Briggs, K.D Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, Oregon, June 1989.

[BDM$^+$88] S.J. Beaty, M.R. Duda, R.A. Mueller, P.H. Sweany, and J Varghese. Optimization issues for a retargetable microcode compiler. *IEEE MicroArch*, 3(1):5–15, December 1988.

[Bea87] S.J. Beaty. Register allocation and assignment in a retargetable microcode compiler using graph coloring. Master's thesis, Computer Science Department, Colorado State University, Fort Collins, CO, 1987.

[BEH91] D.G. Bradlee, S.J. Eggers, and R.R Henry. Integrating register allocation and instruction scheduling for riscs. In *Proceedings of the Forth Internation Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, April 1991.

[Boo87] L. Booker. Improving search in genetic algorithms. In Lawrence Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 61–73. Morgan Kaufmann, 1987.

[BSKT79] U. Banerjee, S. Shen, D.J. Kuck, and R.A. Towle. Time and parallel processor bounds for fortran-like loops. *IEEE Transactions on Computers*, C-28(9):660–670, Sep 1979.

[BW90] Graham Brightwell and Peter Winkler. Counting linear extensions is #p-complete. DIMACS Technical Report 90-49, Bellcore, 445 South Street, Morristown, New Jersey, 07960, July 1990.

[CAC+81] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, and P.W. Hopkins, M.E.and Markstein. Register allocation via coloring. *Computer Languages*, 6, 1981.

[Can90] D.C. et al. Cann. Sisal reference manual - language version 2.0. Technical report, Larrence Livermore National Laboratory and Colorado State University, 1990.

[Cha82] G.J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, pages 201–207, June 1982.

[CNO+88] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Papworth, and P.K. Rodman. A vliw architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, 37(8), August 1988.

[Cof76] E.G Coffman. *Computer and Job-Shop Scheduling Theory.* Jon Wiley & Sons, New York, 1976.

[CS89] Gary A. Cleveland and Stephen F. Smith. Using genetic algorithms to schedule flow shop releases. In *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989.

[Das84] S. Dasgupta. A Model of Clocked Micro-Architectures for Firmware Engineering and Design Automation Applications. In *Proceedings of the 17th Microprogramming Workshop (MICRO-17)*, pages 298–308, New Orleans, LA, November 1984.

[Dav85] L. Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1985.

[DBMS78] J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart. Linpack user's guide. Technical report, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1978.

[DDMS86] W. Damm, G. Doehmen, K. Merkel, and M. Sichelschmidt. The AADL/S^* Approach to Firmware Design Specification. *IEEE Software*, 3(4):27–37, July 1986.

[DeJ86] K. DeJong. *An Analysis of Reproduction and Crossover in a Binary - coded Genetic Algorithm.* PhD thesis, University of Michigan, Ann Arbor, 1986.

[DeW76] D.J. DeWitt. *A Machine-Independent Approach to the Production of Optimal Horizontal Microcode.* PhD thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, MI, 1976.

[Dig81] Digital Equipment Corporation. *VAX Architecture Handbook*, 1981.

[DLSM81] S. Davidson, D Landskov, B.D. Shriver, and P.W. Mallett. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers*, C-30(7), July 1981.

[DT76] S. Dasgupta and J. Tartar. The Identification of Maximal Parallelism in Straight-line Microprograms. *IEEE Transactions on Computers*, C-25(10):986–992, Oct 1976.

[Ell86] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* The MIT Press, Cambridge, MA, 1986. PhD thesis, Yale, 1984.

[FERN84] 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 SIGPLAN '84 Conference on Compiler Construction*, pages 37–47. ACM, June 1984.

[Fis79] J.A. Fisher. *The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling.* PhD thesis, Courant Institute of Mathematical Sciences, New York University, New York, NY, October 1979.

[Fis81] J.A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[FLS81] J.A. Fisher, D. Landskov, and B.D. Shriver. Microcode compaction: Looking backward and looking forward. In *Proceedings of the National Computer Conference*, volume 50, pages 95–102, Montvale, NJ, July 1981. AFIPS Press.

[Gas89] F. Gasperoni. Compilation techniques for vliw architectures. Technical report, Courant Institute of Mathematical Sciences, New York University, March 1989.

