Unification of Register Allocation and Instruction Scheduling in Compilers for Fine-Grain Parallel Architectures

by
David A. Berson
B.A., Coe College, 1984
M.S., University of DePaul, 1988

Submitted to the Graduate Faculty of
Arts and Sciences in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

University of Pittsburgh
1996

UNIVERSITY OF PITTSBURGH

FACULTY OF ARTS AND SCIENCES

This dissertation was presented

by

David A. Berson

It was defended on

Novemember 21 1996

and approved by

Prof. Rajiv Gupta

Prof. Mary Lou Soffa

Prof. Henry Chuang

Prof. Thomas Cain

Committee Chairperson

UNIFICATION OF REGISTER ALLOCATION AND INSTRUCTION SCHEDULING IN COMPILERS FOR FINE-GRAIN PARALLEL ARCHITECTURES

David A. Berson, Ph.D.
University of Pittsburgh, 1996

The interaction between instruction scheduling and register allocation has significant impact on the quality of code generated, particularly in compilers targeting fine grain parallel architectures. The problem results from the fact that instruction scheduling and register allocation have conflicting goals. Instruction scheduling tries to maximize parallelism by scheduling as many instructions as possible in parallel, which requires a large number of values to be held in registers for short periods of time. On the other hand, register allocation attempts to hold a small number of values in registers for long periods of time, resulting in limiting the number of instructions that can be scheduled in parallel.

This dissertation presents a method for unifying these tasks by allocating all needed registers and functional units to an instruction simultaneously. No previous technique has achieved this degree of integration between the two tasks. The work in this dissertation is based on a framework consisting of three components: a technique for measuring a program's demand for all resources, a single intermediate representation of the measured demands, and a set of transformations that perform resource allocation.

The approach taken in this work is based on a new paradigm of resource allocation, called *Measure and Reduce* in which the resource requirements of the program are measured and excessive demands are removed by reduction transformations. The information computed during the measurement of the demands for each resource is incorporated into a single intermediate representation. The reduction transformations for all resources operate on this intermediate representation, allowing transformations for different types of resources to be performed simultaneously. Therefore, an instruction can be allocated all resources it needs at once, resulting in unified resource allocation. The intermediate representation is based on a hierarchical form of dependence DAGs, enabling the transformations to naturally handle instruction level parallelism. In particular, the register transformations form a framework for live range splitting in the absence of a full ordering of the instructions, as required by previous splitting techniques.

Application of the reduction transformations is first demonstrated by a heuristic for performing register allocation during local instruction scheduling. Global register allocation is performed by exploiting the hierarchical nature of the intermediate representation. Heuristics are also given for using the transformations during global code motion, resulting in unified allocation and a more flexible use of available resources than previous resource constrained techniques.

The results of numerous experiments comparing the new techniques to previous attempts at phase integration are reported. The experiments indicate that the unified allocation of resources generates higher quality code than methods that partially integrate the allocation phases. In addition, while precise measurement of register requirements is NP-Complete, in practice precise measurements are obtained easily and efficiently. Thus, when these measurements are combined with traditional register allocation techniques in hybrid algorithms, the quality of code generated is improved.

In memory of Elizabeth Marie Hartquist, who saw her son start.

For everything there is a season, and a time for every matter under heaven.
Ecclesiastes 3:1

Acknowledgements

During the phases of this research I have been aided by a number of people. I take this opportunity to thank them for the various contributions they have made to help me reach my goals.

I would like to express my appreciation and thanks to my co–advisors, Drs. Rajiv Gupta and Mary Lou Soffa for their guidance, examples as teachers and researchers, and their patience. I am also grateful to the members of my thesis committee, Drs. Henry Chuang and Thomas Cain, for their review and suggestions concerning this research.

I acknowledge the Dean of Faculty of Arts and Sciences Office and the Mellon Foundation for providing financial support in the form of fellowships. This work was also supported in part by the NSF[1].

I would like to give special thanks to my fellow graduate students. Thanks to Claude-Nicholas Fiechter, Russ Bodik, and Phil Kamp for their work on the Program Dependence Graph C Compiler and for listening to all my suggestions and implementing the reasonable ones.

I would also like to thank the Intel Corporation for their willingness to hire me before I was quite finished and their patience while I completed the writing.

I would especially like to thank my wife, Karen, for her constant love, emotional support, and understanding through all the stages of this degree. I give my deepest appreciation to my daughter, Alannah, for her child's unconditional love and being my motivation to finish. I would like to thank my parents for their love and for instilling their value of education in me. Without their belief in my abilities and continual encouragement I never would have come so far.

