RESOURCE-AWARE INSTRUCTION SCHEDULING

A Thesis Presented

by

Ivan Dimitrov Baev

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in Electrical Engineering

Northeastern University

Boston, Massachusetts

May 2001
ABSTRACT

The scheduling phase in a compiler back-end is important for fully exploiting the parallel hardware of the target processor. Due to advances in VLSI technology, recent general-purpose and DSP processors (such as the Intel Itanium and Philips Trimedia) provide a large set of functional units and register files of different types. In addition, the scheduler has to deal with many diversions from the ideal uniform processor. Complex resource usage, register file port conflicts, and operation issue restrictions represent potential structural hazards that may become severe performance bottlenecks if not taken into consideration.

This thesis proposes scheduling algorithms and tight bounds that are aware of the processor complexity. We first develop the backtracking approach as a general instruction scheduling strategy for very detailed processor models. Our basic backtracking scheduler only unschedules successors of the current operation and provably terminates without compromising the performance. The more aggressive full-backtracking scheduler enables backtracking for all operations and increases the percentage of superblocks scheduled optimally over a conventional scheduler from an average of 66.9% to 81.4%, an increase of 21.7%.

Second, we establish the relative tightness of known lower bounds on the schedule length and systematically extend the tightest bound to account for complex resource usage of certain operations, a finite number of register file write ports, and operation issue restrictions. Finally, we describe linear programming-based heuristics to calculate operation priorities and show that they outperform all other current techniques. Our new resource-aware bounds and algorithms can be used to enhance present and future high performance compilers for VLIW/EPIC and superscalar processors.
Contents

1 Introduction ................................. 1
  1.1 ILP compilation and scheduling ......... 2
  1.2 Organization and overview of contributions .... 6
    1.2.1 List scheduling algorithms .......... 6
    1.2.2 Backtracking schedulers ............ 7
    1.2.3 Resource-aware bounds .............. 8

2 Scheduling framework .................... 10
  2.1 Processor model ......................... 11
  2.2 Scheduling region ....................... 13
  2.3 Performance objective .................. 14
  2.4 Early and late times .................... 16
  2.5 NP-completeness of superblock scheduling .......... 17

3 List scheduling algorithms ............... 19
  3.1 WCT algorithms ......................... 21
    3.1.1 Single-machine optimal algorithms ....... 23
    3.1.2 Linear programming-based algorithms .... 27
    3.1.3 Experimental evaluation .............. 31
  3.2 Superblock algorithms .................. 33
    3.2.1 Combinatorial algorithms ............ 34
    3.2.2 Experimental evaluation ............ 35
  3.3 Limitations of list scheduling .......... 38
4 Resource-aware backtracking schedulers  
  4.1 Motivation ............................................. 43  
  4.2 Main concepts ........................................ 46  
  4.3 Basic backtracking algorithm ....................... 51  
  4.4 Full backtracking scheduler ....................... 54  
  4.5 Selective backtracking scheduler ................. 55  
  4.6 Experimental evaluation ............................ 59  
    4.6.1 Branch delay slots and optimally scheduled branches and regions .. 60  
    4.6.2 Dynamic cycles .................................. 63  
    4.6.3 Runtime and scheduling steps .................. 64  
  4.7 Related work ......................................... 66  

5 Resource-aware bounds  
  5.1 Scheduling bounds .................................. 73  
  5.2 Lower bounds for basic blocks ................... 75  
  5.3 Lower bounds for superblocks .................... 82  
  5.4 Extending bounds to account for resource constraints .......... 85  
    5.4.1 Register file write ports ...................... 86  
    5.4.2 Complex resource usage ....................... 90  
    5.4.3 Issue constraints ............................. 91  

6 Conclusions and future work  

Bibliography
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Delays may be necessary in optimal schedules for out-trees</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>Backtracking schedulers can fill branch delay slots</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Backtracking schedulers can handle resource conflicts</td>
<td>46</td>
</tr>
<tr>
<td>4.3</td>
<td>A backtracking scheduler may not terminate</td>
<td>49</td>
</tr>
<tr>
<td>4.4</td>
<td>Main loop of the basic backtracking scheduling algorithm</td>
<td>52</td>
</tr>
<tr>
<td>4.5</td>
<td>Main loop of the FullBT backtracking scheduler</td>
<td>55</td>
</tr>
<tr>
<td>4.6</td>
<td>Operation of the FullBT scheduler</td>
<td>56</td>
</tr>
<tr>
<td>4.7</td>
<td>Main loop of the SelectBT backtracking scheduler</td>
<td>58</td>
</tr>
<tr>
<td>4.8</td>
<td>Operation of the SelectBT scheduler</td>
<td>58</td>
</tr>
<tr>
<td>4.9</td>
<td>Percent of filled branch delay slots</td>
<td>61</td>
</tr>
<tr>
<td>4.10</td>
<td>Percent of branches optimally scheduled</td>
<td>62</td>
</tr>
<tr>
<td>4.11</td>
<td>Percent of superblocks optimally scheduled</td>
<td>62</td>
</tr>
<tr>
<td>4.12</td>
<td>Percent of Cycle gap eliminated by backtracking schedulers for 111L3</td>
<td>65</td>
</tr>
<tr>
<td>4.13</td>
<td>Percent of Cycle gap eliminated by backtracking schedulers for 211L3</td>
<td>65</td>
</tr>
<tr>
<td>5.1</td>
<td>Importance of tight bounds</td>
<td>75</td>
</tr>
<tr>
<td>5.2</td>
<td>Deriving a bound that accounts for register file write ports</td>
<td>88</td>
</tr>
<tr>
<td>5.3</td>
<td>Deriving a bound that accounts for the branch resource usage</td>
<td>91</td>
</tr>
<tr>
<td>5.4</td>
<td>Operation issue in the Itanium processor</td>
<td>92</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Quality of single-machine-optimal and LP-based algorithms vs. LP lower bound for WCT instances (1.0 = 1%) ........................................... 32
3.2 Quality of combinatorial and LP-based algorithms vs. LP lower bound for superbloc instances (1.0 = 1%) ........................................... 37

4.1 Benchmark set ........................................................................... 59
4.2 Percent speedup over the Cycle scheduler .................................... 63
4.3 Ratio of scheduling steps of backtracking schedulers over Cycle ....... 66
4.4 Ratio of runtime of backtracking schedulers over Cycle ................. 67

5.1 Quality of lower bounds vs. an upper bound for basic blocks: average absolute difference and % instances equal .................................. 82
5.2 Quality of lower bounds vs. the tightest lower bound for superbloc .... 83
5.3 DP vs. LP lower bounds for WCT on GP2 processor ..................... 85
5.4 Quality of LC and WP lower bounds vs. critical path bound for superbloc 89
Chapter 1

Introduction

Instruction-level parallelism (ILP) is a set of techniques for compiler and processor design that allow machine operations from a sequential program to execute in parallel [80]. Historically, there have been two ways to implement ILP. In a hardware-based implementation, a superscalar processor [60] executes the sequential program by dynamically finding parallelism among the operations and scheduling them on the processor’s multiple functional units. In a software-based implementation [31], it is the compiler that statically parallelizes and schedules the operations; the resulting parallel code then executes on a VLIW processor. These two approaches are not mutually exclusive, and profitable combinations have been studied and built [51, 74, 87]. Indeed, to reduce hardware complexity and improve performance, most superscalar processors now employ a compiler similar to that for VLIW machines. Thus from the perspective of ILP compilation, the processor type does not matter; whether for a superscalar or VLIW processor, compilers can analyze and transform the sequential program to increase parallelism, and schedule the program to exploit parallelism.
1.1 ILP compilation and scheduling

ILP compilation comprises two major tasks, scheduling and register allocation, as well as related machine-dependent optimizations. Since large sets of registers are available in modern processors, register allocation is of secondary importance in the ILP compilation process. This thesis focuses on the scheduling task which is crucial for fully exploiting the parallel hardware. We develop scheduling algorithms and tight bounds to improve efficiency of compiled code on ILP processors.

Scheduling algorithms assign each operation a clock cycle (or issue slot) so that dependencies between operations and resource constraints are satisfied, and the schedule’s length or some other performance measure is minimized. Traditional schedulers operate on a basic block basis and minimize the number of executed operations by eliminating redundancies [6]. This strategy is well understood and on a sequential RISC processor it effectively minimizes code runtime. However, ILP scheduling, and compilation in general, present challenges that cannot be appropriately addressed by the conventional approaches.

First, since instruction-level parallelism is generally insufficient within a single basic block, ILP compilers operate on larger scheduling regions that each consist of several consecutive basic blocks. Such ILP regions include traces [35], superblocks [57], hyperblocks [69], and decision trees/triegions [13, 48, 54]. When jointly scheduling consecutive basic blocks, i.e., several control flows, the performance objective is to minimize the total weighted completion time: for each control flow, we calculate the number of cycles from its entry point to its exit weighted (multiplied) by the flow's frequency of execution, and sum over all the control flows.
Second, ILP compilers use statistical information to predict the outcome of conditional branches. Branch profile information collected from sample runs [12] or determined by static analysis [11] first guides region formation by selecting most likely flows of control. Aggressive operation scheduling and register allocation within a region then also actively use the branch statistics.

Third, ILP compilers use a model of the target processor [31, 45]. This machine model contains the visibly exposed features of the processor and allows the compiler to evaluate the cost of any transformation and scheduling decision. The model must be similar to the target processor so the optimizations for the model produce a high-quality schedule for the processor.

These new dimensions make ILP scheduling difficult even with the simplest parallel machine model. We show that the problem of scheduling a superblock with unit latency operations on a processor with uniform functional units is NP-complete. Real processors are considerably more complex. Due to advances in VLSI technology, recent general-purpose ILP processors such as the Intel Itanium [89] and Sun Microsystems MAJC [95], as well as embedded VLIW processors such as the Philips Semiconductors Trimedia [77], Texas Instruments TMS320C6000 [94], and Analog Devices TigerSHARC [38], provide many functional units and register files of different types. They invariably support predication, speculation, and software pipelining, which complicate the scheduling task.

Furthermore, the scheduler has to deal with functional units that are not fully pipelined and complex resource usage by some operations. Certain processor resources such as register write ports and buses do not scale well, and their number is rarely sufficient to support
all sequences of operations that can be executed by the functional units. Also, even though there are available resources for the execution and completion of an operation, limited issue bandwidth prevents moving the operation along the pipeline. Thus, various resource and issue constraints on scheduling continue to represent a performance bottleneck that can offset the benefits of extra hardware. Finally, simply increasing available resources may not always allow the compiler to produce better schedules. It is well known to researchers in scheduling that a multiprocessing system consisting of identical processors may exhibit unexpected anomalies; for instance, increasing the number of processors can actually increase the time to execute a given set of jobs [42, 43].

Current schedulers address these restrictions in several ways. Some schedulers, notably those for superscalar processors, use a simplified machine model, leaving the hardware responsible for ensuring that all dependences and resource constraints are satisfied [60]. Hardware interlocks insert stall cycles that in some cases could be avoided by a more resource-aware compiler.

In contrast, VLIW schedulers are usually exposed to all architectural and microarchitectural features of the target processor but they tend to focus on one processor feature (resource conflict) at a time and not on overall performance. For example, Fogel and Dehmert describe a code motion technique for reducing the effect of anti-dependencies introduced by unique dedicated resources [37]. Their scheduling technique is motivated by the special memory staging registers present in the Silicon Graphics GE11 graphics processor. An instruction scheduler for the TMS320C6201 processor explicitly deals only with the restrictive interconnection network between the processor’s two datapaths A and B [67]. Since a single
value is allowed to transfer from A to B and similarly from B to A within a cycle, the proposed scheduler incorporates a partitioning technique. Hoogerbrugge and Augusteijn show how an instruction scheduler for the Trimedia processor addresses memory bank conflicts and fills the three branch delay slots [52]. The three papers describe algorithms that successfully create high-quality schedules in the presence of a resource conflict; however, they use ad hoc techniques. There is no unified approach to address the different types of scheduling restrictions.

Unlike basic block schedulers, most ILP schedulers do not assess the quality of the partial or complete schedules they produce; creating a good bound for a complex processor is as difficult as creating a scheduling algorithm. If present, tight bounds on scheduler performance can be used to identify sub-optimally scheduled code, either after or during scheduling.

Backtracking schedulers may use scheduling bounds to evaluate and possibly undo scheduling decisions after they have been made. By identifying and undoing critical scheduling decisions, higher quality code can be produced. Conventional, cycle- or operation-based schedulers may use scheduling bounds to predict the effect of scheduling a particular operation in the current cycle. Tight bounds can also substantially reduce the search space of branch-and-bound schedulers, and thus increase the ability of such schedulers to solve larger instances in a shorter period of time.
1.2 Organization and overview of contributions

In this thesis we study approaches for ILP scheduling that are aware of the complexity of target processors; our algorithms and bounds thus apply for very detailed ILP processor models. We develop the backtracking approach for transforming processor-insensitive scheduling algorithms into schedulers that tightly integrate detailed processor information within the scheduling process. We design lower bounds on the schedule length to account for important resource constraints, and we propose linear programming-based heuristics to calculate operation priorities.

The thesis is organized in six chapters. Chapter 2 introduces the main dimensions of our scheduling problem: processors, operations, scheduling region, and optimality criteria. We also prove that the superblock scheduling problem is NP-complete. The next three chapters present our main work on algorithms and bounds for resource-aware instruction scheduling. Chapter 6 summarizes our results and describes directions for future research.

1.2.1 List scheduling algorithms

List scheduling algorithms for the superblock and the more general, total weighted completion time scheduling problems are the subject of Chapter 3. We first evaluate a family of combinatorial scheduling algorithms that optimally solve the uniprocessor problem, and show that they can be used to achieve good performance for a processor with multiple functional units. These algorithms are efficient and find schedules that are on average within 1.5% of optimal over a large synthetic benchmark consisting of trees, chains, and instances with no precedence constraints. We then design several ways to create feasible schedules
from non-integral solutions to a new linear programming relaxation for the multiple unit problem. The best of these linear programming-based approaches finds schedules that are within 0.2% of optimal over our benchmark.

We then formulate the superblock scheduling as a weighted completion time scheduling problem and compare our algorithms against algorithms designed specifically for superblock scheduling. We use a set of instances extracted from the SPECint95 compiler benchmark. For these instances with arbitrary precedence constraints, our best linear programming-based algorithm clearly outperforms all other known algorithms and finds schedules that for a six-issue VLIW processor are within 1.4% of optimal. We thus show that for assigning static priorities of operations, which is a necessary step in any instruction scheduler, an LP-based approach can be the best choice.

1.2.2 Backtracking schedulers

Conventional schedulers schedule operations in dependence order and never revisit or undo a scheduling decision on any operation. In contrast, backtracking schedulers may unschedule operations and can often generate better schedules. Chapter 4 develops and evaluates the backtracking approach for acyclic instruction scheduling.

We explore the design space by creating three schedulers that each backtrack in a different manner. We first develop a generic backtracking scheduling algorithm and prove that it terminates. To the best of our knowledge, this is the first termination proof for an acyclic backtracking scheduling algorithm. We then describe two more aggressive backtracking schedulers and evaluate their effectiveness. The full-backtracking FullBT scheduler enables
backtracking for all operations and unschedules already scheduled operations to make space for the current operation. For the SPECint95 benchmark, the FullBT scheduler increases the percentage of superblocks scheduled optimally over a conventional non-backtracking scheduler from an average of 66.9% to 81.4%. The selective backtracking SelectBT scheduler enables backtracking only when scheduling certain types of operations for which backtracking is likely to be advantageous. This hybrid scheduler is almost as good as the FullBT scheduler in terms of generated schedule length but backtracks about four times less often. We conclude that aggressive backtracking-based instruction schedulers can effectively improve schedule quality with a small amount of additional computation.

Our three backtracking schedulers are evaluated on a set of VLIW processors with complex resource usage of branch operations and branch delay slots exposed to the compiler. The results clearly demonstrate the ability of backtracking algorithms to produce near-optimal schedules in presence of resource constraints.

1.2.3 Resource-aware bounds

Chapter 5 develops resource-aware lower bounds for superblock and basic block scheduling. We first evaluate known combinatorial lower bounds on superblock and basic block scheduling, and then describe new tighter bounds for complex resource usage of operations, register file write ports constraints, and issue restrictions.

Basic block scheduling minimizes the number of cycles between the basic block's entry and exit points, and a tight lower bound can be obtained by replacing precedence constraints with early and late times, giving a problem that can be efficiently solved. We prove that
recursively applying this approach yields a bound that is tighter than other known bounds, and we experimentally show that the bound achieves the optimal value for 86.5% of the time. We use the basic block bounds to derive efficient and practical bounds for superblocks where the performance objective is to minimize the average weighted completion time, i.e. the number of cycles from the entry point to each exit weighted by the exit probability.

We propose a uniform method for extending the tightest bound to handle common structural constraints and microarchitectural features in recent microprocessors. In particular, we design lower bounds that are aware of complex resource usage, register file write ports conflicts, and issue restrictions. Our new bounds are based on augmenting the dependence graph of the scheduling region with resource-type edges and nodes that model as closely as possible the resource usage in consideration. Optimally solving the relaxed problem by working with the modified dependence graph gives a tight lower bound.