[Gib85] A. Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, Cambridge, England, 1985.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP–Completeness.* W.H. Freeman and Company, San Francisco, CA, 1979.

[GL85] D. Goldberg and R Lingle. Alleles, loci, and the traveling salesman problem. In *Proceedings of the International Conference on Genetic Algorithms and their Applications*, 1985.

[Gol89] David Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[Hec77] M.S. Hecht. *Flow Analysis of Computer Programs.* North-Holland, New York, NY, 1977.

[HJP$^+$82] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill. MIPS: A Microprocessor Architecture. In *Proceedings of the 15th Microprogramming Workshop (MICRO-15)*, pages 17–22, Palo Alto, CA, October 1982.

[HL85] T.L. Harmon and B. Lawson. *The Motorola MC 68000 Microprocessor Family.* Prentice Hall, Englewood Cliffs, NJ, 1985.

[HMS87] M.A. Howland, R.A. Mueller, and P.H. Sweany. Trace scheduling optimization in a retargetable microcode compiler. In *Proceedings of the 20th Microprogramming Workshop (MICRO-20)*, Colorado Springs, CO, December 1987.

[Hol75] John Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

[How87] M.A. Howland. Integration of a Trace Scheduling Optimizer in a Retargetable Microcode Compiler. Master's thesis, Computer Science Department, Colorado State University, Fort Collins, CO, 1987.

[HS80] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, Potomac, MA, 1980.

[IBM90] IBM. *IBM Journal of Research and Development*, January 1990.

[Int90] Intel. *i860 64-bit Microprocessor Programmer's Reference Manual*, 1990.

[KA91] R.F. Krick and D. Apostolos. The evolution of instruction sequencing. *Computer*, 24(4), April 1991.

[Kan87] Gerry Kane. *mips R2000 RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1987.

[Kat85] M.G.H. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI*. The MIT Press, Cambridge, MA, 1985.

[KD81] A. Klassen and S. Dasgupta. S^*(QM-1): An Instantiation of the High-Level Microprogramming Language Schema S^* for the Nanodata QM-1. In *Proceedings of the 14th Microprogramming Workshop (MICRO-14)*, pages 124–130, Chatham, MA, Oct 1981.

[Knu73] D.E. Knuth. *The Art of Computer Programming*, volume I: Fundamental Algorithms. Addison-Wesley, Reading, MA, second edition, 1973.

[LA89] J.L. Linn and C.D. Ardoin. An example of using pseudofields to eliminate version shuffling in horizontal code compaction. In *Proceedings of the 22nd Microprogramming Workshop (MICRO-22)*, Dublin, Ireland, 1989.

[LDSM80] D. Landskov, S. Davidson, B.D. Shriver, and P.W. Mallett. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3):261–294, September 1980.

[McM86] F.H. McMahon. The livermore fortran kernels: A computer test of numerical performance range. Technical report, Lawrence Livermore National Laboratory, December 1986.

[MDSW88] R.A. Mueller, M.R. Duda, P.H. Sweany, and J.S. Walicki. Horizon: A Retargetable Compiler for Horizontal Micro-Architectures. *IEEE Transactions on Software Engineering (Special Section on Microprogramming)*, 14(5):575–583, May 1988.

[MS86] R.A. Mueller and P.H. Sweany. Horizon Code Generator Series-Parallel DDG Coupler/Decoupler (Version 3.1). Technical Report MAD-86-10, Firmware Engineering and Micro-Architecture Design Laboratory, Colorado State University, Fort Collins, CO, September 1986.

[MSDP86] R.A. Mueller, B. Su, M.R. Duda, and B. Plomondon. A Case Study in Signal Processing Microprogramming with the URPR Software Pipelining Technique. In *Proceedings of the 19th Microprogramming Workshop (MICRO-19)*, New York, NY, October 1986.

[Nan74] Nanodata Corporation, Williamsville, NY. *QM-1 Hardware Level User's Manual*, 2 edition, 1974.

[Nic84] Alexandru Nicolau. *Parallelism, Memory Anti-aliasing, and Correctness Issues for a Trace Scheduling Compiler.* PhD thesis, Department of Computer Science, Yale University, New Haven, Conn, December 1984.