Finally, I thank God for this experience. Without His love and gifts none of this would have been possible.

[1] National Science Foundation Presidential Young Investigator Award CCR-9157371 and Grant CCR-91090808 to the University of Pittsburgh

Table of Contents

List of Figures . . . x

List of Tables . . . xii

1 Introduction . . . 1
- 1.1 Previous Approaches . . . 2
- 1.2 Unification . . . 5
- 1.3 Organization of the Thesis . . . 7

2 Related Work . . . 8
- 2.1 Program Representations . . . 8
- 2.2 Global Code Motion . . . 9
- 2.3 Register Allocation . . . 10
- 2.4 Integrated Register Allocation and Instruction Scheduling . . . 13

3 Overview . . . 15
- 3.1 Unified Resource Allocation Framework . . . 15
- 3.2 URSA Applications . . . 17

4 Resource Requirements . . . 20
- 4.1 Measuring Resource Requirements . . . 21
  - 4.1.1 Measurement definitions . . . 21
  - 4.1.2 Measurement specifics for functional units and registers . . . 25
- 4.2 Excessive Sets . . . 27
- 4.3 Resource Holes . . . 28

5 Global Unified Resource Requirements Representation . . . 32
- 5.1 Integrated Resource Allocation Representation . . . 32
  - 5.1.1 Integrated Resource Allocation Properties . . . 33
  - 5.1.2 GURRR . . . 33
- 5.2 Computing GURRR . . . 40
  - 5.2.1 Construction of GURRR . . . 40
  - 5.2.2 Incremental Updating of GURRR . . . 41

6 Resource Spackling . . . 43
- 6.1 Filling Spanning Resource Holes . . . 44
- 6.2 Filling Non-spanning Resource holes . . . 53

7 Unified Allocation . . . 55
- 7.1 Local Scheduling and Register Allocation . . . 55
- 7.2 Finding Overlapping Holes . . . 58
- 7.3 Selection Heuristics . . . 59

8 Global Code Motion . . . 62
- 8.1 Resource Conscious Global Code Motion . . . 62

8.2 Selection of Fill Sets . . . 68

9 HARE . . . 69
9.1 Overview of HARE . . . 69
9.2 Allocation and spill code placement . . . 72
9.3 Assignment and placement of copy instructions . . . 77

10 Architectural Considerations . . . 81
10.1 Instruction Pipelining . . . 81
10.2 Modeling Architectural Constraints . . . 86
10.2.1 Reserved Resource Copies . . . 86
10.2.2 Generic Instructions . . . 88

11 Implementation . . . 91
11.1 URSA Interfaces . . . 91
11.2 Representation . . . 93
11.3 *Reuse* DAG Decomposition . . . 96

12 Experimentation . . . 99
12.1 Experimental Design . . . 99
12.2 Overview of the Algorithms . . . 101
12.3 Register Sensitive Schedulers (RSS) . . . 102
12.3.1 Base Techniques . . . 103
12.3.2 Hybrid Register Sensitive Schedulers . . . 104
12.4 Schedule Sensitive Register Allocation (SSRA) . . . 106
12.4.1 Base Techniques . . . 106
12.4.2 Hybrid SSRA techniques . . . 107
12.5 Unified Resource Allocation . . . 109
12.6 Measurement and Compile Time Statistics . . . 112
12.6.1 Measurement Heuristics . . . 112
12.6.2 Compilation Time . . . 115
12.7 Comments . . . 115

13 Concluding Remarks . . . 117
13.1 Summary . . . 117
13.2 Future Work . . . 119

Appendix A NP-Completeness and a Heuristic for Computing $Kill()$ . . . 122
A.1 NP-Completeness . . . 122
A.2 Computing $Kill()$ . . . 124

Appendix B Speedup Tables . . . 128

Bibliography . . . 147

List of Figures

1.1 Intermediate program representations 3

1.2 Resource conscious schedule 5

3.1 Comparison of back end organizations 18

4.1 Example code and corresponding DAG 22

4.2 Function *measureRequirements()* 24

4.3 A bipartite graph matching 25

4.4 A complex case for defining *Kill()* 25

4.5 Example DAG of a basic block 30

5.1 Example of GURRR 36

5.2 Sample Code and GURRR Dependence Subgraphs 38

5.3 GURRR Resource Usage Information 39

6.1 Filling a non-spanning resource hole 44

6.2 Procedure *spackleSpanning()* 46

6.3 Procedure *reduceSpanning()* 47

6.4 All unique orderings of two interfering spanning instances 51

6.5 Filling a non-spanning resource hole 53

7.1 Function *reduceBlock()* 56

7.2 Local reductions of resource requirements 57

7.3 Function *findOverlappingHoles()* 58

8.1 Code motion techniques and architectural supports 63

8.2 Function *fill()* 64

8.3 Example of global code motion 65

9.1 Top level register allocation algorithm 70
9.2 Example of coalescing 72
9.3 Region 1 Register usage 72
9.4 Example of rematerialization 73
9.5 CFGs for the spill options 74
9.6 Allocation cost estimation algorithm 75
9.7 Assignment algorithm 78
9.8 SSA assignment algorithm 79
9.9 Final register assignment 80