This thesis demonstrates that the scheduler quality can be improved by systematically including information about resource and issue restrictions in the scheduling process. Our new resource-aware scheduling bounds and algorithms enhance high performance compilers for VLIW/EPIC and superscalar processors.
Chapter 2

Scheduling framework

The three main parts of any scheduling problem are the set of operations to be scheduled, the model of the processor that executes the operations, and the optimality criterion or performance objective, all of which we define in this chapter. Our thesis concentrates on static scheduling, that is scheduling performed by a compiler. We will not consider dynamic scheduling where the hardware decides which operations are issued and it is possible to issue a varying number of operations per clock cycle.

Following VLIW terminology, operations correspond to RISC-style instructions, and instructions are a group of operations that issue in a cycle. Instruction or code scheduling is a compiler phase that assigns each operation a clock cycle (or issue slot) so that dependencies between operations and resource constraints are satisfied, and the schedule’s length or some other performance measure is minimized.
2.1 Processor model

The processor (machine) model represents the compiler’s view of the processor. The models range from very simple, e.g. assuming identical general-purpose functional units and operations of the same type, to realistic ones. Simple models are commonly used to develop provably optimal or near-optimal scheduling algorithms. Such algorithms are then modified and evaluated to become efficient heuristics for complex models.

We use a family of VLIW/EPIC processors based on the Hewlett-Packard PlayDoh architecture specification [61] and exemplified by the Intel Itanium processor [58] and Sun Microsystems MAJC architecture [95]. Explicitly Parallel Instruction Computing (EPIC) [87, 88] is an evolution of VLIW architecture. EPIC’s basic characteristics come directly from VLIW, namely multiple operations per instruction, architecturally visible register structure and latencies. EPIC processors may include a number of additional features such as heterogeneous functional units, multiple register files of the same type, shared register ports and instruction fields, predication, data and control speculation.

A basic VLIW/EPIC processor has a set of integer, floating point, memory (load/store), and branch units. A particular processor is described by a sequence of numbers, e.g. 3221 denotes a processor that can issue up to three integer operations, two floating point operations, two memory operations, and one branch operation. All functional units are fully pipelined. Each processor instruction consists of a set of RISC-style machine operations such that the number of operations of a certain type does not exceed the number of functional units of that type.

A MAJC processor is even simpler than the basic VLIW/EPIC processor. It has four
identical general-purpose functional units, called instruction slices, each capable of executing any type of operation. Such a processor is described using the notation GPr, where r denotes the number of identical general-purpose functional units.

The basic VLIW/EPIC processor model omits many key issue and resource constraints that can affect scheduler performance. Certain operations may have a complex resource usage pattern. For example, a branch uses an integer functional unit in a generic VLIW processor [80], or a branch is stored not only in the memory buffer but also in the ALU buffer of the processor instruction reorder unit as in the Hewlett-Packard HP-8000 processor [56]. As a result, issuing a branch operation may affect the scheduler's ability to issue an integer operation in the same or later cycle.

A different type of resource sharing that restricts scheduling is seen in the Silicon Graphics GE11 graphics processor. This processor has three types of memory components on chip, and a value is loaded or stored in two steps. To load a value from memory the address is first sent to memory, and then the result is read from a special load staging register. Similarly, stores use a store staging register. Data is first copied to that register, and then a store instruction specifies the address to which the data is stored. Each of the two memory staging registers represents a bottleneck resource [37].

Another processor feature not included in the basic model are register read/write ports. While recent processors provide enough register ports and buses to support the maximum issue width, longer latency operations can cause the available register write ports to be oversubscribed. When an operation of this type is issued earlier than an operation with a short latency a register write port conflict may result. Examples of these types of operations
include the floating-point divisions and square roots in the Trimedia processor [77].

The Itanium processor can issue up to six operations, grouped in two instructions, in a cycle. However, some pairs of instructions cannot be issued together without stalling the processor. For example, an MMF type instruction (a floating-point and two memory operations) causes a stall if it is paired with any other instruction. There are also issue rules, e.g., a floating-point (FP) operation from the first instruction executes on the first FP unit, and an FP operation from the second instruction executes on the second FP unit.

In summary, operation issue restrictions, resource sharing, and register write port conflicts represent structural/resource hazards spanning all pipe stages in execution of an operation. Such hazards need to be considered as resource constraints on scheduling, and therefore we include them in our machine models. We refer to the basic model augmented with the above resource constraints as an extended VLIW/EPIC processor model.

2.2 Scheduling region

Before the advance of ILP processors, basic blocks had been the main scheduling region for many years. However, a single basic block, i.e., a maximal sequence of operations in which control flow enters at the beginning and leaves at the end of the sequence, does not generally provide sufficient instruction-level parallelism. ILP schedulers therefore operate on a region larger than a basic block.

For concreteness, we assume that the scheduling unit is a superblock [57], i.e., a sequence of consecutively executed basic blocks with a single entry point (at the entry point of the first block in the sequence) and multiple exit points (at the exit point of each block in the
sequence). Two other linear scheduling regions are extensions of the superblock. A trace [35] has not only multiple exits but also multiple entry points; trace scheduling requires complex bookkeeping and adding compensation code. A hyperblock [69] is formed by if-conversion and can be viewed as a superblock of predicated basic blocks. Because of its relative simplicity, the superblock is the most popular choice of a scheduling region while still providing a sufficient amount of ILP [23, 28, 30].

A superblock is passed to the scheduler as a dependence graph, which is a directed acyclic graph (DAG). The operations to be scheduled form the nodes in the DAG, and any dependence (data or control) between two operations is expressed by an edge between the corresponding nodes [93]. A control dependence arises from the control flow of the program while a data dependence arises from the flow of data between operations.

In this thesis, we assume an infinite number of registers, or equivalently we consider the pre-pass scheduling phase only. (This assumption is not too restrictive since for EPIC processors most operands are in registers and register spills are infrequent [81].) Thus the only data dependencies present in the DAG are true data dependencies, that is, when an operation uses a result produced by a previous operation. The superblock DAG also contains profile information; each exit (branch operation) \( b \) has a weight \( w_b \) indicating how often the branch is taken in a sample run.

### 2.3 Performance objective

In a valid schedule, each operation is assigned to an issue slot subject to the following constraints:
• *dependence constraints* are satisfied, i.e. for each data or control dependence edge the difference between the issue cycle of the destination and source operations is not less than the edge latency, and

• *resource constraints* are satisfied, e.g. the number of operations of a certain type scheduled in a particular cycle does not exceed the number of units of that type, or the number of operations that write to a register does not exceed the number of register file write ports.

In a valid schedule, the start (issue) time and completion time of an operation $i$ are denoted $S_i$ and $C_i$, respectively. Scheduling in this thesis is non-preemptive: once issued, any operation runs to completion. The scheduler minimizes the expected completion time of each superblock. The true superblock completion time for a particular input can be determined at runtime only. Thus, schedulers assume that each operation has a particular weight, i.e. frequency of execution, and attempt to find a schedule that minimizes the total weighted operation completion time (WCT).

The execution frequency of an operation can be approximated by a static program analysis [11] or by profiling using actual runs with sample input data [12]. The superblock WCT is obtained by summing up contributions of each of its exits (branches). The branch contribution is the product of the number of times the branch is taken during profiling (i.e. the branch's weight) and its completion time. The completion time of a branch is the sum of the branch's issue time and the branch latency. Formally, if $SCh$ denotes a valid superblock
schedule, then $WCT(Sch) = \sum_{b \in \text{branches}} w_b C_b$ and our objective becomes:

$$\min_{Sch} \sum_{b \in \text{branches}} w_b C_b.$$ 

2.4 Early and late times

As a preprocessing step, most schedulers calculate early and late times for each operation; these times are used and possibly recalculated during the scheduling process. We distinguish between static and dynamic early/late times. The static early time of an operation $i$, $E_i$, is the earliest cycle that it can be issued on a processor with infinite resources. The static late time of an operation $i$ with respect to a branch $b$, $L_i(b)$, is the latest cycle that the operation can be issued on a processor with infinite resources while still issuing $b$ at its static early time. Clearly, the static early and late times of an operation depend only on the dependence graph. The early times can be calculated using a top-down traversal of the graph, and the late times can then determined using a bottom-up graph traversal.

The dynamic early time of an operation is the maximum of its static early time and the earliest cycle the operation can be issued on an infinite resource processor while satisfying all dependence edges from currently scheduled predecessors. Similarly, the dynamic late time of an operation is the latest time the operation can be issued on an infinite resource processor while satisfying all dependence edges to currently scheduled successors. Note that the dynamic early and late times of an operation change as scheduling progresses.

We can tighten the early and late times, both static and dynamic, by considering not only the relations among the operations in the dependence graph, but also the finite resources
of the target processor. We discuss these resource-aware bounds in Chapter 5.

2.5 NP-completeness of superblock scheduling

We conclude this chapter by demonstrating the NP-completeness of superblock scheduling. Following Chekuri et al. [24], an S-graph is the graph theoretic abstraction of a superblock. It is a direct acyclic graph (DAG) \( G = (V, E, v_{sink}, P, W) \) where \( V \) represents the set of superblock operations and \( E \) - the set of immediate dependence edges between the operations. The graph has a single sink \( v_{sink} \), and only the vertices along a single path \( P \) from a source to the sink have weights assigned by a weight function \( W(v) \). The weights represent the exit probabilities of the corresponding branch operations. We assume unit processing times for all operations (vertices), that is, \( p_i = 1 \).

**S-GRAPH SCHEDULING**

INSTANCE: An S-graph \( G = (V, E, v_{sink}, P, W) \), a positive integer \( m \) of identical processors, and a positive real number \( D \).

QUESTION: Is there an \( m \)-processor schedule such that WCT of \( G \) is \( D \) or less?

**Theorem 1** S-GRAPH SCHEDULING is NP-complete.

*Proof.* S-GRAPH SCHEDULING is in NP because a feasible schedule, if it exists, can be exhibited and checked for feasibility in polynomial time.

We now transform the DAG makespan (basic block) scheduling problem, \( m|\text{prec} = \text{DAG}, p_i = 1|C_{\text{max}} \), which is NP-complete [66] to the S-GRAPH SCHEDULING problem \( m|\text{prec} = S - \text{graph}, p_i = 1|WCT \). Let a DAG \( G_1 = (V_1, E_1) \) and a positive integer \( D_1 \)
constitute an arbitrary instance of DAG makespan scheduling. We construct an instance of S-GRAPH SCHEDULING (an S-graph $G = (V, E, v_{\text{sink}}, P, W)$ and a positive real number $D$) as follows. $V = V_1 \cup v_{\text{sink}}$, $E = E_1 \cup \{ \text{additional edges that connect all sinks of } G \text{ to } v_{\text{sink}} \}$, and $P$ is an arbitrary path from a source to $v_{\text{sink}}$. $W(v_{\text{sink}}) = 1$, and for all other vertices $v$ of $P$, $W(v) = 0$. Finally $D = D_1 + 1$. This construction can be done in polynomial time.

We will show that there is a schedule $Sch_1$ for $G_1$ such that $C_{\text{max}}(Sch_1) \leq D_1$ if and only if there is a schedule $Sch$ for $G$ such that $WCT(Sch) \leq D$.

$\implies$ Assume that there exists a schedule $Sch_1$ for $G_1$ such that $C_{\text{max}}(Sch_1) \leq D_1$. Let a schedule $Sch$ for $G$ be exactly the schedule $Sch_1$ plus the vertex $v_{\text{sink}}$ at its end. We show that $WCT(Sch) \leq D$.

$WCT(Sch) = W(v_{\text{sink}}) \times C_{v_{\text{sink}}} = C_{v_{\text{sink}}}$ because $v_{\text{sink}}$ is the only vertex with a non-zero weight ($C_{v_{\text{sink}}}$ denotes $v_{\text{sink}}$’s completion time in $Sch$). Since $v_{\text{sink}}$ must be the last vertex in any feasible schedule for the S-graph, $C_{v_{\text{sink}}} = C_{\text{max}}(Sch_1 \setminus v_{\text{sink}}) + p_{v_{\text{sink}}} = C_{\text{max}}(Sch) + 1 \leq D_1 + 1 = D$, i.e. $WCT(Sch) \leq D$.

$\Longleftarrow$ Assume that there exists a schedule $Sch$ for $G$ such that $WCT(Sch) \leq D$. Let a schedule $Sch_1$ for $G_1$ be exactly the schedule $Sch$ without its last vertex which is $v_{\text{sink}}$. Then $C_{\text{max}}(Sch_1) \leq D_1$. Indeed, using similar arguments as above, $C_{\text{max}}(Sch_1) = C_{v_{\text{sink}}} - p_{v_{\text{sink}}} = WCT(Sch) - 1 \leq D - 1 = D_1$. \hfill $\square$

Having defined the basic elements of the scheduling framework, we are ready to present our resource-aware algorithms and bounds. We begin with the general class of list scheduling algorithms.
Chapter 3

List scheduling algorithms

List scheduling is a simple yet powerful technique for solving a wide range of scheduling problems. Such algorithms first assign priorities to the jobs/operations, and then use the list scheduling policy to assign each job a start time. In this chapter, we describe and experimentally evaluate list scheduling algorithms for the superblock scheduling problem as well as for the more general, weighted completion time (WCT) scheduling problem. Our results demonstrate that explicitly parallel linear programming (LP)-based approaches clearly outperform the more traditional, single machine-based algorithms, for both WCT and superblock scheduling. Thus for applications where compilation times are not a limiting factor, substantial improvements can be achieved by using LP-based algorithms.

In designing our algorithms we draw many insights from scheduling theory, an area of operations research and computer science concerned with the optimal allocation of scarce resources to activities over time. The classical problem of scheduling jobs on machines (uniprocessors) bears a close similarity to instruction scheduling within a compiler. In fact,
many code optimization problems for pipelined and parallel processors can be modeled as deterministic scheduling problems.

Our motivation for using results from job scheduling came from three sources. First, most researchers agree that basic block instruction scheduling is generally a well-solved problem, even for complex machine models [80]. This satisfactory status has been largely obtained by incorporating many algorithms and notions from scheduling theory. The consensus reached is that list scheduling using critical path priorities is relatively efficient and near-optimal most of the time [4]. The basic critical path algorithm was developed by Hu [55] for allocating men to tasks.

Second, a recent study by Chekuri et al. [23] proposed a successful superblock heuristic, referred to as G* here, by reformulating earlier results in job scheduling by Sidney and Lawler in the context of instruction scheduling. In the 70’s, Sidney developed a set of algorithms for optimally solving the WCT problem for a uniprocessor [90] which were improved and generalized by Lawler [65].

Third, our own experimental study [8] demonstrated that linear programming (LP)-based list algorithms substantially outperform combinatorial list algorithms for the WCT scheduling problem. Based on the conclusions of this study, we conjectured that for the superblock scheduling problem, a special case of the WCT problem, our LP-based list algorithms would experimentally outperform the best known list algorithm G*.

The very recent results have confirmed our conjecture: while the average difference between the cost of G* schedules and the cost of the LP lower bound is 12.2%, the corresponding difference for the LP-based schedules is only 11.2%.
This chapter is organized as follows. We begin by stating the weighted completion time scheduling problem, and then present and evaluate two groups of list scheduling algorithms for solving it, combinatorial and LP-based algorithms. In Section 3.2, we formulate superblock scheduling as a special case of the WCT problem, and apply a set of the WCT algorithms to schedule superblocks. We also evaluate combinatorial algorithms specifically designed for the superblock scheduling problem. Section 3.3 determines the least complex class of precedence constraints for which list scheduling algorithms may not be able to find an optimal solution.

3.1 WCT algorithms

A basic scheduling problem is \( P|\text{prece}| \sum w_i C_i \) [44], the problem of scheduling \( n \) jobs under precedence constraints on \( m \) identical parallel machines to minimize total weighted job completion time. Each job \( i \) has a non-negative weight \( w_i \) and processing time \( p_i, i = 1, \ldots, n \). As in Section 2.3, the start and completion times of job \( i \) in a schedule are denoted \( S_i \) and \( C_i \), respectively. Similarly, a DAG describes the precedence constraints among the jobs such that an edge from job \( i \) to job \( j \) in the graph implies that \( C_i \leq S_j \). In a valid schedule no more than one job executes on any machine at any time, each job is scheduled non-preemptively, and the precedence constraints are satisfied. The goal of the total weighted completion time problem is to find a feasible schedule of the \( n \) jobs that minimizes \( \sum_{i=1}^{n} w_i C_i \).

Clearly, the WCT problem generalizes the superblock problem as defined in Chapter 2; while a superblock has nodes with a non-zero weights on a single path only, in the WCT
problem all nodes are allowed to have non-zero weights.

The WCT problem is NP-hard even for fixed $m = 2$ and the empty precedence graph [20], but it can be solved efficiently if $m = 1$ for certain classes of precedence constraints. If the precedence graph is empty, an optimal schedule is found by sequencing the jobs in non-increasing order of $w_i/p_i$, giving a largest ratio first (LRF) schedule [92]. Generalizations of this approach find optimal schedules for chains, trees, and series-parallel precedence graphs in $O(n \log n)$ time [53, 90, 5, 65], but the single-machine WCT problem with arbitrary precedence constraints is NP-hard [65].

Two general approaches have been traditionally employed for solving the WCT problem. Enumerative methods most often use integer linear programming (ILP) formulations of the problem and rely on branch-and-bound techniques for solving it. These techniques do not run in polynomial time and therefore only small problem instances can be optimally solved. For example, for the WCT problem without precedence constraints, $P|| \sum w_i C_i$, Belouadeh and Potts [15] show that a branch and bound algorithm based on a Lagrangian relaxation can solve instances for $n \leq 30$.