[Nic85] Alexandru Nicolau. Percolation scheduling: A parallel compilation technique. Technical report, Department of Computer Science, Cornell University, Ithaca, New York, May 1985.

[OSH87] I.M. Oliver, D.J. Smith, and J.R.C. Holland. A study of permutation crossover operators on the traveling salesman problem. In John Grefenstette, editor, *Genetic Algorithms and Their Applications: Proceeding of the 2nd International Conference*, pages 224–230. L. Ebaum Assoc., 1987.

[Pat89] Y.N. Patt. Microarchitecture choices (implementation of the vax). In *Proceedings of the 22nd Microprogramming Workshop (MICRO-22)*, Dublin, Ireland, 1989.

[Pat90] Kimberly Patch. Intergraph issues superscalar clipper. *digital review*, September 10 1990.

[PKL80] D.A. Padua, D.J. Kuck, and D.H. Lawrie. High speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29(9):763–776, Sept 1980.

[PS81] D.A. Patterson and C.H. Sequin. RISC-1: A Reduced Instruction Set VLSI Computer. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 443–458, Minneapolis, MN, May 1981.

[PW86] D.A. Padua and M.J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, Dec 1986.

[RT74] C.V. Ramamoorthy and M. Tsuchiya. A High-Level Language for Horizontal Microprogramming. *IEEE Transactions on Computers*, C-23(8):791–801, August 1974.

[RT81] J.H. Reif and R.E. Tarjan. Symbolic program analysis in almost-linear time. *SIAM Journal of Computing*, 11(1):81–93, February 1981.

[SB90] P. Sweany and S. Beaty. Post-compaction register assignment in a retargetable compiler. In *Proceedings of the 23th Microprogramming Workshop (MICRO-23)*, Orlando, FL, November 1990.

[SDX86] B. Su, S. Ding, and J. Xia. URPR – An extension of URCR for Software Pipelining. In *Proceedings of the 19th Microprogramming Workshop (MICRO-19)*, pages 94–103, New York, NY, December 1986.

[SDX87] B. Su, S. Ding, and J. Xia. Microcode Compaction with Timing Constraints. In *Proceedings of the 20th Microprogramming Workshop (MICRO-20)*, Colorado Springs, CO, December 1987.

[SMM+91] T. Starkweather, S. McDaniel, K. Mathias, C. Whitley, and D. Whitley. A comparison of genetic sequencing operators. In *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann, 1991.

[Sys90] Gilbert Syswerda. Schedule optimization using genetic algorithms. In L. Davis, editor, *The Genetic Algorithms Handbook*. 1990.

[Tre82] N. Tredennick. Cultures of Microprogramming. In *Proceedings of the 15th Microprogramming Workshop (MICRO-15)*, pages 79–83, Palo Alto, CA, Oct 1982.

[Veg82] S.R. Vegdahl. *Local Code Generation and Compaction in Optimizing Microcode Compilers*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1982.

[WA89] P. Wijaya and V.H. Allan. Incremental foresighted local compaction. In *Proceedings of the 22nd Microprogramming Workshop (MICRO-22)*, Dublin, Ireland, August 1989.

[Wal91] D.W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Forth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, April 1991.

[Woo78] G. Wood. On the Packing of Micro-operations into Micro-instruction Words. In *Proceedings of the 9th Microprogramming Workshop (MICRO-9)*, December 1978.

[WSF89] D. Whitley, T. Starkweather, and D. Fuquay. Scheduling problems and traveling salemen: The genetic edge recombination operator. In *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989.

[WSS90] D. Whitley, T. Starkweather, and D. Shaner. The traveling saleman and sequence scheduling quality solution using genetic edge recombination. In L. Davis, editor, *The Genetic Algorithms Handbook*. 1990.

[YST74] S.S. Yau, A.C. Schowe, and M. Tsuchiya. On Storage Optimization of Horizontal Microprograms. In *Proceedings of the 7th Microprogramming Workshop (MICRO-7)*, pages 98–106, Palo Alto, CA, Oct 1974.