10.1 Pipeline Example 84
10.2 Relaxation procedures 84
10.3 DAG with reserved copy instructions 87
10.4 DAG with a generic requirement 89

11.1 Chain decomposition steps 97

12.1 Comparison of base RSS techniques 103
12.2 Comparison of hybrid RSS techniques 105
12.3 Comparison of SSRA and RSS techniques 107
12.4 Comparison of hybrid SSRA techniques 108
12.5 Comparison of variant URSA techniques 110
12.6 Comparison of base and best techniques 111
12.7 Comparison of compilation times 114

A.1 A complex case for defining *Kill()* 123
A.2 A special case for partitioning into bipartite subDAGs 125
A.3 All combinations of Out and In nodes 125
A.4 Function *computeKill()* 127

List of Tables

4.1 Computation of hole properties 29

6.1 Invalid and symmetrical orderings removed from consideration 50

6.2 Splittings of interfering spanning instances 52

12.1 Benchmarks used for experimentation 100

12.2 Critical path lengths and number of instructions for 2-4 architecture 111

B.1 Individual speedups for the rss heuristics on architecture 2-4 128

B.2 Individual speedups for the rss heuristics on architecture 2-8 129

B.3 Individual speedups for the rss heuristics on architecture 2-16 129

B.4 Individual speedups for the rss heuristics on architecture 2-32 130

B.5 Individual speedups for the rss heuristics on architecture 4-4 130

B.6 Individual speedups for the rss heuristics on architecture 4-8 131

B.7 Individual speedups for the rss heuristics on architecture 4-16 131

B.8 Individual speedups for the rss heuristics on architecture 4-32 132

B.9 Individual speedups for the rss heuristics on architecture 6-8 132

B.10 Individual speedups for the rss heuristics on architecture 6-16 133

B.11 Individual speedups for the rss heuristics on architecture 6-32 133

B.12 Individual speedups for the ssra heuristics on architecture 2-4 134

B.13 Individual speedups for the ssra heuristics on architecture 2-8 134

B.14 Individual speedups for the ssra heuristics on architecture 2-16 135

B.15 Individual speedups for the ssra heuristics on architecture 2-32 135

B.16 Individual speedups for the ssra heuristics on architecture 4-4 136

B.17 Individual speedups for the ssra heuristics on architecture 4-8 136

B.18 Individual speedups for the ssra heuristics on architecture 4-16 137
B.19 Individual speedups for the ssra heuristics on architecture 4-32 137
B.20 Individual speedups for the ssra heuristics on architecture 6-8 138
B.21 Individual speedups for the ssra heuristics on architecture 6-16 138
B.22 Individual speedups for the ssra heuristics on architecture 6-32 139
B.23 Individual speedups for the ursa heuristics on architecture 2-4 140
B.24 Individual speedups for the ursa heuristics on architecture 2-8 140
B.25 Individual speedups for the ursa heuristics on architecture 2-16 141
B.26 Individual speedups for the ursa heuristics on architecture 2-32 141
B.27 Individual speedups for the ursa heuristics on architecture 4-4 142
B.28 Individual speedups for the ursa heuristics on architecture 4-8 142
B.29 Individual speedups for the ursa heuristics on architecture 4-16 143
B.30 Individual speedups for the ursa heuristics on architecture 4-32 143
B.31 Individual speedups for the ursa heuristics on architecture 6-8 144
B.32 Individual speedups for the ursa heuristics on architecture 6-16 144
B.33 Individual speedups for the ursa heuristics on architecture 6-32 145

Chapter 1

Introduction

The relentless push for more computing power has brought computer architectures that exploit instruction level parallelism (ILP) into every day environments in the form of superscalar and VLIW workstations and desktop computers. With each advance in computer architectures, the synergy between architectures and compilers becomes more apparent, presenting new challenges for compilers. These challenges introduce new complexities to previously addressed tasks and highlight the drawbacks of the way in which compilers were structured for previous architectures. This dissertation presents new methods to generate higher quality code for ILP architectures by creating advanced techniques for register allocation and instruction scheduling. These techniques are specifically designed to address the new complexities caused by ILP while avoiding the drawbacks resulting from straightforward extensions of previous register allocation and instruction scheduling techniques.