On the other hand, approximation algorithms run efficiently and come with performance guarantees. More specifically, a $\delta$-approximation algorithm is a polynomial-time algorithm that for every problem instance finds a solution whose cost is within a factor $\delta$ of optimal. Kawaguchi and Kyan [62] show that the LRF algorithm gives a $1.21$-approximation for $P|| \sum w_i C_i$, but until recently little was known about approximation algorithms for the general WCT problem. Hall et al. [47, 46] present a $(3 - 1/m)$-approximation algorithm for the case of unit job processing times, $P|\text{prec}, p_i = 1| \sum w_i C_i$, and a $7$-approximation
algorithm for $P|r_{i}, prec| \sum w_{i}C_{i}$. Chekuri et al. [24] give 4- and 2-approximation algorithms for series-parallel and in-tree precedence graphs, respectively. Baev and Meleis [7] analyze the quality of the single-machine-optimal algorithm for chains, $P|chains| \sum w_{i}C_{i}$, and show a worst-case bound of 1.33 when the ratio of $w_{i}/p_{i}$ is the same for all jobs.

There has been recent interest in experimentally evaluating the quality of solutions provided by approximation scheduling algorithms. Savelsbergh, Uma, and Wein [85] apply linear programming-based approximation algorithms to the single-machine WCT problem without precedence and with release dates. They show that the best algorithms produce near optimal solutions in $O(n \log n)$ time. Our work extends their study by evaluating the performance of a class of known scheduling algorithms when applied to the general WCT problem. The algorithms are divided into two groups: efficient algorithms that solve single machine WCT problems, and algorithms based on LP relaxations.

Both groups of our algorithms are based on list scheduling. They first assign priorities to the jobs, i.e. make a list of jobs in non-decreasing priority order, and then use the list scheduling policy to assign each job a start time. A job $i$ is ready at any point in the schedule if every job that has precedence over $i$ has completed execution. When a machine becomes free, the highest priority ready job that has not begun or completed execution is assigned to that machine.

3.1.1 Single-machine optimal algorithms

Single machine scheduling problems for WCT are well understood, and efficient algorithms exist for a range of special cases. We describe algorithms that apply when the precedence
graph is either empty, a set of chains, or a tree, and give the worst-case performance ratio for each.

The largest ratio first (LRF) algorithm finds a minimum cost schedule when the precedence graph is empty and \( m = 1 \) [92]. This can be seen by the following argument: if there is a job \( i \) that is immediately followed in the schedule by job \( j \) such that \( w_i/p_i < w_j/p_j \), then interchanging \( i \) and \( j \) in the schedule reduces the cost of the schedule by \( w_ip_i - wjp_j = p_ip_j(w_j/p_j - w_i/p_i) \), which is positive.

**LRF algorithm**

1. The priority of every job \( i \) is \( \rho_i = w_i/p_i \).

2. Apply list scheduling.

The algorithm does not necessarily find a minimum cost schedule if \( m > 1 \) or if there are precedence constraints. Furthermore, the cost of the schedule found can be arbitrarily many times worse than the optimal schedule [9]. It is this poor worst-case performance that makes the LRF algorithm unappealing for solving the general WCT problem. Furthermore, our experiments show that the LRF algorithm gives the worst results of the algorithms we consider (see Section 3.1.3).

Several algorithms are known to solve the single-machine WCT problem for special classes of non-empty precedence constraints [53, 90, 5, 65]. The decomposition approach of Sidney subdivides the precedence graph into modules and then uses optimal schedules of each module to construct the final schedule. We selected this approach because it is widely applicable and easy to implement.

We first describe an algorithm that applies when the precedence constraints form a set
of \( r \) chains. In this case every job has no more than one immediate predecessor and no more
than one immediate successor. Let \( n_i \) be the number of jobs in the \( i \)th chain, and let \( i : j \)
represent a job where \( i \) is the chain number and \( j \) is the job’s position within the chain.
Then for any \( i : j \) such that \( j \in [2, n_i] \), job \( i : (j - 1) \) is an immediate predecessor of job
\( i : j \).

We also define \( \text{Pred}(i : j) = \{ i : k | 1 \leq k \leq j \} \). That is, \( \text{Pred}(i : j) \) equals the job \( i : j \)
and all of its predecessors. We let \( H_i \) equal the set of jobs in the \( i \)th chain. For a set of jobs
\( U \) we define the ratio \( \rho(U) = \sum_{i \in U} w_i / \sum_{i \in U} p_i \).

\( \rho \)-max algorithm for chains

1. Begin with an empty priority list \( \beta \).

2. Repeat the following until the chains are all empty:

   (a) For every nonempty chain \( H_i \), let \( \delta_i = \max_j \{ \rho(\text{Pred}(i : j)) \} \).

   (b) Let \( \delta = \max_i \{ \delta_i \} \), and let \( i^* \) and \( j^* \) be the largest values such that \( \rho(\text{Pred}(i^* : j^*)) = \delta \).

   (c) Add the jobs in \( \text{Pred}(i^* : j^*) \) in order to the end of \( \beta \), and remove these jobs
   from \( H_{i^*} \).

3. Apply list scheduling, using \( \beta \) as the priority list.

This algorithm can be implemented to run in \( O(n \log n) \) time [65]. The set of consecutive
jobs selected in step 2(b), referred to as a module, minimizes the ratio over all other such
sets. A schedule is called module-intact if every module is scheduled entirely on a particular
machine and all jobs in a module are consecutively scheduled. Sidney [90] shows that for
arbitrary precedence constraints, an optimal module-intact single-machine schedule always
exists. However in this general case finding modules and scheduling within a module can be difficult. When the precedence graph is a set of chains, both steps can be implemented efficiently, as shown in the ρ-max chain algorithm.

When \( m = 1 \), the priority list and the final schedule are identical and the cost is minimized. However the chain algorithm does not always find minimum cost schedules for \( m > 1 \), as shown in [9].

A variation of this chain algorithm can also be used to find optimal schedules for \( m = 1 \) when the precedence graph is a tree where each job has at most one immediate successor (i.e. an in-tree). Jobs that have more than one immediate predecessor are called branch jobs, and a branch job that does not have a direct or indirect predecessor that is a branch job is called an initial branch job. Note that initial branch jobs depend only on chains, and if there are no branch jobs the precedence graph must be a single chain. Here we let \( \text{Pred}(i) \) equal all predecessors of job \( i \).

**ρ-max algorithm for trees**

1. Repeat the following until there are no remaining branch jobs:

   (a) Select an initial branch job \( i \).

   (b) Use the chain algorithm to optimally order the jobs in \( \text{Pred}(i) \) and set \( \beta \) equal to this order.

   (c) Add a precedence arc \((j, k)\) to the precedence graph if job \( j \) precedes job \( k \) in \( \beta \).

2. Create a priority list using the total ordering of jobs, and apply list scheduling.

The tree algorithm applies the chain algorithm to initial branch jobs, replacing two chains with a single chain composed of a series of modules. This entire procedure can be
implemented to run in $O(n \log n)$ time [65]. Chekuri et al. [24] show that the worst-case performance ratio for the $\rho$-max algorithms applied to chains and trees is 2.

3.1.2 Linear programming-based algorithms

There has been recent interest in using optimal solutions to linear programming relaxations of scheduling problems as the basis for approximation algorithms [47, 46, 24]. These techniques typically solve an easier, less constrained version of a scheduling problem. That schedule is then used as a priority list for a list scheduler that produces the final schedule. Savelsbergh, Uma, and Wein [85] show that such algorithms can find high quality solutions for the single machine WCT with release dates, but without precedence. We use a group of similar ideas to produce solutions to the general WCT problem from solutions to the LP relaxation.

We formulate the WCT problem as follows,

$$\text{minimize} \sum_{i=1}^{n} \sum_{t=1}^{T} w_t \cdot x_{it} \cdot t$$ \quad (3.1)

subject to

$$\sum_{t=1}^{T} x_{it} = 1, \quad (i = 1 \ldots n) \quad (3.2)$$

$$\sum_{i=1}^{n} \sum_{t'=1}^{T} x_{it'} \leq m, \quad (t = 1 \ldots T) \quad (3.3)$$

$$\sum_{i=1}^{T} (x_{it} \cdot t) \leq \sum_{j=1}^{T} (x_{jt} \cdot t) - p_j, \quad ((i, j) \in R) \quad (3.4)$$

$$x_{it} \in \{0, 1\}. \quad (3.5)$$
The binary variables \( x_{i,t} \) indicate whether job \( i \) completes at time \( t \), and parameters \( p_i \) and \( w_i \) give the processing time and weight of job \( i \). Constraints (3.2) and (3.3) ensure that every job completes exactly once and that no more than \( m \) jobs are executing at any time. There is an arc \((i, j)\) in \( R \) for each precedence arc from job \( i \) to job \( j \), and for each such arc constraint (3.4) ensures that the completion time of \( i \) is at least time \( p_j \) before the completion time of \( j \). The formulation minimizes the objective function (3.1) which gives the weighted sum of the completion times of the jobs. This formulation has approximately \( n \cdot T \) variables, where \( T \) is the sum of the latencies of all the jobs. As a result, memory restrictions prevent the model from solving large WCT instances on available computing resources.

We use this formulation to find exact solutions for small instances of the WCT problem. We compute a lower bound on the optimal value of the objective function for large instances using a relaxation of this formulation where the variables \( x_{i,t} \) can take on values between 0 and 1. We show in [9] that this bound is very tight and we use it to evaluate the quality of the solutions found by the scheduling heuristics.

The non-integral solutions to the WCT problem found by the relaxed formulation are not necessarily solutions to the exact formulation. The relaxed formulation allows multiple jobs to execute simultaneously on a machine. In this case, constraint (3.2) ensures that for each job \( i \), the sum of \( x_{i,t} \) over all time periods \( t \) equals 1. However, the completion time of the \( i \)th job may now be distributed over many time periods. Consider for example some job \( i \) with \( p_i = 3 \), and \( x_i = (0, 0, .1, 0, 0, .1, 0, 0, 0) \) in a solution to the relaxed formulation. This indicates that .1 of job \( i \) is executing from time 1 to time 3, .1 is executing from time 4 to
time 6, and 8 is executing from time 6 to time 8. Notice that each fractional component of job \( i \) has the same processing time, and that two of the components overlap at time 6. By constraint (3.3), the total number of fractional components executing at any time does not exceed the number of machines, and constraint (3.4) ensures that for each arc \((i, j)\) in \( R \) the average completion time of job \( i \) does not exceed the average start time of job \( j \). The average completion time \( \sum_{t=1}^{T} (x_{it} \cdot t) = 7.3 \) is the sum of the completion times of the fractional components of job \( i \), weighted by the fractional size of each component. The average start time is then computed by subtracting the processing time of job \( i \).

An alternate relaxation of the WCT scheduling problem allows jobs to execute preemptively while only allowing one job to execute on a machine at any time. This formulation has been successfully used to develop approximation algorithms for the single processor WCT problem without precedence. The preemptive relaxation is appealing in this case because the preemptive WCT scheduling problem under these conditions is solvable in polynomial time. The exact solution to the preemptive problem is therefore a good starting point for solving the non-preemptive problem. However, the general WCT problem with preemption remains intractable.

We use a non-integral solution to the WCT problem to construct priority lists that are then list scheduled to give a feasible schedule. There are several ways that the priority lists can be constructed. For some fixed \( \alpha \in [0, 1] \), an \( \alpha \)-schedule is created using the following algorithm.

\[ \text{\textbf{\( \alpha \)-scheduling algorithm}} \]

1. Solve the relaxed integer-programming formulation
2. For each job \( i \), select the earliest time \( t_i \) such that \( \sum_{j=1}^{t_i} x_{ij} \geq \alpha \).

3. Set the priority of the \( i \)th job equal to \( t_i - p_i \) and apply list scheduling.

The second step selects for each job the earliest time where at least an \( \alpha \) fraction of that job has completed executing. If \( \alpha = .01, .5, \) or 1.0, then the respective completion times are \( t = 3, 8, \) or 8 for the job considered above. Note that setting \( \alpha = 0 \) always gives a priority of \( -p_i \) for job \( i \). The third step creates a priority list based on the corresponding job start times \( t = 0, 5, \) or 5. The \( \alpha \) parameter allows the scheduler to produce a family of related feasible schedules that are derived from a single solution to the relaxed formulation. For example, \( \alpha = .01 \) gives each job a priority that closely approximates the earliest time when that job executes in the relaxed formulation. Similarly, \( \alpha = 1 \) gives each job a priority equal to the latest time when that job executes in the relaxed formulation. Values of \( \alpha \) between these extremes allow job priorities to be determined by different fractions of job start times. Finally, list scheduling creates a feasible \( m \)-machine schedule.

In our paper [9] we evaluate the quality of \( \alpha \)-schedules for a range of fixed \( \alpha \) values between .01 and 1.0. We also evaluate an algorithm, \( \text{Best-\( \alpha \)} \), that returns the minimum cost \( \alpha \)-schedule. A related scheduling algorithm lets a job's priority be its average start time in the non-integral schedule, as described below.

**Average start-time algorithm**

1. Solve the relaxed integer programming formulation

2. Set the priority of the \( i \)th job equal to \( \sum_{j=1}^{\tau} (x_{ij} \cdot t) - p_i \) and apply list scheduling

The priorities computed by the Average start-time algorithm are similar to those found by the \( \alpha \)-scheduler with \( \alpha = .5 \). However the Average start-time algorithm computes a
true average start time for each job, (7.3 in the above example) while the $\alpha$-scheduler with $\alpha = .5$ finds the earliest time such that half the job has been scheduled (8 in the example). The $\alpha$-scheduler selects some priority equal to $t$ such that $x_{ijt}$ is non-zero, while the Average start-time algorithm can select a time when a job is not executing at all or a non-integral time.

Note that while the single-machine-optimal algorithms differ for different types of precedence, and there is no such algorithm for DAGs, the LP-based algorithms above can deal with arbitrarily complex precedence relations.

3.1.3 Experimental evaluation

We evaluate the single-machine-optimal (SMO) and LP-based algorithms on a set of 240 synthetic WCT instances. An instance consists of a set of $n$ jobs, each with a processing time and a weight, a precedence graph on the jobs, and a set of $m$ identical parallel machines. Twenty random instances are generated for each $n = 40, 60, 80, 100, 120, 140,$ and 160, with $m = 2, 4,$ or 6 for each. The processing time of each job is uniformly distributed in $[1, 5]$, and the weight of each job is uniformly distributed in $[1, 100]$. We randomly select precedence constraints for each instance such that every unique graph is equally likely to occur.

We use the relaxed (non-integral) version of the IP formulation from Section 3.1.2 to compute a lower bound on the cost of the optimal schedule. We have shown in [9] that the relaxed formulation gives a very tight lower bound, usually within a fraction of a percent of the optimal value. For a particular tree instance, the bound may understate the optimal value by several percent, but over a wide range of instances sizes we can reliably use the
Table 3.1: Quality of single-machine-optimal and LP-based algorithms vs. LP lower bound for WCT instances (1.0 = 1%)

bound to evaluate the quality of our heuristics.

The average percentage difference between the heuristics and the lower bound is shown in Table 3.1. The results indicate that without precedence, the SMO algorithm finds schedules that are on average 0.03% from the optimal value. When this algorithm is applied to chains and trees, the average difference rises to 0.40% and 5.40%, respectively.

The average percentage differences between the costs of the Best-α schedules and Average start-time schedules, and the cost of the lower bound, are also shown in Table 3.1. We compute α-schedules for 10 values of α between 0.1 and 1.0 in increments of 0.1. We also use α = 0.01 to give priorities equal to the earliest starting time for each job. The Best-α algorithm then selects the best α-schedule for each instance. The Average start-time algorithm computes a weighted average of the fractional start times for each job.

The Average start-time algorithm finds schedules that are on average 0.11% and 1.03% from the lower bound for chains and trees, respectively. Much of these differences can be attributed to the gap between the lower bound and the optimal value. The Best-α algorithm
finds schedules that are on average $0.07\%$ and $0.94\%$ from the lower bound for chains and trees, respectively. Varying $\alpha$ introduces small perturbations in the priorities which allows the scheduler to find a locally optimal schedule that outperforms the schedule found for any fixed $\alpha$. It is significant that the Average start-time algorithm does an order of magnitude less work to transform non-integral solutions into feasible schedules, and finds schedules that are only slightly worse than those found by the best-$\alpha$ algorithm.

The higher quality of the LP-based schedules overall can be explained by the fact that the priority lists are constructed while taking into account the presence of multiple resources. In contrast, the priority lists used by the SMO algorithms are created by scheduling on a single machine.

### 3.2 Superblock algorithms

In this section we formulate the superblock instruction scheduling problem as a special case of the WCT job scheduling problem. We can then automatically apply all WCT algorithms. In addition, two combinatorial algorithms specifically designed for superblock scheduling are described. A thorough evaluation of all the algorithms completes the section.

In scheduling operations in a region larger than a basic block, we are given $n$ operations under precedence constraints and a processor with $m$ functional units. Each operation $i$ has a non-negative weight $w_i$ and processing time $p_i$, $i = 1, \ldots, n$. The start time of operation $i$ in a schedule is denoted $S_i$ and the completion time is denoted $C_i$. A directed acyclic graph describes the precedence constraints among the operations such that an edge from operation $i$ to operation $j$ in the graph implies that $C_i \leq S_j$. In a feasible schedule no more
than one operation executes on any functional unit at any time, each operation is scheduled non-preemptively, and the precedence constraints are satisfied. Given a superblock as a scheduling region, we let each branch operation's weight equal the probability that branch is the last operation executed in the superblock. All other operations have a weight of zero.

The goal of the superblock scheduling problem is to find a feasible schedule of the n operations that minimizes \( \sum_{i=1}^{n} w_i C_i \), or equivalently, \( \sum_{b \in \text{branches}} u_b C_b \).

### 3.2.1 Combinatorial algorithms

We have shown that the superblock scheduling problem for multiple machines is NP-hard. Even for a single machine, the more general WCT scheduling problem is also NP-hard. However, due to the fixed number of non-zero weight nodes along a single path in the superblock dependence graph, the superblock single-machine problem can be efficiently solved.

An optimal algorithm, called Successive retirement, assigns the highest priority to the operations in the first basic block in the superblock, the second highest priority to the operations in the second block, and so on. Since each branch is scheduled as early as possible, the successive retirement algorithm clearly produces a single-machine optimal schedule. How the operations in a block are scheduled does not matter; any topological order is optimal.

We can use the single-machine optimal schedule produced by the Successive retirement algorithm as a priority list for superblock scheduling on multiple machines. Here, the order of operations of a block does matter; for our experiments we choose the critical path policy.
similarly to other studies [23, 28, 30]. This overall procedure for superblock scheduling on multiple machines will be referred to as the single-machine optimal (SMO) algorithm.

Chekuri et al. [23] propose a successful superblock heuristic, referred to as G* here, by reformulating earlier results in job scheduling by Sidney and Lawler in the context of instruction scheduling. The G* heuristic builds on the uniprocessor optimal algorithms by applying the notion of a module, which is a set of consecutive operations. Sidney [90] shows that for arbitrary precedence constraints, an optimal module-intact single-machine schedule always exists. However in this general case finding modules and scheduling within a module can be difficult.

For a superblock, the only modules are the subgraphs rooted at the branches, which simplifies finding the critical module or branch. To determine the critical branch in a superblock, we schedule the subgraph rooted at each branch using a secondary heuristics, e.g. the critical path, and divide that partial schedule’s length by the sum of exit probabilities in the subgraph. The branch with the smallest number becomes the critical branch, i.e. the corresponding subgraph becomes the critical module, and we schedule the jobs from the critical module first. G* then removes the subgraph for the superblock DAG and proceeds recursively with the remaining operations. G* heuristic is shown to experimentally outperform all other list scheduling algorithms, including the Successive retirement [23].

3.2.2 Experimental evaluation

We evaluate the least ratio first (LRF), SMO, G*, and LP-based algorithms on 162 superblocks extracted from the SPECint95 benchmark suite. Classic compiler optimizations
are applied to each benchmark program using the IMPACT compiler. The benchmarks are then converted to the Rebel textual intermediate representation by the Elcor compiler from Hewlett-Packard Laboratories. Superblock formation is applied within the LEGO research compiler, which has been developed by the TINKER group at North Carolina State University. Dependencies among operations consist of all flow dependences, memory dependences between consecutive stores, anti and output dependences due to live values at the side exits of the superblocks. Also, control flow dependences were introduced to prevent operations from moving below branches. However, operations are free to move above branches. The exit probabilities of each branch in the benchmark was measured during runs of the SPECint95 benchmark.

The size of the 162 superblocks varies from 30 to 339 operations with an average of 92. The weights of all non-branch operations are zero. Branch operations' weights are equal to their frequency of execution and vary from 0 to $10^7$. The instances’ precedence graphs are directed acyclic graphs, and in general an operation may have more than one successor. The complex structure of the precedence makes scheduling these instances much more difficult than scheduling the synthetic instances considered in the previous section.

The average percentage differences between the costs of the LRF, SMO, and G* schedules, and the cost of the LP lower bound are shown in the upper part of Table 3.2. These combinatorial algorithms find schedules that are on average 14.2%, 12.6%, 12.2%, respectively, from the lower bound. Our results confirm that G* is the best combinatorial list scheduling algorithm for superblocks. However, even G* cannot match the quality of the Average start-time algorithm.
<table>
<thead>
<tr>
<th></th>
<th>Processor</th>
<th>GP2</th>
<th>GP4</th>
<th>GP6</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRF</td>
<td>avg</td>
<td>28.1</td>
<td>11.0</td>
<td>3.5</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>105.0</td>
<td>49.5</td>
<td>41.7</td>
</tr>
<tr>
<td>SMO (successful</td>
<td>avg</td>
<td>21.3</td>
<td>11.8</td>
<td>4.6</td>
</tr>
<tr>
<td>retirement)</td>
<td>max</td>
<td>62.0</td>
<td>48.2</td>
<td>41.0</td>
</tr>
<tr>
<td>G* with</td>
<td>avg</td>
<td>25.9</td>
<td>8.2</td>
<td>1.9</td>
</tr>
<tr>
<td>critical path</td>
<td>max</td>
<td>65.2</td>
<td>46.1</td>
<td>33.5</td>
</tr>
<tr>
<td>Average start-time</td>
<td>avg</td>
<td>24.4</td>
<td>7.9</td>
<td>1.4</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>65.4</td>
<td>46.3</td>
<td>33.6</td>
</tr>
<tr>
<td>Best-α</td>
<td>avg</td>
<td>28.1</td>
<td>12.0</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>69.0</td>
<td>50.3</td>
<td>37.8</td>
</tr>
</tbody>
</table>

Table 3.2: Quality of combinatorial and LP-based algorithms vs. LP lower bound for superblock instances (1.0 = 1%)

The average percentage differences between the costs of the Average start-time and Best-α schedules, and the cost of the same LP lower bound are shown in the lower part of Table 3.2. The LP-based algorithms find schedules that are on average 11.2% and 14.5%, respectively, from the bound. For a processor with six general-purpose units GP6, the Average start-time algorithm finds optimal solutions for 126 out of the total 162 instances, i.e. in about 78% of cases.

Unlike the WCT problem, Best-α shows the worst results among the five algorithms. One possible explanation is that the vast majority of operations in the superblock instances have a unit latency. The computation time for solving the linear program was an average of 5.9, 6.1, and 8.1 seconds per instance, for GP2, GP4, and GP6 processors, respectively.

In summary, the excellent performance of LP-based scheduling algorithms on statically scheduled processors with general purpose functional units suggests that such algorithms can be applied in compilers for more complex VLIW/EPIC processors. The time required for solving a linear program such as (3.1)-(3.5) is acceptable for embedded system design.
where the code is compiled only once but an optimal or close to optimal performance is required.

### 3.3 Limitations of list scheduling

The single-machine-optimal and LP-based algorithms considered in this chapter belong to the class of list scheduling algorithms. By construction any list schedule does not contain delays, where a *delay* is defined as a time period during which a machine is idle and there is a ready job available. However, optimal solutions to the WCT problem with arbitrary precedence constraints may include delays.

Consider the following example. We schedule the out-tree dependence graph, shown in the left side of Figure 3.1, on a processor with two non-pipelined functional units. Each operation is labeled with its processing time and weight, e.g., operation 3 has a processing time (latency) of 2 and a weight of 1. For this example, there are four possible list schedules; we show the one created by consecutively scheduling operations 1, 3, 2, 4, and 5. The cost
of all four list schedules is 37.

Note that the optimal schedule with a cost of 34 contains a delay at cycle 1: even though the first functional unit is free, and the low priority (with the largest latency but with the smallest weight) operation 2 is ready, its execution is postponed. This delay allows both higher priority operations 4 and 5 to be immediately issued when they become ready at cycle 2.

If the precedence graph is an in-tree or a set of chains, no delays are necessary. In fact, the following lemma demonstrates that no gaps are created by a list scheduler for an in-tree or a set of chains, where a gap is defined as a time period during which a machine is idle. Clearly, gaps include delays, but not vice versa.

**Lemma 2** Given an instance of the WCT scheduling problem where the precedence graph is an in-tree, an optimal schedule exists without gaps.

*Proof.* Given a minimum cost schedule that contains a gap, we show that it is possible to create another schedule containing no gaps whose cost is not larger. We start with an optimal schedule containing at least one gap and consider the last time \( t \) at which a job \( i \) is issued on a machine immediately following a gap. We let \( m_i \) be this machine.

If \( i \) does not depend on a job that completes at time \( t \), then \( i \) can be scheduled at time \( t - 1 \) without violating the precedence constraints or increasing the cost of the schedule. This transformation eliminates the gap between time \( t - 1 \) and \( t \) on machine \( m_i \), and possibly creates up to one new gap after job \( i \). Therefore the number of gaps does not increase.

If \( i \) does depend on such a job, then let \( k_c \) be the number of jobs that complete at time
$t$ and $k_s$ be the number of jobs that start execution at $t$ that depend on one of the $k_c$ jobs. We then have $k_s \leq k_c$ since at most one job can depend on each of the $k_c$ jobs. If more than $k_s$ jobs begin execution at time $t$, there must exist a job $j$ on machine $m_j$ that does not depend on any of the $k_c$ jobs that complete at time $t$. We exchange $j$ and all jobs following it on machine $m_j$ with $i$ and all jobs following it on machine $m_i$, and move job $j$ one time period earlier. This transformation does not violate the precedence constraints or increase the cost of the schedule. The gap between time $t - 1$ and $t$ on machine $m_i$ has been eliminated, and possibly up to one new gap is created after job $j$. Therefore the number of gaps does not increase.

Now consider the case where exactly $k_s$ jobs begin execution at time $t$. At time $t$, there are $k_c + 1$ machines available to start executing jobs, and only $k_s < k_c + 1$ jobs that actually begin executing. Therefore at least one machine is idle at time $t$, and this machine must also be idle from time $t + 1$ until the end of the schedule because by assumption there are no gaps after $t$. We assign $i$ and all jobs following $i$ on machine $m_i$ to that machine, eliminating the gap between time $t - 1$ and $t$ on machine $m_i$, and creating no new gaps. This transformation does not violate the precedence constraints or increase the cost of the schedule.

This process is repeated until no gaps remain by again considering the last time at which a job is issued on a machine immediately following a gap. The process must terminate because at each step either one job is moved earlier in the schedule and the number of gaps does not increase, or jobs are reassigned to another machine and the number of gaps decreases. In this way we produce a schedule without gaps without increasing the cost. □
We conclude that by using an appropriate priority assignment, list scheduling algorithms can find optimal solutions to WCT problems when the precedence graph is an in-tree or a set of chains. However, for the more general precedence constraints no list scheduler may produce a minimum cost schedule no matter what priority lists the scheduler uses.

Previous studies investigated the delay scheduling [24], where issuing a ready operation is postponed so that when a higher priority operation becomes ready there are available resources for its execution. Speculative hedge heuristic [28] schedules frequently taken branches early without needlessly delaying other branches. Another branch-based heuristic, balance scheduling [30] works by determining the operations that each branch needs to be schedule early and selecting a set of branches with compatible requirements.

In this thesis, we develop the backtracking approach to overcome the inherent limitations of list scheduling, thereby a scheduled operation can be unscheduled to make space for the current operation. The next chapter describes the class of backtracking schedulers in detail.
Chapter 4

Resource-aware backtracking

schedulers

Backtracking, or undoing previously taken decisions, is a general problem-solving strategy that has proven to be useful in diverse optimization areas. Conventional instruction schedulers consider operations in dependence order and never revisit or undo a scheduling decision on any operation. In contrast, backtracking schedulers may unschedule operations and can often generate better schedules. This chapter develops and evaluates the backtracking approach for acyclic instruction scheduling. We demonstrate that with a small amount of additional computation, backtracking-based instruction schedulers can effectively improve schedule quality for detailed processor models.

The organization of this chapter is as follows. After motivating the need of backtracking, we introduce several forcible scheduling policies which are a necessary part in any backtracking scheduler. We then present the structure of a generic backtracking scheduling algorithm.
and prove that it terminates. We describe two more aggressive backtracking schedulers in Sections 4.4 and 4.5, respectively. The full-backtracking FullBT scheduler enables backtracking for all operations and unschedules already scheduled operations to make space for the current operation. The selective backtracking SelectBT scheduler enables backtracking only when scheduling certain types of operations for which backtracking is likely to be advantageous. We present a detailed evaluation of the three schedulers in Section 4.6. An overview of related work and conclusions complete the chapter.

4.1 Motivation

Major trends in processor design highlight the need for backtracking schedulers. The first trend is towards deeper pipelines. A simple RISC processor may have five pipeline stages for instruction fetch, decode, register read, execute, and write back. Modern superscalar or VLIW processors may have as many as 15 or 20 stages. In these processors, a branch is resolved in the register read or execute phases. The true branch latency is the number of cycles from the branch’s issue until the point where the branch is resolved. Processor designs attempt to hide the branch latency by predicting the branch. However, if the branch prediction is wrong, the early stages of the pipeline have to be emptied which causes a substantial misprediction penalty. A supplement or alternative to branch prediction is to expose some or all of the latency of the branch to the compiler and to let the compiler fill the branch delay slots. Consider an arithmetic operation $i$, generating a live-out value, and a succeeding branch $b$. The latency of the dependence edge from $i$ to $b$ must guarantee that the branch does not transfer control to another scheduling region before $i$ generates
its live-out value. If the branch latency exceeds the arithmetic operation latency, this edge latency is negative, permitting \( i \) to descend below \( b \) into branch delay slots.

Current schedulers that schedule instructions cycle by cycle are not effective in handling such negative latencies and filling delay slots. Consider the example in Figure 4.1, where we assume a single-issue processor with an arithmetic operation latency of one, a load latency of one, and a branch latency of three. On the left we show the operations in the body of a while loop that scans to the end of a linked list. The \texttt{cmpeq} operation sets the predicate register \texttt{pr4} to \texttt{true} if register \texttt{r2} is 0. The \texttt{bfalse} operation branches to the \texttt{Loop} label if the register \texttt{pr4} is \texttt{false}. The edges between operations in the DAG are labeled with their associated latencies. For example, the edge from the \texttt{load} to the \texttt{bfalse} indicates that the live-out value of \texttt{r2} must be generated before control is transferred to another block. This edge latency is the difference between the \texttt{load} latency of one and the \texttt{bfalse} latency of 3. The \texttt{add} and \texttt{load} operations can descend below the \texttt{bfalse} into the branch's delay slots. However, the conventional schedulers schedule operations cycle by cycle and make sure that all predecessors of an operation are scheduled before the operation. As a result, the \texttt{bfalse} is scheduled after all other operations, and consequently its delay slots are unfilled as shown.
in Figure 4.1. The example also show the schedule produced by a backtracking scheduler. As in the conventional scheduler, we attempt to schedule the branch after the first three operations have been scheduled in consecutive cycles. The \texttt{bfalse} has a higher priority than the \texttt{add} and in the backtracking scheduler it can displace the scheduled \texttt{add}, that in turn can displace the scheduled \texttt{load} from cycle 2 to cycle 3. Both the delay slots of the \texttt{bfalse} are now filled. The schedule length can therefore be reduced from 6 to 4, for a reduction in cycles of 33%.

The second trend is towards power-sensitive processor designs for mobile applications. Effective branch prediction requires large memories to maintain a sufficient amount of branch behavior history. The power consumption of such memories accessed frequently is high and can account for a large fraction of total on-chip consumption. Therefore, exposing the branch latency and reducing/eliminating prediction hardware may be an attractive alternative for mobile processor designs.

The third trend is towards wide-issue processors. Media-intensive applications have large amount of parallelism that can be effectively exploited by processors that issue many operations in each cycle. However, wide-issue processors also require a commensurate increase in the number of register read and write ports. Increasing the number of write ports is especially difficult and expensive. A possible solution places the responsibility on the compiler to ensure that there is no resource conflict on the write ports, even among operations with different latencies.

The example in Figure 4.2 demonstrates why conventional scheduling may not handle write port conflicts effectively. Here we assume a single-issue processor where the multiply
(0) cmpeq pr6 <- r1, r2
(1) mul r3 <- r3 * r5
(2) add r2 <- r5 + 1
(3) load r5 <- (r4)
(4) bfalse pr6, Loop

Figure 4.2: Backtracking schedulers can handle resource conflicts

and add operations share the same write port, and their latencies are two and one, respectively. A conventional scheduler first schedules the multiply and then cannot schedule the add in the next cycle due to the resource conflict on the write port. None of the other operations can be issued in cycle 1. Thus, the schedule length is 8. A backtracking scheduler also schedules the multiply first. Since the delay in scheduling the add in cycle 3 extends the entire schedule, the add displaces the multiply. The load operation uses distinct ports and can be issued in cycle 3. Finally, as in the example in Figure 4.1, the branch displaces other scheduled operations and is eventually issued in cycle 3. The schedule length is reduced by three cycles, two due to filling branch delay slots and one due to filling a write port conflict slot.

4.2 Main concepts

This section introduces the basic elements behind the backtracking approach to instruction scheduling. We begin by presenting steps performed by any instruction scheduler and
showing differences when backtracking is employed.

A *ready* operation is an operation whose predecessors have all been scheduled. Conventional schedulers issue operations in dependence order, that is, at any scheduling step they consider only ready operations. A *cycle/instruction-based* scheduler fills the current cycle (i.e., instruction) with operations before moving on to the next cycle. At any cycle, it schedules the highest priority ready operation that has no resource conflicts with previously scheduled operations. An *operation-based* scheduler considers ready operations in priority order. It assigns the highest priority ready operation the earliest cycle such that its resource and dependence constraints are satisfied. We evaluate the proposed backtracking schedulers by comparing them to a cycle-based one, the Cycle scheduler. Details about its implementation are given in a technical report [1].

Conventional schedulers always issue operations in dependence order which restricts their ability to produce high quality schedules. For instance, such a dependence order strategy unnecessarily schedules a branch after all of its predecessors, and thus it often leaves the branch delay slots unfilled as shown earlier in Section 4.1. Scheduling by priority can be beneficial since some operations are more important than others and, intuitively, the important ones should be scheduled earlier. However, for cycle- and operation-based schedulers, a high priority operation is not scheduled if it is not ready.

Backtracking schedulers lift this restriction: a high priority operation can be scheduled even if some of its predecessors have not been scheduled. We attempt to schedule the highest priority operation so that both resource constraints and dependence constraints with respect to previously scheduled operations are satisfied. When no cycles are available
to schedule a given operation, one or more operations are unscheduled to make room for
the new operation. Such backtracking schedulers are the focus of this chapter.

Our backtracking schedulers and the Cycle scheduler begin by computing the static
early (\(E_i\)) and late (\(L_i\)) times as well as a priority for each operation \(i\). The static early
and late times assume a target processor with infinite resources and were formally defined
in Section 2.4. Let \(E_{\text{max}}\) be the maximum static early time among all operations in the
superblock, and the height of an operation \(i\) with respect to a branch \(b\), \(h_i(b)\), equal \(E_{\text{max}} -
L_i(b)\).

We set the priority of an operation equal to the sum over all superblock branches, \(b\),
of the product of the profiled weight of \(b\) and the height of the operation with respect to
\(b\), i.e. equal to \(\sum_{\text{branches}} w_{b} h_{i}(b)\) [36]. For superblocks, the weighted height priority is
a natural extension of the critical path priority, which has been successfully used in basic
block scheduling [4].

The schedulers considered in this chapter also use dynamic early (\(\text{EarlyTime}\)) and late
(\(\text{LateTime}\)) times of an operation. These times depend on the operation’s currently sched-
uled predecessors and successors, respectively, and were formally defined in Section 2.4.
Since the rest of this chapter only deals with the dynamic times we will refer to them just
as early/late times. An operation’s late time is infinite if no successors of the operation
are currently scheduled. Note that the early and late times of an operation \(i\) change as
scheduling progresses; the interval \(\left[i.\text{EarlyTime}, i.\text{LateTime}\right]\) thus represents the range of
cycles in which \(i\) may be scheduled without violating dependencies with currently scheduled
operations.
Figure 4.3: A backtracking scheduler may not terminate

Since backtracking schedulers do not always schedule operations in dependence order, it is possible that the interval \([EarlyTime, LateTime]\) is empty. Furthermore, even when a range of cycles is available, the operation may have resource conflicts that prevent it from being scheduled within this range. In this case we undo previous scheduling decisions. A backtracking scheduler is said to \textit{forcibly schedule} an operation \(i\) if it unschedules currently scheduled operations that have resource or dependence conflicts with \(i\) (conflicting operations), and then schedules \(i\). A scheduled operation is \textit{unscheduled} by removing its association with a particular schedule cycle, releasing resources it has reserved, and putting it back in the set of operations to be scheduled. A backtracking scheduler \textit{normally schedules} an operation \(i\) if it schedules \(i\) at a cycle in \([i.EarlyTime, i.LateTime]\) interval without needing to unschedule other conflicting operation(s).

There are many natural approaches for forcible scheduling. We first consider a forcibly scheduling policy in which each operation can only be scheduled in its \([EarlyTime, LateTime]\) range and in which any operation can unschedule any other operation.

However, this very general policy can lead to an infinite loop as shown in Figure 4.3. We schedule the five unit-latency operations on a two-issue uniform processor. If the operations are scheduled in priority order \((a, b, c, d, e)\) then after issuing operations \(a\) and \(b\) in cycles
0 and 2, respectively, the scheduler enters an infinite loop involving the remaining three operations. It repeatedly fails to schedule operations $c$, $d$, and $e$ in the $[1, 1]$ interval, i.e. on two issue slots.

To eliminate this problem, we need to extend the range of possible slots for scheduling an operation. In order to make forward progress, we allow an operation to be forcibly scheduled in any cycle after its early time, i.e. the operation's extended range includes cycles after its late time. It follows that an operation can only unschedule its successors and unrelated operations in the dependence graph. Even with this restriction, the design space of backtracking schedulers is large.

A backtracking scheduler has to decide where in the extended range of issue slots to schedule an operation. If the normal scheduling fails, the scheduler then has to select a conflicting operation to be unscheduled. A conflicting operation may have a higher or lower priority than the current operation, and it may be a successor or an unrelated operation. Since the set of conflicting operations usually includes more than one operation, there are two additional options of where to forcibly schedule an operation: at the earliest cycle such that only lower priority operations are unscheduled, or at the earliest cycle that contains at least one lower priority conflicting operation.

A well designed backtracking algorithm limits the number of backtracking steps to avoid approaching the computation time of an exhaustive search. It is also desirable that the algorithm be guaranteed to terminate after a finite number of steps. Finding the right tradeoff between the solution quality and the computation time is the major challenge in designing backtracking algorithms.
We thoroughly explore the backtracking space through three practical forcible scheduling policies. First, to guarantee the scheduler's termination we allow an operation to unschedule only its successors; one such policy is discussed in Section 4.3. Because of its relative simplicity, we are able to formally demonstrate that the proposed backtracking algorithm always completes.

For better performance, it is natural to allow an operation to unschedule not only its successors but also lower priority unrelated operations. An aggressive scheduler of this kind is described in Section 4.4. However, such a policy may cause excessive backtracking. Thus, to reduce the compile time we allow only certain operations to unschedule other operations. Such a selective backtracking scheduler is studied in Section 4.5. These last two schedulers avoid repeatedly scheduling and unscheduling an operation in the same cycle by remembering the cycle where the operation was forcibly scheduled; if they forcibly schedule the operation again, it is scheduled one cycle later.

### 4.3 Basic backtracking algorithm

This section presents the structure of a generic backtracking algorithm, BasicBT. We also show that the algorithm terminates.

The main loop of the algorithm, is shown in Figure 4.4. Each iteration begins by picking the highest priority unscheduled operation $Op$ (line 2). We compute the $EarlyTime$ and $LateTime$ of $Op$ and then attempt to normally schedule the operation in each cycle from $EarlyTime$ to $LateTime$ (line 7). If $Op$ is not scheduled after iterating through the cycles up to and including $LateTime$, we forcibly schedule it in the cycle determined by the procedure
(1) while (the list of unscheduled operations is not empty)
(2) \( Op = \text{HighestPriorityOperation}() \) \( // Op \) may not be ready
(3) Compute \( Op, \text{EarlyTime} \) and \( Op, \text{LateTime} \)
(4) \( \text{success} = \text{false} \)
(5) for (cycle from \( Op, \text{EarlyTime} \) through \( Op, \text{LateTime} \))
(6) \( \text{if} (\text{resources required by} \ Op \ \text{available}) \)
(7) \( \text{Schedule}(Op, \text{cycle}) \) \( // \) no unscheduling
(8) \( \text{success} = \text{true} \)
(9) \( \text{break} \)
(10) endfor
(11) if (\( \text{success} \neq \text{true} \) \( // Op \) may unschedule a higher priority operation
(12) \( \text{ForciblySchedule}(Op, \text{ForceCycle}(Op)) \)
(13) endwhile

Figure 4.4: Main loop of the basic backtracking scheduling algorithm

\text{ForceCycle}(Op) \) (line 12). In the basic backtracking algorithm, \text{ForceCycle}(Op) \) returns the earliest cycle in which \( Op \) can be scheduled by unscheduling only conflicting successors of \( Op \) (no predecessors of \( Op \) or unrelated operations are unscheduled).

The forcible scheduling mechanism ensures that once we select an operation, we are always able to schedule it, even if unscheduling other operations is necessary. The following theorem demonstrates that the basic backtracking algorithm terminates.

\textbf{Theorem 3} The basic backtracking scheduling algorithm terminates.

\textit{Proof.} Consider the while-loop iteration \( k \) of the backtracking algorithm when we first attempt to schedule an arbitrary operation \( i \). It follows that \( i \), for this iteration, is the highest priority unscheduled operation. Let \( j \) be the next highest priority unscheduled operation, if it exists, and let \( S \) denote the set of operations that are currently scheduled. Note that the highest priority operation in each iteration is always scheduled. Set \( S \) therefore includes all the operations in the dependence graph with higher priority than \( i \). We will
show that after a finite number of iterations, which may include unscheduling and scheduling of operations in $S$, operation $i$ and all the operations in $S$ are scheduled. That is, after a finite number of iterations the number of scheduled operations increases by one.

There are two ways to schedule operation $i$ during iteration $k$. First, if there exists a cycle in interval $[i.EarlyTime, i.LateTime]$ for which all the resources required for the execution of $i$ are available, the algorithm schedules $i$ normally in the earliest such cycle. Note that if $i$ has no scheduled successor, its late time is infinite and we are always able to schedule it without unscheduling other operations.

If $i$ cannot be scheduled normally, then the scheduling of $i$ is restricted by one or more currently scheduled successors of $i$. Let $G(i)$ be the subgraph of the dependence graph induced by $i$ and the immediate and non-immediate successors of $i$ in $S$. We now consider the operations scheduled and unscheduled from iteration $k$ until the iteration that operation $j$ is scheduled for the first time. Since an operation can only unschedule its successors, during these iterations only operations in $G(i)$ are scheduled and unscheduled. The number of times an operation in $G(i)$ is unscheduled cannot be larger than the number of times that all its predecessors in $G(i)$ are scheduled. Since operation $i$ has no predecessors in $G(i)$, it is scheduled exactly once before $j$ is first scheduled. Therefore, there is a finite bound on the number of times any operation in $G(i)$ is scheduled before $j$ is scheduled for the first time. We have thus shown that the algorithm schedules operation $i$ and all the operations in $S$ after a finite number of iterations.

This argument can be reapplied to operation $j$, the operation with the next highest priority after $i$. The same argument applies if operation $i$ is the lowest priority operation,
i.e. if operation $j$ does not exist. Since the number of operations in the dependence graph is finite, the backtracking algorithm terminates. 

4.4 Full backtracking scheduler

The full-backtracking FullBT scheduler is an aggressive implementation of the basic backtracking algorithm. To improve schedule quality, we allow an operation to unschedule not only its successors but also unrelated operations. In addition to normal scheduling, an operation can unschedule unrelated lower priority operations within its $[\text{EarlyTime}, \text{LateTime}]$ interval, and unrelated higher priority operations after its late time. This allows the scheduler to more aggressively schedule important operations as early as possible. To avoid repeatedly scheduling and unscheduling an operation in the same cycle, the FullBT scheduler maintains the $\text{AttemptedCycle}$ for each operation, the last cycle where it was forcibly scheduled, and if we forcibly schedule the operation again, we do it in $\text{AttemptedCycle} + 1$.

We show the pseudocode of the FullBT scheduler in Figure 4.5. It differs from the basic backtracking algorithm in two places. First, the code in lines 10 through 13 allows $Op$ to be forcibly scheduled at a cycle in $[Op.\text{EarlyTime}, Op.\text{LateTime}]$ if $Op$ has a higher priority than a conflicting operation and if $\text{cycle} > Op.\text{AttemptedCycle}$. Second, in line 16 the procedure call $\text{ForceCycle}(Op)$ is instantiated by the expression $Op.\text{AttemptedCycle} + 1$. That is, if $Op$ was last forcibly scheduled at $\text{AttemptedCycle}$, then if $Op$ cannot be scheduled in its $[\text{EarlyTime}, \text{LateTime}]$ interval we forcibly schedule it at $\text{AttemptedCycle} + 1$, even if this requires unscheduling a higher priority operation. If an operation is forcibly scheduled for the first time, we schedule it at its early time.
while (the list of unscheduled operations is not empty)

\( Op = \text{HighestPriorityOperation}(\)  // Op may not be ready

Compute \( Op.\text{EarlyTime} \) and \( Op.\text{LateTime} \)

\( success = false \)

for (cycle from \( Op.\text{EarlyTime} \) through \( Op.\text{LateTime} \))

if (resources required by \( Op \) available)

Schedule(\( Op, \text{cycle} \))  // no unscheduling

\( success = true \)

break

elseif (\( \text{cycle} > Op.\text{AttemptedCycle} \) AND \( \text{HasHigherPriority}(Op, \text{cycle}) \))

ForciblySchedule(\( Op, \text{cycle} \))  // \( Op \) unschedules only lower priority operations

\( success = true \)

break

endfor

if (success \( \not= \) true)  // \( Op \) may unschedule a higher priority operation

ForciblySchedule(\( Op, Op.\text{AttemptedCycle} + 1 \))

endwhile

Figure 4.5: Main loop of the FullBT backtracking scheduler

Experimental results indicate that the FullBT scheduler is very effective in filling the delay slots of branches. However, the number of unscheduling steps can be excessive as shown in Figure 4.6. We thus develop a modified backtracking scheduler that reduces the compilation time.

### 4.5 Selective backtracking scheduler

The SelectBT backtracking scheduler selectively enables backtracking to control the amount of unscheduling, while still maintaining the quality of the overall schedule. Given the objective of filling branch delay slots, only operations with negative incoming latencies are allowed to unschedule other operations. For our machine models, branches have negative incoming edge latencies, and therefore branches can be forcibly scheduled. On the other
Figure 4.6: Operation of the FullBT scheduler
hand, if our objective were to handle write port resource conflicts between certain high-latency and low-latency operations, we would allow backtracking for selected low-latency operations.

The SelectBT scheduler enables backtracking for branches as well as for operations that have been previously unscheduled. In addition, as in the conventional Cycle scheduler, but unlike FullBT, the SelectBT scheduler further reduces the number of unscheduling steps by only scheduling ready operations.

The main loop of the SelectBT scheduler is shown in Figure 4.7. The SelectBT scheduler picks the highest priority ready operation $Op$ (line 2) and attempts to schedule it in each cycle from its early to late time. $Op$ is scheduled if resources are available (line 7), or if backtracking is enabled for $Op$ and all conflicting operations have lower priorities (line 11). As in FullBT, if $Op$ is not scheduled after iterating through the cycles up to and including its late time, we forcibly schedule it in $Op.AttemptedCycle + 1$ (line 17).

At the beginning, backtracking is enabled for branch operations only. However, if an operation is unscheduled by a branch, the operation may have no valid cycles for scheduling within its $[EarlyTime, LateTime]$ range. We prevent deadlock by enabling backtracking for any unscheduled operation (lines 12 and 18).

In Figure 4.8 we show the scheduling steps of the SelectBT scheduler when applied to the example in Figure 4.1. In this case SelectBT finds an optimal schedule. Note that SelectBT schedules the first three operations without backtracking. Since $bfalse$ has higher priority than the $add$ operation, in step 4 it is forcibly scheduled at its early time of 1. The $add$ operation is unscheduled and returned to the $ReadyList$, and backtracking is enabled for
while (ReadyList is not empty)
  $Op = \text{HighestPriorityOperation}(\text{ReadyList})$

  Compute $Op.\text{EarlyTime}$ and $Op.\text{LateTime}$

  $success = \text{false}$

  for (cycle from $Op.\text{EarlyTime}$ through $Op.\text{LateTime}$)
    if (resources required by $Op$ available)
      Schedule($Op$, cycle)       // no unscheduling
      $success = \text{true}$
      break
    elseif (backtracking enabled for $Op$ AND cycle > $Op.\text{AttemptedCycle}$ AND HasHigherPriority($Op$, cycle))
      ForciblySchedule($Op$, cycle)  // $Op$ unschedules only lower priority operations
      Enable backtracking for unscheduled operations
      $success = \text{true}$
      break
  endfor

  if ($success \neq \text{true}$) // $Op$ may unschedule a higher priority operation
    ForciblySchedule($Op$, $Op.\text{AttemptedCycle} + 1$)
    Enable backtracking for unscheduled operations
  endwhile

Figure 4.7: Main loop of the SelectBT backtracking scheduler

<table>
<thead>
<tr>
<th>cmp.ep pr4 = r2, 0</th>
<th>Ops</th>
<th>E</th>
<th>L</th>
<th>Pr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cmp</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>add r3 = r1 + r2</td>
<td>add</td>
<td>0</td>
<td>2</td>
<td>−1</td>
</tr>
<tr>
<td>load r2 = (r3)</td>
<td>ld</td>
<td>1</td>
<td>3</td>
<td>−2</td>
</tr>
<tr>
<td>−3 = −2</td>
<td>bf</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4.8: Operation of the SelectBT scheduler

0: cmp cmp cmp cmp cmp cmp
1: add add bf bf bf
2: ld ld add add
3: ld

Unsched.ops - - - add ld -
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines of code</th>
<th>Static operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td>29K</td>
<td>290K</td>
<td>game of go</td>
</tr>
<tr>
<td>m88ksim</td>
<td>18K</td>
<td>69K</td>
<td>processor simulator</td>
</tr>
<tr>
<td>compress</td>
<td>1K</td>
<td>9K</td>
<td>file compression</td>
</tr>
<tr>
<td>li</td>
<td>7K</td>
<td>27K</td>
<td>Lisp interpreter</td>
</tr>
<tr>
<td>ijpeg</td>
<td>28K</td>
<td>99K</td>
<td>image compression</td>
</tr>
<tr>
<td>perl</td>
<td>23K</td>
<td>161K</td>
<td>Perl interpreter</td>
</tr>
<tr>
<td>vortex</td>
<td>52K</td>
<td>275K</td>
<td>database</td>
</tr>
</tbody>
</table>

Table 4.1: Benchmark set

it. In step 5 the add then unschedules the lower priority load operation which is finally scheduled in cycle 3.

4.6 Experimental evaluation

We implemented our scheduling algorithms within the Trimaran compilation system, version 2.00 [96] running on Linux. The system runs on a 4x450MHz Intel Pentium II Xeon with 2GB of RAM. We use a variety of measures, including number of empty branch delay slots, optimally scheduled branches and regions, dynamic cycles, to show that the opportunity exists for backtracking schedulers to produce significantly better schedules. We also measure the runtime of each scheduler to show that backtracking schedulers can indeed be used in production compilers.

We apply the schedulers to seven benchmarks from the SPECint95 suite whose characteristics are given in Table 4.1. The gcc benchmark is not successfully compiled by the front end of the Trimaran compiler and is not included in our experiments. Our thesis targets general-purpose integer applications that are well represented by these benchmarks.
We evaluate the schedulers on two VLIW/EPIC processors: 1111 and 2111. Both processors have one branch unit that is used by the branch operations in all cycles in their execution. In addition, the branches use an integer functional unit during the first cycle. The branch latency is varied from 1 through 3 and the concise notation for a particular processor design encodes the branch latency as a suffix, e.g. 211L3 denotes a 2111 processor with a branch latency of three. The latencies of all other operations are fixed as follows: integer ALU 1, float ALU 3, int/float multiply 3, int/float divide 8, load 2, and store 1. We thus work with six processor configurations: 111L1, 211L1, 111L2, 211L2, 111L3, and 211L3, whose characteristics make them representative of a wider class of processors.

4.6.1 Branch delay slots and optimally scheduled branches and regions

A branch has an empty delay slot if there is an empty instruction (instruction that contains no operations) during the cycles of the branch execution. A branch delay slot is filled if it is not empty. We count branch delay slots statically after all scheduling phases have completed. We show in Figure 4.9 the percentage of branch delay slots that are filled by each scheduler and the percentage improvement over Cycle. Over all processors and benchmarks the Cycle scheduler leaves 24.2% of all delay slots empty. Increasing the branch latency from 2 to 3 causes the fraction of empty delay slots to increase from an average of 7.9% to 37.6%, that is, more than a fourfold increase. These results indicate that as the branch latency increases, it may be possible to to significantly improve schedules generated by the conventional scheduler. With a branch latency of 3, the FullBT scheduler fills 69.2% of all delay slots, reducing the number of unfilled slots from 37.6% to 30.8% of all delay slots.
The SelectBT scheduler performs nearly as well, filling 68.6% of all delay slots.

The number of filled delay slots is a measure of how well the scheduler is exploiting the processor's available parallelism. A more direct measure of schedule quality is the number of optimally scheduled branches and superblocks. We compute early time bounds using the tight, dependence and resource-based lower bound described in Section 5.4.2. A branch is scheduled optimally if its issue time equals its early time bound. A superblock is scheduled optimally if every branch within it is scheduled optimally.

We show in Figure 4.10 the percent of branches that are optimally scheduled, and in Figure 4.11 the percent of superblocks that are optimally scheduled. For non-unit branch latency the FullBT scheduler increases the percentage of superblock scheduled optimally over the Cycle scheduler from an average of 66.9% to 81.4%, an increase of 21.7%. For non-unit branch latency the SelectBT scheduler schedules optimally 80% of superblocks, an increase of 19.6% over Cycle. Note that the actual fraction of superblock scheduled
Figure 4.10: Percent of branches optimally scheduled

Figure 4.11: Percent of superblocks optimally scheduled
<table>
<thead>
<tr>
<th>Processor</th>
<th>BasicBT</th>
<th>SelectBT</th>
<th>FullBT</th>
<th>Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg</td>
<td>max</td>
<td>avg</td>
<td>max</td>
</tr>
<tr>
<td>111L1</td>
<td>0.0</td>
<td>0.2</td>
<td>-0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>211L1</td>
<td>0.6</td>
<td>1.7</td>
<td>-0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>111L2</td>
<td>0.4</td>
<td>0.7</td>
<td>1.1</td>
<td>1.7</td>
</tr>
<tr>
<td>211L2</td>
<td>1.6</td>
<td>3.1</td>
<td>1.3</td>
<td>3.2</td>
</tr>
<tr>
<td>111L3</td>
<td>0.6</td>
<td>1.1</td>
<td>1.6</td>
<td>2.5</td>
</tr>
<tr>
<td>211L3</td>
<td>2.7</td>
<td>7.3</td>
<td>2.5</td>
<td>7.4</td>
</tr>
</tbody>
</table>

Table 4.2: Percent speedup over the Cycle scheduler

optimally may even be higher since the bounds may differ from the optimal values.

## 4.6.2 Dynamic cycles

We compute a tight lower bound on the number of dynamic cycles consumed by any benchmark by taking the sum over all branches of the branch’s early time bound plus its latency, times its weight (number of times the branch is taken during profiling). The gap between this bound and the number of dynamic cycles in a schedule produced by the Cycle scheduler represents the size of the potential improvement. We show in Table 4.2 that the size of this gap increases from 3.4% to 4.8% as the branch latency increases from 1 to 3. The table also gives the maximum values over the six benchmarks. For the branch latency of 2 the backtracking schedulers produce schedules that are on average 1.1% better in terms of dynamic cycles than schedules produce by the conventional scheduler, and for branch latency of 3 the speedup increases to 1.9%.

Given the gap between the bound and schedules produced by the Cycle scheduler, we measure the fraction of this performance gap that is eliminated by the backtracking schedulers. This data is shown in Figure 4.12 for the 111L3 processor, and Figure 4.13 for the
211L3 processor. For a branch latency of 3 the SelectBT scheduler reduces the performance gap by 42.8%; i.e. more than 42.8% of the performance lost by the conventional scheduler has been regained. Similarly, the FullBT scheduler regains more than 42.1% of the performance lost by the Cycle scheduler, again with a branch latency of 3.

From the graphs we can see that both backtracking schedulers eliminate more of the performance gap when the processor is wider. On a wider processor it is easier for the backtracking schedulers to move operations and fill delay slots. This added flexibility accounts for the performance improvement. We also note that FullBT outperforms SelectBT for the 111L3 processor, while the opposite is true for 211L3.

4.6.3 Runtime and scheduling steps

As described above, backtracking schedulers schedule operations normally and forcibly. We estimate the amount of work done by each scheduler by counting the total number of times operations are scheduled. This data is shown in Table 4.3. Note that for unit branch latency the number of scheduling steps is exactly the same for the three schedulers; the weighted priority function used guarantees that no operation needs to be forcibly scheduled. For the narrow 111 processor with non-unit branch latency, the FullBT scheduler uses 50% more scheduling steps than the Cycle scheduler, while for the wider 211 processor this number decreases to 9% because of the greater availability of scheduling slots. For non-unit branch latency, on average the SelectBT scheduler uses about 20% fewer scheduling steps than FullBT.

The number of forcible scheduling steps measure the extra, backtracking work. As noted
Figure 4.12: Percent of Cycle gap eliminated by backtracking schedulers for 111L3

Figure 4.13: Percent of Cycle gap eliminated by backtracking schedulers for 211L3
Table 4.3: Ratio of scheduling steps of backtracking schedulers over Cycle

<table>
<thead>
<tr>
<th>Processor</th>
<th>BasicBT</th>
<th></th>
<th></th>
<th>SelectBT</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>FullBT</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg</td>
<td>max</td>
<td>avg</td>
<td>max</td>
<td>avg</td>
<td>max</td>
<td>avg</td>
<td>max</td>
<td>avg</td>
<td>max</td>
<td>avg</td>
<td>max</td>
</tr>
<tr>
<td>111L1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>211L1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111L2</td>
<td>1.01</td>
<td>1.03</td>
<td>1.05</td>
<td>1.09</td>
<td>1.56</td>
<td>2.90</td>
<td>1.42</td>
<td>1.42</td>
<td>1.42</td>
<td>1.42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>211L2</td>
<td>1.01</td>
<td>1.02</td>
<td>1.01</td>
<td>1.03</td>
<td>1.11</td>
<td>1.42</td>
<td>1.43</td>
<td>2.42</td>
<td>1.43</td>
<td>2.42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111L3</td>
<td>1.01</td>
<td>1.03</td>
<td>1.07</td>
<td>1.14</td>
<td>1.43</td>
<td>2.42</td>
<td>1.43</td>
<td>2.42</td>
<td>1.43</td>
<td>2.42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>211L3</td>
<td>1.01</td>
<td>1.02</td>
<td>1.02</td>
<td>1.02</td>
<td>1.07</td>
<td>1.23</td>
<td>1.07</td>
<td>1.23</td>
<td>1.07</td>
<td>1.23</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

earlier there is no backtracking for unit branch latency. On average the SelectBT scheduler uses five times fewer forcible scheduling steps than the FullBT scheduler for a branch latency of 2, and about three times fewer for a latency of 3.

We also measure the runtime of the three schedulers by calls to Trimaran timing routines that use the clock() library function. The data shown in Table 4.4 gives the runtime of the superblock scheduling routines only. As expected, the FullBT scheduler is the most time consuming, taking up to 26% longer than the conventional scheduler. However, the SelectBT scheduler outperforms FullBT and even Cycle. The Cycle scheduler operates cycle by cycle, and repeatedly attempts to schedule high priority operations even if resource constraints force them to be scheduled later. Runtime results indicate that SelectBT and even FullBT are fast enough to be used in production compilers.

4.7 Related work

The early work on parallel instruction scheduling for processors was carried out in the context of microcode compaction [26, 80]. Heuristics for scheduling instructions in a basic block...
for pipelined processors [41, 49] and superscalar processors [99] typically rely on ordering the operations in a block based on priority functions and scheduling the operations one by one as in the Cycle scheduler. Approaches for basic block scheduling using optimization techniques such as finite state automata have been studied [10].

The amount of parallelism within a basic block is limited and not sufficient for modern EPIC processors. Global schedulers use a larger scheduling region and perform code motion between basic blocks. The trace scheduler constructs a trace consisting of a linear chain of basic blocks with multiple entries and exits [35, 68]. Global schedulers restrict the scheduling regions to reduce the complexity of inserting compensation code in the side entries/exports due to code motion [16, 34, 73]. Wavefront scheduling uses a region consisting of a tree of basic blocks [17]. Meld scheduling extends the scope of the optimization without increasing the complexity associated with code motion by allowing latency constraints to propagate outside the scheduling region [2].

Schedulers may incorporate various levels of backtracking to improve the quality of schedules. For example, the Multiflow trace scheduling compiler [68] employs a limited form of backtracking, for memory operations only. The need to backtrack arises because
memory operations can be replicated in the dependence DAG and scheduled multiple times. For example, due to register pressure a scheduled load is replicated to reclaim its target register from a higher priority operation. If none of the readers of the original load have been scheduled, then the original load can be safely removed from the schedule. The Multiflow scheduler does not refill the issue slot of the unscheduled operation and guarantees that an operation is unscheduled at most once. Wang et al. [98] apply backtracking to schedule precedence-constrained tasks on parallel processors. Their algorithm uses a single level of backtracking. It allows immediate predecessor rescheduling, in certain circumstances, to find better schedules with only a small increase in processing times. In contrast, our SelectBT and especially FullBT schedulers apply unrestricted, aggressive backtracking policies.

Special scheduling techniques have been developed for loops since programs spend a significant fraction of the total execution time in these regions [22, 27, 82]. Rau proposed the iterative modulo scheduler that backtracks in a manner similar to the SelectBT scheduler [79]. A modulo scheduler attempts to schedule for a particular initiation interval (II) and typically increases the II if it is unable to generate a schedule within the prescribed compile time budget. All schedules that meet a particular II are essentially equivalent. These unique characteristics of modulo scheduling give rise to a different set of algorithmic choices. First, the iterative modulo scheduler may occasionally get locked in a repetitive orbit. However, once the compile time budget is exhausted, the iterative modulo scheduler attempts to schedule using a different II. The FullBT and SelectBT schedulers are guaranteed to never revisit the same partial schedule. This guarantee is essential because there is no option similar to increasing the II and trying to schedule again. Second, the iterative
modulo scheduler must account for a cyclic dependence graph, which may constrain the operation late time even when no successor from the same loop iteration has been scheduled. Our schedulers do not face such constraints. Third, the schedule length is the primary optimization metric in acyclic scheduling as opposed to being a secondary metric in modulo scheduling.

Hoogerbrugge and Augusteijn [52] describe a simple backtracking strategy to fill branch delay slots when scheduling for the Philips Trimedia processor. Their scheduler ignores the control dependencies from non-branch operations to a branch b and schedule b as early as possible. If it turns out that b was scheduled too early, the scheduler unschedules the branch as well as the operations scheduled after it and attempts to schedule b one cycle later. To reduce the amount of backtracking, a lower bound on the cycle where a branch can be legally scheduled is computed. Such bounds are the focus of the next chapter.
Chapter 5

Resource-aware bounds

Most realistic problems in scheduling are very difficult; even their simplest versions when formulated as optimization problems belong to the class of NP-hard problems. In general, there are three approaches to deal with such intractable problems [18, 40], and all of them use bounds in significant ways.

As a first approach, one tries to develop and evaluate “natural” algorithms for obtaining a good, not necessarily optimal, solution. These types of algorithms are called heuristics. Having a tight bound can help evaluate the proposed heuristic and give some insight into the problem structure [3, 4].

In another approach, one attempts to exhaustively search the problem solution space. Since the space is prohibitively large, the goal here is to prune it, that is to exclude from the search procedure as much space as possible. Algorithms of this type are called branch-and-bound algorithms, and as the name suggests, bounds are a required part [15]. Furthermore, the tightness of bounds, both lower and upper, can substantially speed up the search.
For the resource-constrained scheduling problem in high-level synthesis, Narasimhan and Ramamujam show that a branch-and-bound technique which computes both upper and lower bounds is several orders of magnitude faster than a procedure that uses the conventional LP-bounds [76].

In a third approach, one tries to establish a performance guarantee for an algorithm. The goal is to show that the solution given by the proposed algorithm is within a certain range of the optimal value. Such algorithms are called approximation algorithms. Since we do not know the optimal value, lower and upper bounds are heavily used in deriving approximation ratios [24, 42].

Overall, bounds are an invaluable tool for evaluating any type of algorithm. Tight bounds can often be incorporated in the algorithm under consideration itself. We believe that studying tight bounds enhances our understanding of a problem’s structure, and this knowledge is a key to creating high quality algorithms.

The standard way to derive a lower (upper) bound for a minimization (maximization) problem is to relax the original problem along one or more of its dimensions, and then optimally solve the relaxed problem. The relaxation is selected to be as close to the original problem as possible, while still being tractable. The optimal solution of the relaxed problem then clearly gives a bound on any solution of the original problem.

One commonly used relaxation disregards the integer constraints on binary variables in an integer linear formulation (ILP) of the problem as shown in Section 3.1.2. We thus obtain a linear program that can be efficiently (from a theoretical point of view) solved. This relaxation illustrates the tradeoffs in designing good bounds. While an LP-based re-
laxation can be easily designed with a little or no understanding of the problem structure, its calculation is time consuming in practice and not appropriate in many contexts. It is desirable that the bound computation be at least an order of magnitude faster than the algorithms for the problem itself. In a production compiler environment, this requirement clearly excludes LP-based bounds, and we do not consider them in this chapter. Nevertheless, they are a useful tool for deriving approximation algorithms and branch and bound procedures, and can be used off-line for evaluating the proposed heuristics [84].

Of course, some trivial bounds can be obtained from first principles, in an ad-hoc manner. They usually are not sharp enough, but serve as a fast initial estimation. In Section 5.2, we will illustrate both approaches for deriving bounds for basic block scheduling.

This chapter develops and evaluates resource-aware lower bounds for superblock and basic block scheduling. We analytically establish the relative tightness of known combinatorial bounds and propose a methodology for systematically deriving bounds for scheduling on the extended VLIW/EPIC processor as defined in Section 2.1. The case study on register file write ports shows that our new write port-aware bound can be considerably better than the tightest known bound.

The organization of this chapter is as follows. We begin with a review of related work on bounds in scheduling. We then evaluate fast, combinatorial lower bounds for superblock and basic block scheduling in Sections 5.2 and 5.3, respectively. Basic block bounds are used to derive tight practical bounds for superblocks. Section 5.4 describes our uniform approach to develop lower bounds that account for register file write ports constraints, complex resource usage of operations, and issue restrictions.
5.1 Scheduling bounds

In one of the first scheduling papers, Hu describes an algorithm for optimally assigning jobs under tree-like precedence constraints on identical processors, now known as the critical path algorithm [55]. Along with the algorithm, he also presents two bounds, on the schedule length and on the number of processors. In another seminal work on scheduling, Graham initiated the field of approximation algorithms through studying scheduling bounds [42, 43].

In the field of multiprocessor scheduling, Fernandez and Bussel [33] developed bounds on the number of parallel machines as well as on the schedule length. This work has been extended for scheduling with communication delays [39] and scheduling for high level synthesis [78, 83].

We will briefly review scheduling algorithms for high level (or behavioral) synthesis since it is closely related to our interest in dealing with structural and issue constraints in instruction scheduling for compilers. High-level synthesis (HLS) is the translation of an algorithmic-level specification of the behavior of a digital system into a register-level structure that implements that behavior [70, 71]. One of its most important phases is scheduling. It involves assigning each operation a starting time, or control step(s), such that dependence and resource constraints are satisfied. A control step is one cycle of the system clock. Given the number of instances of each resource, the resource-constrained scheduling (RCS) problem minimizes the number of control steps required to execute all operations, or equivalently the completion time of the last operation. Given the number of control steps, the time-constrained scheduling (TCS) problem minimizes the number of instances of each resource. Since instruction scheduling in compilers is closely related to
the RCS problem, we will not consider the TCS problem further. A number of researchers have evaluated algorithms and bounds for the RCS problem [64, 75, 78, 83].

Bounds have played an important role in understanding the basic block instruction scheduling problem and solving it in practical terms. As discussed in Section 2.2, various extensions of a basic block have been proposed, each usually paired with a set of scheduling heuristics. However, little is known about the empirical and worst-case behavior of these heuristics, and we believe that the lack of appropriate bounds is one reason for this unsatisfactory situation. Eichenberger and Meleis recently use bounds to estimate the quality of scheduling heuristics for superblocks as well as to guide schedulers for better performance [30]. In the area of acyclic instruction scheduling, Rau uses two types of bounds in the iterative modulo scheduling algorithm [79]. The minimum initiation interval between two consecutive loop iterations, a crucial parameter in the algorithm, is the larger of the resource-constrained and recurrence-constrained lower bounds.

Bounds on schedule length are useful for creating near-optimal instruction schedules. Schedulers use bounds to predict the effect of scheduling one operation on the minimum issue time of a later operation. In this way, the scheduler can decide which operations are more critical than others. The bounds commonly used for this purpose emphasize dependence constraints and tend not to consider detailed resource constraints. We now demonstrate that a scheduler based on detailed resource-aware bounds can outperform a conventional scheduler.

Consider the two-exit superblock in Figure 5.1, taken from [30]. All operations have unit latencies except operation 4 which has a latency of 2. The target processor is fully pipelined
and can issue two operations in each cycle. A simple resource bound indicates that branch 3 can be scheduled no earlier than cycle \([3/2] = 2\), and that at least one of operations 0, 1, or 2 must be scheduled in cycle 0 to avoid delaying this branch. Similarly, branch 9 can be scheduled no earlier than cycle \([9/2] = 5\) and that at least one of operations 0, 1, 2 or 4 must be scheduled in cycle 0 to avoid delaying this branch.

It appears that operation 4 does not need to be scheduled in cycle 0 because the length of the longest dependence chain passing through it is only 4 cycles long. However not scheduling operation 4 in cycle 0 causes branch 9 to be delayed by one cycle, as shown in Figure 5.1. Delaying operation 4 must delay branch 9 because operations 6, 7, and 8 cannot all be scheduled in the same cycle. Operation 4 has a late time of 0 with respect to branch 9 and must be scheduled at cycle 0 to give the optimal schedule shown in Figure 5.1.

5.2 Lower bounds for basic blocks

Precise bounds on schedule length are needed to convey detailed information about resource and dependence constraints to the scheduler. We now describe the tightest known combi-
natorial bounds for basic blocks. These recently developed RCS bounds have not been previously applied to instruction scheduling, and we will show both analytically and experimentally that they outperform all other known basic block bounds. The bounds are also particularly useful because they can be extended to handle a range of processor features.

For the purpose of this chapter, a basic block is formally defined as a directed acyclic graph (DAG) where each node represents one of the \( n \) operations in the basic block, and each arc represents one of the \( p \) data and control dependencies among operations. We let \( b \) denote the last operation in the basic block, which must be a branch.

Using the notation introduced in Section 2.3 for an operation's start and completion times, a dependence arc from operation \( i \) to operation \( j \) with latency \( l(i,j) \) in the DAG implies that \( S_i + l(i,j) \leq S_j \) in any valid schedule. In the basic block scheduling problem, we search for a valid schedule that minimizes the basic block completion time \( C_{\text{max}} = \max_{i=1,\ldots,n} C_i \), or equivalently the issue time \( S_b \) of the branch operation. Note that the basic block completion time \( C_{\text{max}} \) equals \( S_b \) plus the branch latency, which is an architectural constant.

Efficient solutions for the basic block scheduling problem exist for the special cases of tree-structured precedence constraints [55] or when scheduling on a uniform two-unit processor GP2 [25]. The general problem, which includes all practical instruction scheduling tasks, is NP-hard [97]; however, the availability of optimal algorithms for special cases of the problem has facilitated the development of a set of basic block lower bounds. Most known bounds are described and evaluated in this section.

For clarity of the presentation, we assume a target processor with \( m \) uniform functional
units, GPM, and unit latency operations. We show in Section 5.4 how the tightest bound can be extended to cover the basic EPIC/VLIW processor with special-purpose functional units, and non-unit latency operations.

The static early time $E_i$, introduced in Section 2.4, is clearly zero for any operation $i$ with no predecessors, and for all other operations $E_i = \max\{E_j\} + 1$ over all immediate predecessors $j$ of $i$. Similarly, $L_i = \min\{L_j\} - 1$ over all immediate successors $j$ of a non-branch operation $i$. For the branch $b$, $L_b = E_b$. We now describe two trivial basic block bounds that ignore either the resource or dependence constraints on scheduling as introduced in Section 2.3.

First, the early time $E_b$, is a bound on the issue time of the branch operation, and thus $E_b + 1$, known as the Critical Path (CP), represents a bound on the basic block completion time. Note that the CP bound does not consider any processor resources.

On the other hand, if we disregard the dependence constraints, a simple resource bound can be obtained by dividing the number of operations by the number of functional units, i.e., $\lceil n/m \rceil$. We used this bound in Section 5.1.

We next describe four basic block bounds that take into account both resource and dependence constraints. We derive these lower bounds by systematically relaxing the basic block scheduling problem along its dimensions. The first bound combines the two simple bounds from the previous section. The second and third bounds replace the precedence constraints with appropriate early and late times, and the last bound replaces the precedence constraints with in-tree constraints, so that in all three cases the corresponding relaxed problem can be optimally solved.
**Hu bound (Hu) [55]:** For any integer $k$, $0 \leq k \leq CP$, let $n(k)$ denote the number of operations with $L_i \leq k$. A lower bound on the completion time of these operations is $[n(k)/m]$. An additional time of at least $CP - k$ is needed to schedule the remaining operations, so a lower bound on the completion time of all operations is therefore $\max_{k=0}^{\omega CP} ([n(k)/m] + CP - k)$. Note that this expression, for $k = 0$ includes the CP bound, and for $k = CP$ includes the simple resource bound. The Hu bound can be computed in $O(n + p)$ time.

**Rim and Jain (RJ) bound [83]:** Tight lower bounds on the solution to the basic block scheduling problem can be computed by minimizing the maximum lateness of any operation in the relaxed problem in which the static early and late times replace the precedence constraints. Since $C_{\max} - CP \geq \max_{i=1\ldots n} \{C_i - L_i\}$, minimizing the maximum lateness of any operation in this relaxed problem and adding $CP$ gives a lower bound on $C_{\max}$ in the original problem.

The relaxed problem can be optimally solved by list scheduling, with the priority of operation $i$ set to $L_i$, such that $S_i \geq E_i$ [91]. That is, operations are considered in order of nondecreasing late times, and each operation $i$ is scheduled at the earliest time greater than or equal to its early time such that the resource constraints are satisfied. The lower bound is then equal to the maximum of $CP + (C_i - L_i)$ over all operations $i$. Note that the bound is not simply $C_{\max}$ for the relaxed problem. The RJ bound can be computed in $O(n^2)$ time.

**Langevin and Cerny (LC) bound [64]:** A better bound on the start time of an operation can be computed by recursively using the RJ bound to find tighter lower bounds on the
earliest completion time of all preceding operations. That is, the early times that were originally computed by finding longest paths in the precedence graph can themselves be computed using the RJ algorithm.

The lower bound $E_i$ on the start time $S_i$ of operation $i$ is tightened by applying the same technique to a subset of the original problem. We let $E'_i = 0$ for any operation $i$ with no predecessors, as before. For all other operations we let $E'_i$ equal one less than the RJ lower bound computed for the subproblem consisting of $i$ and all its predecessors (we subtract one to account for the unit latency). The value of $E'_i$ is computed for operation $i$ only after $E'$ has been computed for all predecessors of $i$. Since the LC relaxation is more constrained than the RJ relaxation, $LC \geq RJ$ for any instance of the problem. The LC bound can be computed in $O(n^3)$ time.

**Brucker, Garey and Johnson (Br) bound:** We obtain another relaxation of the basic block scheduling problem by deleting edges of the precedence graph so that the remaining edges form an in-tree. We again minimize the maximum lateness of any operation.

The relaxed tree scheduling problem can be also optimally solved by list scheduling [19]. The priority of operation $i$ is set to $L_i$ calculated from the original DAG precedence graph, but the schedule satisfies only the tree precedence constraints. While there are many ways to extract a tree from a DAG, in this thesis we only consider a tree formed by connecting each operation to a single successor with the smallest late time. Note that the late time for each operation is unchanged in this tree. The Br bound can be computed in $O(n + p)$ time [72].

We now establish the relative tightness of the basic block bounds described in this
section. We first show that the Hu and Br bounds are equal, and then demonstrate that
the Hu bound is less than or equal to the RJ bound for any instance of the basic block
scheduling problem.

Lemma 4 For any instance, $Hu = Br$.

Proof. We first show that $Br \geq Hu$. We apply the Br algorithm and find the corresponding
schedule. Let $k$ be some integer in $[1...CP]$ and let $j$ be the operation with the largest
completion time among all operations such that $L_i = k$. Clearly $C_j - L_j \geq C_i - L_i$ for all
operations $i$ such that $L_i = k$; so it suffices to show that $(C_j - L_j) + CP \geq \lfloor n(k)/m \rfloor + CP - k$.
This is true because $C_j \geq \lfloor n(k)/m \rfloor$ and $L_j = k$. The same argument applies for all values
of $k$, so $Br \geq Hu$.

We now show that $Br \leq Hu$. The Hu bound is the same for the original graph and
the Brucker tree (the tree extracted from the graph by the Br algorithm) because the late
times $L_i$ are unchanged and therefore the values of $n(k)$ are unchanged. The same is true
for the Br bound. But for the Brucker tree, which is an in-tree, the Hu lower bound is in
fact the optimal value for the problem [55]. Therefore $Br \leq Hu$. \hfill \square

Lemma 5 For any instance, $Hu \leq RJ$.

Proof. It suffices to show that $\lfloor n(k)/m \rfloor + CP - k \leq RJ$ for all integers $k \in [0...CP]$.
We let $RJ'$ equal the bound computed by the Rim and Jain algorithm when applied to
the following relaxed scheduling problem: delete all operations $i$ with $L_i > k$, set $L_i = k$
for the remaining operations, set $E_i = 0$ and leave $CP$ unchanged. Notice that this new
scheduling problem is a relaxation of the scheduling problem solved by the original Rim and Jain algorithm, so \( RJ' \leq RJ \). When applied to this new problem, the Rim and Jain algorithm schedules all the operations as early as possible with a maximum completion time equal to \( \lceil n(k)/m \rceil \). Since \( RJ' \) is determined by the maximum lateness plus the critical path, \( \lceil n(k)/m \rceil - k + CP = RJ' \leq RJ \) for all \( k \). 

Therefore we have established the following theorem.

**Theorem 6** For any instance, \( CP \leq Hu = Br \leq RJ \leq LC \).

We experimentally evaluate the lower bounds for basic blocks by constructing a set of 100 hard synthetic instances. The instances have an average of 1500 jobs each, and the number \( m \) of general-purpose functional units varies from 3 to 6. The corresponding processor configurations are referred to as GP3, GP4, GP5, and GP6. We compare all lower bounds to the upper bound computed by the critical path list scheduling algorithm [55]. For each value of \( m \) we experimentally select the edge density that maximizes the difference between the lower and upper bounds. Table 5.1 shows the average absolute difference between each lower bound and the upper bound, and the percentage of the instances for which the lower bound equals the upper bound.

The results indicate that relaxing precedence constraints into early times gives tight lower bounds when combined with an algorithm that minimizes maximum lateness. Even better bounds can be computed by recursively tightening early times: the LC lower bound is on average only 0.16 units from the upper bound and equals it in 86.5% of instances.
<table>
<thead>
<tr>
<th>Processor</th>
<th>$Hu, Br$</th>
<th>$RJ$</th>
<th>$LC$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP3</td>
<td>13.60 / 0%</td>
<td>9.18 / 1%</td>
<td>0.38 / 70%</td>
</tr>
<tr>
<td>GP4</td>
<td>8.29 / 11%</td>
<td>3.31 / 27%</td>
<td>0.20 / 81%</td>
</tr>
<tr>
<td>GP5</td>
<td>3.95 / 28%</td>
<td>0.58 / 77%</td>
<td>0.03 / 97%</td>
</tr>
<tr>
<td>GP6</td>
<td>2.50 / 44%</td>
<td>0.26 / 88%</td>
<td>0.02 / 98%</td>
</tr>
</tbody>
</table>

Table 5.1: Quality of lower bounds vs. an upper bound for basic blocks: average absolute difference and % instances equal

5.3 Lower bounds for superblocks

Since the basic block bounds described in the previous section compute a tight lower bound on the completion time of the last, branch operation in a basic block, they can easily be used to compute a lower bound on the overall weighted completion time of a superblock. We simply apply any basic block scheduling bound to every branch in a superblock. The weighted sum of these lower bounds then gives a lower bound on the overall cost of the superblock.

By Theorem 6 and their construction, the basic block-based bounds on the superblock completion time have the same relative order as the corresponding basic block bounds. That is, for any instance of the superblock scheduling problem, $CP \leq Hu = Br \leq RJ \leq LC$.

Eichenberger and Melcës experimentally show that the RJ and LC bounds outperform all other known bounds [30]. They apply the bounds to 6615 superblocks extracted from SPECint95 benchmarks on 3 processor configurations with special-purpose functional units (1111, 2121, and 3221) and 3 processor configurations with 1, 2, and 4 general-purpose functional units (GP1, GP2, and GP4). All types of functional units are fully pipelined. The performance of the bounds is given in Table 5.2 and is quantified by (1) the average
<table>
<thead>
<tr>
<th>Processor</th>
<th>CP</th>
<th>Hu</th>
<th>RJ</th>
<th>LC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg</td>
<td>max</td>
<td>avg</td>
<td>max</td>
</tr>
<tr>
<td>1111</td>
<td>27.55</td>
<td>93.15</td>
<td>0.59</td>
<td>27.44</td>
</tr>
<tr>
<td>2121</td>
<td>4.79</td>
<td>86.49</td>
<td>0.36</td>
<td>27.53</td>
</tr>
<tr>
<td>3221</td>
<td>1.55</td>
<td>80.00</td>
<td>0.64</td>
<td>29.65</td>
</tr>
<tr>
<td>GP1</td>
<td>44.55</td>
<td>93.51</td>
<td>0.09</td>
<td>17.39</td>
</tr>
<tr>
<td>GP2</td>
<td>13.89</td>
<td>87.18</td>
<td>0.31</td>
<td>20.00</td>
</tr>
<tr>
<td>GP4</td>
<td>1.24</td>
<td>75.00</td>
<td>0.08</td>
<td>15.67</td>
</tr>
</tbody>
</table>

Table 5.2: Quality of lower bounds vs. the tightest lower bound for superblocks

percentage difference between the indicated bound and the tightest lower bound, and (2) the maximum percentage difference between the indicated bound and the tightest lower bound. The results demonstrate that the CP bound is much weaker than the other bounds and that the use of resource constraints in Hu, RJ, and LC greatly improves the quality of the lower bounds. The results for RJ and LC indicate that these resource-aware bounds outperform the CP and Hu bounds.

In the remainder of this section we present a lower bound on the weighted completion time (WCT) of a schedule and show that this bound outperforms an LP-based bound. Until recently, all non-trivial lower bounds on WCT were based on a linear programming relaxation since most of the WCT problem relaxations along the natural dimensions are NP-hard. One exception is the WCT problem with early times but without precedence constraints for which Baptiste recently discovered a polynomial-time algorithm [14].

We outline the two key observations behind Baptiste's algorithm. First, the times at which operations issue and complete in the optimal schedule belong to the set of early times and their multiples (up to a factor of $n$). Next, a resource profile $\xi$ is defined as a vector $(\xi_1, \xi_2, \ldots, \xi_m)$ such that $\xi_1 \leq \xi_2 \leq \cdots \leq \xi_m$ and $\xi_m - \xi_1 \leq 1$. Each entry $\xi_i$ in the
resource profile divides the available execution times for processor \( i \). Given two resource profiles \( \xi \) and \( \xi' \), \( \xi \ll \xi' \) denotes that for any index \( i \) in \( \{1, \ldots, m\} \), \( \xi_i \leq \xi'_i \). Baptiste proves that an optimal partial schedule for the first \( k \) operations between any two resource profiles \( \xi \ll \xi' \) can be found recursively from optimal partial schedules for the first \( k - 1 \) operations between all resource profiles \( \theta \) and \( \theta' \) such that \( \xi \ll \theta \ll \theta' \ll \xi' \). While this gives a dynamic programming algorithm whose time and space complexities are respectively \( O(n^{3m+4}) \) and \( O(n^{2m+2}) \), we show below that the actual runtime and memory usage are much lower.

**Dynamic Programming (DP) bound:** We first apply the LC algorithm to find tight early times for the operations. We then use a version of Baptiste’s algorithm where the times at which operations start and end in the optimal schedule belong to \( \{0, 1, \ldots, n\} \).

We compare the DP bound on weighted completion time to an LP-based lower bound. The time-indexed LP formulation from Section 3.1.2 is used. We evaluate the DP bound against the LP bound on a processor with two general-purpose functional units, GP2. We use a set of 100 instances each with 90 operations, and the weight of each operation is uniformly distributed in \([1, 10]\). Here, we also vary the edge density, expressed as the number of edges over the maximum possible number of edges in the DAG. The results in Table 5.3 show that the DP bound clearly outperforms the LP bound across a variety of edge densities. While the DP algorithm has an unappealing asymptotic time and space complexity, efficient implementations are possible for small values of \( m \).
<table>
<thead>
<tr>
<th>Edge density</th>
<th>LP</th>
<th>DP</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>10.261</td>
<td>11.187</td>
<td>9.0%</td>
</tr>
<tr>
<td>30%</td>
<td>11.010</td>
<td>11.285</td>
<td>2.5%</td>
</tr>
<tr>
<td>50%</td>
<td>13.006</td>
<td>13.345</td>
<td>2.6%</td>
</tr>
</tbody>
</table>

Table 5.3: DP vs. LP lower bounds for WCT on GP2 processor

5.4 Extending bounds to account for resource constraints

Operation issue restrictions, complex resource usage, and register write port conflicts represent structural/resource hazards spanning all pipeline stages in execution of an operation. Such hazards need to be considered as resource constraints in the extended VLIW/EPIC processor model, and then exposed to the scheduler to generate high quality schedules. In this section, we show that the tightest basic block and superblock bounds can be further improved by taking into account the three types of resource constraints discussed above. A unique advantage of our method is its generality: the same procedure can be applied to a wide variety of architectural and microarchitectural processor features.

We begin by briefly describing how to deal with functional units of multiple types and with functional units that are not fully pipelined. Since most functional units in modern processors are fully pipelined, operations with non-unit latencies can often be treated as if they have unit latency by the bounds. If a non-unit operation with a latency of \( l \) executes on a non-fully pipelined functional unit, for the purpose of the bound calculation we replace it with a chain of \( l \) unit latency operations in the dependence graph. The RJ and LC algorithms are then applied on the modified graph to obtain valid lower bounds \[83\]. When multiple types of operations and functional units (or other resources) are present, we
compute a lower bound with respect to each type and select the tightest lower bound [83].

5.4.1 Register file write ports

Exploiting the available parallelism in a program requires a proportional increase of the number of hardware resources. One resource that does not scale well is the set of the register file write ports. For example, the Itanium processor [58] has an 128-entry integer register file with eight read and six write ports. While the number of ports is sufficient for most instruction sequences, certain combinations should be avoided by the scheduler. A postincrement load updates the operation’s address register and requires two register writes during its second cycle. Thus, issuing two postincrement loads in some cycle \( c \) followed by four integer operations in cycle \( c + 1 \) requires a total of eight write ports in cycle \( c + 1 \) to avoid a stall.

The clustered approach alleviates the problem by partitioning register files so that each cluster of functional units accesses its own set of register files [21, 32]. For a centralized or clustered register file organization, exposing the number of register write ports to the scheduler can reduce stalls caused by port oversubscription. The compiler becomes responsible for scheduling operations so that there is no resource conflict on the write ports.

A bound on the schedule length that is aware of restrictions on the number of register file write ports can be determined as follows. We first create a set of write port resource types, one type for each register file in the processor. Each write port type has a certain number of resources (write ports). For example, we model the Itanium’s integer register file that has six write ports as an integer write port type with six resources. Similarly, the
Itanium's predicate register file with 11 write ports is modeled as a binary write port type with 11 resources.

Second, for any operation \( i \) that writes to a register, we create a new node \( i_w \) of the appropriate type. For example, the new node for an `add` or an integer `load` has an integer write port type, while the node for a `compare` operation has a binary write port type. We add a dependence edge from \( i \) to \( i_w \) with a latency equal to one less than \( i \)'s latency. The edge \((i, i_w)\) models the fact that operation \( i \) uses a register file write port only during its last cycle. In addition, for any true data dependence edge from operation \( i \) to some other operation \( j \) we create an edge from \( i_w \) to \( j \) with a latency of 1, and for any control dependence edge from \( i \) to a branch \( b \) we create an edge \((i_w, b)\) with a latency equal to one minus the branch latency.

Finally, we apply the Langevin and Cerny (LC) algorithm to the modified DAG with respect to each resource type including the write port resource types. This procedure of modifying the dependence graph clearly produces a lower bound since the new nodes and edges are less restrictive than the actual usage of write ports. For example, if an operation \( i \) completes in some cycle \( c \), the edge \((i, i_w)\) requires that \( i \) uses a write port no earlier than cycle \( c \), but not in cycle \( c \) itself.

We illustrate this procedure on the dependence graph shown on the left side of Figure 5.2. The DAG consists of an integer `load` operation of latency 2 and three integer operations of latency 1; we label each operation with its type. All four operations write to integer registers. We schedule the DAG on a processor with an infinite number of functional units but with an integer register file that has only two write ports.
Figure 5.2: Deriving a bound that accounts for register file write ports

For each operation, we create a node of integer write port type (W) and add appropriate edges. The modified DAG is shown on the right side of Figure 5.2. The LC algorithm requires the static early and late times and they are also shown. For example, node W1 has early and late times of 1 because the corresponding load operation has latency of 2, i.e., cycle 1 is the earliest cycle the operation can write its destination register through a register file write port.

If we do not take into account the limited number of write ports, the LC algorithm gives a bound of 2 cycles. However, the bound based on operations of type W, 3 cycles, is tighter.

We implemented the new bound, denoted by WP, as well as the critical path (CP) and original LC bounds within the Trimaran compilation system [96]. We demonstrate the quality of the WP bound by using an experimental setup that is similar to the one described in Section 4.6. We apply the three bounds to the benchmarks shown in Table 4.1 on two processor configurations: 1111 with one integer register file write port (1111wp1), and 2111 with two integer write ports (1111wp2). Branches and integer ALU operations take 1 cycle, an integer load takes 2 cycles, an integer multiply takes 3 cycles, and an integer divide takes 8 cycles.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1111wp1</th>
<th>2111wp2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LC</td>
<td>WP</td>
</tr>
<tr>
<td>go</td>
<td>40.9</td>
<td>43.1</td>
</tr>
<tr>
<td>m88ksim</td>
<td>67.9</td>
<td>70.5</td>
</tr>
<tr>
<td>compress</td>
<td>126.6</td>
<td>126.8</td>
</tr>
<tr>
<td>li</td>
<td>83.8</td>
<td>84.2</td>
</tr>
<tr>
<td>jpeg</td>
<td>181.2</td>
<td>214.1</td>
</tr>
<tr>
<td>perl</td>
<td>87.0</td>
<td>87.7</td>
</tr>
<tr>
<td>vortex</td>
<td>79.7</td>
<td>80.7</td>
</tr>
<tr>
<td>gmean</td>
<td>86.8</td>
<td>90.4</td>
</tr>
</tbody>
</table>

Table 5.4: Quality of LC and WP lower bounds vs. critical path bound for superblocks

In Table 5.4 we show the percent increase of the LC and WP bounds over the critical path bound. Both LC and write port-aware bound significantly outperform CP. By construction, WP is tighter than LC. The largest performance difference between the two bounds is for the jpeg benchmark; WP is 18% and 17% better than LC on 1111wp1 and 2111wp2 processors, respectively. While nearly 1% percent of all operations are integer multiply and divide operations for jpeg, for the remaining six benchmarks this value is only about 0.1%. Thus, we expect that for code that contains a large number of integer operations of longer latencies and a number of write ports that are mainly designed for unit latency operations, WP should clearly outperform the original LC bound.

Our procedure for deriving a tight lower bound by extending the dependence graph with resource-type nodes and edges can be applied to most processor features of interest. The next section demonstrates the generality of our methodology: we develop a bound that is aware of the complex resource usage of branch operations.
5.4.2 Complex resource usage

The bounds considered so far all assume that each operation uses only one type of resource during its execution. However, this is not always true for real processors. Rau and Fisher [80] describe a generic ILP processor with two integer and two floating point functional units, but without a dedicated branch unit. This processor executes branch operations on its second integer unit. Clearly, for an instruction that contains a branch operation, only one integer operation can be issued. In the Hewlett-Packard HP-8000 [56], an out-of-order superscalar processor, branches also use integer operation resources during their execution. For example, each branch is stored not only in the memory buffer but also in the ALU buffer of the processor instruction reorder unit [63].

In this section we show how to compute a scheduling bound that accounts for the complex resource usage of branches. Specifically, we assume that each branch operation has a latency of three, and that it uses a branch unit in all three cycles and an integer functional unit during the first cycle. We describe the procedure by applying it to the DAG of Figure 4.1. This DAG is repeated on the left-hand side of Figure 5.3 where we label each operation with its type. For example, the last operation is a branch that takes three cycles and has three incoming dependence edges.

Unlike the write port case, the branches use existing types of resources. We thus first create a new unit-latency integer operation with the same predecessors and successors as the branch operation in the original DAG. The two operations, the branch and new integer operation, approximate the resource usage of the original branch. The modified dependence graph is shown on the right-hand side of Figure 5.3.
Figure 5.3: Deriving a bound that accounts for the branch resource usage

We then apply the LC algorithm for each resource type separately. We traverse the modified DAG in a topological order, and for each operation we calculate bounds with respect to the integer, memory, and branch resources. Each single-type bound is a lower bound on the operation completion time. The tightest of these bounds represents a lower bound that is aware of the branch operation resource usage.

We implemented this bound to evaluate the backtracking schedulers from the previous chapter. The bound was used to demonstrate that these schedulers generate near-optimal schedules for detailed processor models. We found that the bound is only between 1.9% and 3.6% from the best of the three backtracking schedulers on the six processor configurations 111L1, 211L1, 111L2, 211L2, 111L3, and 211L3. This small gap demonstrates the high quality of our systematically derived lower bounds.

5.4.3 Issue constraints

We describe in detail the issue restrictions on scheduling present in the Intel Itanium processor [58]. This EPIC-style processor has nine functional units: two integer, two floating-point,
two memory, and three branch units, so in our notation it is referred to as a 2223 processor. While all the computational functional units are fully pipelined, they are generally asymmetric with respect to the set of operations they can execute. For example, even though there are two integer units, I0 and I1, only I0 can execute the `shl` operation. Eight rules define precisely the classes of operations that each of the nine functional units can execute.

The Itanium processor issues and executes operations in software-supplied order, and it makes no attempt to reorder operations to avoid stalls. An operation cannot be issued out-of-order even if there are sufficient resources for its execution. Therefore, the compiler must keep track of the number, type, and order of operations inside an instruction to avoid unnecessary stalls.

Each Itanium instruction (bundle) includes three operations. The hardware issues at most two instructions, i.e. six operations, at a time as shown in Figure 5.4. An instruction rotation causes new instructions to be brought into the two-instruction window of operations being considered for issue. The following rules govern the instruction rotation [59]:

- Any stall due to an operation from the first instruction of the issue window disallows any instruction rotation.
- In the absence of a stall due to an operation from the first instruction, any stall
due to an operation from the second instruction of the issue window causes a single instruction rotation to occur.

- If all the six operations from the issue window are issued, a double instruction rotation results.

To simplify the dispersion logic, i.e. the mapping of operations to functional units, as well as the resource oversubscription check, the Itanium instruction set architecture features instruction templates. An instruction template is an arrangement of three operation types in an instruction. The 128-bit instruction encoding thus contains not only three 41-bit operation slots but also a 5-bit template field. The template field indicates the type of each operation: integer (I), floating-point (F), memory (M), or branch (B), and the order of the three types in an instruction.

Certain operation types can only occur on specific slots within an instruction. Further, only a subset of all possible mappings of operations to functional units are allowed. With some simplification, there are nine valid instruction templates: MII (one memory and two integer operations, placed in that order), MMI, MFI, MMF, MIB, MBB, BBB, MMB, and MFB.

Not all $9 \times 9 = 81$ possible pairs of two instruction templates can be issued together without stalling the processor. Some pairs are impossible because of resource oversubscription. For example, the MII and MFI instruction templates cannot be issued in one cycle since this pair contains three integer operations, and the processor only has two integer functional units.
In addition to resource oversubscription, there are several Itanium processor-specific special cases based on instruction templates and operation types [59]:

- An instruction with the MMF template (an MMF instruction, for short) causes a stall if it is paired with any other instruction.

- An BBB instruction causes a stall if it is the first instruction in the pair; the same restriction holds for an MBB instruction.

- An MIB instruction causes a stall if it is the first instruction in the pair, unless the B type operation is a \texttt{nop}, \texttt{b} or \texttt{brp}; the same restriction holds for an MFB and MMB instruction.

A lower bound on the length of a schedule for the Itanium can be derived as follows. We first count the number of all pairs of instruction templates that can be issued in a cycle. Taking into consideration the number of functional units of each type to avoid resource oversubscription as well as the first two of the above three special cases, there are 18 ordered pairs of instruction templates that can issue together without causing a stall:

- (MII MBB), (MII BBB), (MII MFB),
- (MMI BBB),
- (MFI MFI), (MFI MIB), (MFI MBB), (MFI BBB), (MFI MFB),
- (MIB MFI), (MIB MIB), (MIB MBB), (MIB MFB),
- (MFB MII), (MFB MFI), (MFB MIB), (MFB MBB), and (MFB MFB).

Since a compiler can place the six operation types within a pair in any order, from the compiler perspective there are only 10 different groups of 6 operation types that can issue
in parallel without a stall:

(MM BBB I), (MM BBB F),

(MM BB II), (MM BB FF), (MM BB I F),

(MM B II F), (MM B I FF), (MM II FF),

(MBB II), and (M BBB I F).

We further relax the scheduling problem by also allowing the remaining 7 forbidden combinations:

(M BBB I F), (M BB II F), (M BB I FF), (M B II FF),

(BBB II F), (BBB, I FF), and (BB II FF).

That is, at most two instructions can still be issued in a cycle, but all possible groups of 6 operation types are allowed. Note that scheduling in this relaxed problem is more restrictive than scheduling on a processor with six identical general-purpose functional units. Therefore, we can use the RJ and LC algorithms on the problem of scheduling on the GP6 processor and thus obtain a lower bound on the schedule length for the Itanium processor.
Chapter 6

Conclusions and future work

In this thesis, we have developed scheduling algorithms and tight bounds to improve the efficiency of compiled code on ILP processors. Our scheduling approaches are aware of the complexity of target processors, and thus apply to very detailed processor models. We designed the backtracking approach for acyclic instruction scheduling motivated by branch conflicts and resource conflicts that cannot be addressed adequately by conventional, non-backtracking, schedulers. Our generic backtracking scheduler only unschedules successors of the current operation and provably terminates without compromising the performance. The more aggressive backtracking scheduler, FullBT, assigns operations in priority order and permits any operation to unschedule already scheduled operations. The selective backtracking scheduler SelectBT assigns operations primarily in dependence order and therefore greatly reduces the need for backtracking. Experiments demonstrate that our backtracking schedulers successfully fill a significant fraction of branch delay slots, outperforming the conventional scheduler by about 10%. The overall performance improvement is between
1 - 7%. We conclude that with a small amount of additional computation, backtracking-based schedulers can effectively improve schedule quality.

We developed and thoroughly evaluated a group of resource-aware scheduling bounds. We showed that a tight bound on the basic block schedule length can be obtained by replacing precedence constraints with early and late times, and by minimizing the maximum lateness of the basic block operations. We demonstrated that recursively applying this approach yields a bound that is provably tighter than other known bounds, experimentally shown to achieve the optimal value at least 86.5% of the time, and useful for deriving bounds for superblocks. We also systematically designed new bounds to address complex resource usage of certain operations, register file write ports constraints, and issue restrictions. For a processor in which branch operations use branch and integer resources, our lower bound is within 3.6% of an upper bound. For a processor with a fixed number of register file write ports, our resource-aware bound is 17.5% better than the tightest known lower bound over an image compression benchmark. We conclude that extending the dependence graph with resource-type nodes and edges can lead to tight lower bounds on scheduling in the presence of a wide variety of issue and resource constraints.

We evaluated a family of combinatorial uniprocessor-based scheduling algorithms and demonstrated that they can be used to achieve good performance, within 1.5% of optimal over a large synthetic benchmark, for a processor with multiple functional units. We also designed several ways to create feasible schedules from non-integral solutions to a new linear programming relaxation for the multiple machine problem. Our best LP-based approach finds schedules that are within 0.2% of optimal over the same benchmark. We formulated
the superblock scheduling as a weighted completion time scheduling problem and compared our algorithms against algorithms designed specifically for superblock scheduling. For SPECint95 instances with arbitrary precedence constraints, our LP-based algorithm clearly outperforms all other known algorithms and finds schedules that for a six-issue VLIW processor are within 1.4% of optimal. We conclude that for assigning static priorities of operations, which is a necessary step in any instruction scheduler, an LP-based approach can be the best choice.

Although the investigations in this dissertation provide a solid understanding of the resource-aware scheduling for modern ILP processors, there are still several open issues that can lead to promising future research. While we developed the backtracking approach for acyclic instruction scheduling as a general scheduling policy for dealing with resource and issue constraints, further studies can evaluate the effectiveness of backtracking in addressing particular architectural or microarchitectural features. Finding the best forcible scheduling policy and operation static priority in each case represents a challenging problem. On the theoretical front, we would like to extend the termination proof for the aggressive FullBT and SelectBT schedulers. We have shown that adding appropriate resource edges and nodes in a scheduling region's dependence graph is useful to systematically derive tight resource-aware bounds. On the other hand, our general LP-based list scheduling algorithms easily outperform the more specific superblock schedulers. Therefore, any combination of the two approaches seems promising. As one example, we can add the new resource constraints to the LP formulation and obtain LP-based resource-aware list scheduling algorithms.
Bibliography


102