Because of the number and complexity of tasks that must be performed in a compiler, the compilation process has been divided into manageable units called phases. These phases are grouped together based on the representation of the program that they use. Each phase typically solves one particular problem. As a result of addressing each problem separately, effective heuristics can be developed for each problem, allowing a good solution to the individual translation problems. However, the results of one phase can have an impact on the solution of the next phase, and thus the overall quality of the generated code. This impact can have a negative result on the overall code quality.

The separation of the compilation process into phases does not consider possible interactions between the tasks performed. For example, consider the interaction between instruction scheduling and register allocation. Both precedence constrained instruction scheduling and register allocation are well known NP-complete problems. Therefore, for purposes of achieving reasonable compile times, heuristics are used for both tasks. Instruction scheduling tends to require a large number of values to be live in registers to keep all of the functional units busy. On the other hand, register

allocation tends to keep fewer values live at a time in an effort to avoid the need for expensive memory accesses through register spills.

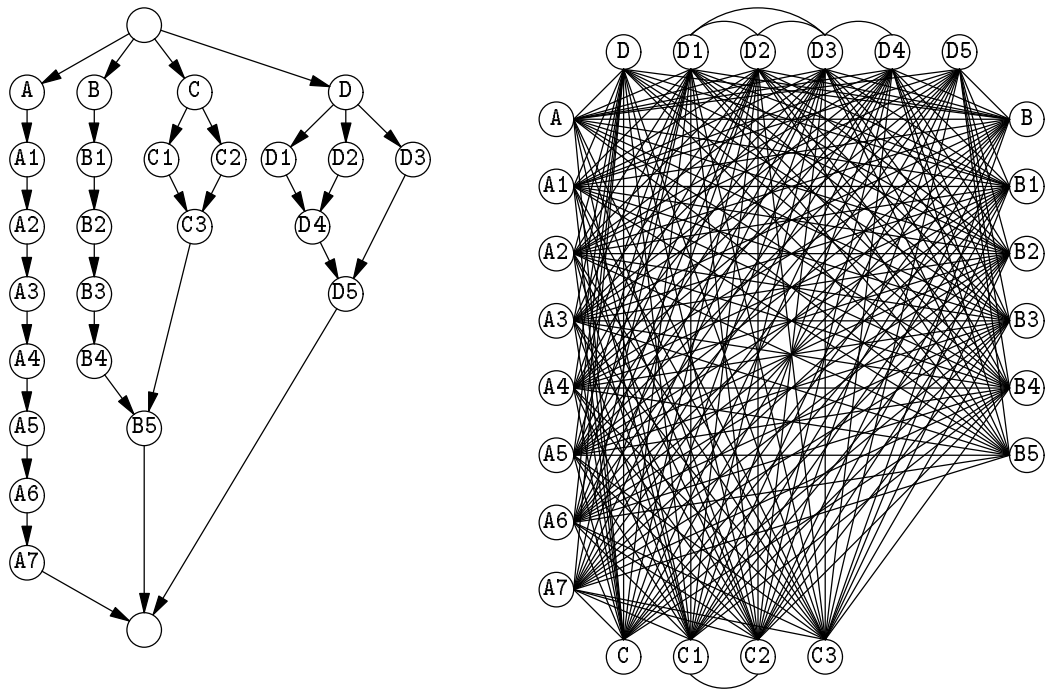
If register allocation is performed first, it limits the amount of ILP available by introducing additional dependences between the instructions based on temporal sharing of registers. If instruction scheduling is performed first it can create a schedule demanding more registers than are available, creating more work for the register allocator. In addition, the spill code subsequently generated must be placed in the schedule by a post pass cleanup scheduler.

In addition to the problem of interactions between the register allocation and instruction scheduling phases, the heuristics themselves must be changed to compensate for the added complexities of ILP. To fully exploit ILP the sequential order in which the instructions appear in the source code must be replaced with the partial ordering imposed by the data and control dependencies inherent in the program. In this partial ordering there is some freedom for instructions to move around in the schedule. While some instructions are on a maximum length path through the ordering, others have slack time in when they can be scheduled. Techniques must be developed to guide the selection of an instruction order that exploits ILP while maintaining resource requirements at a level supported by the architecture.

The goal of this research is to redesign the compiler back end to unify the phases performing resource allocation and to make the remaining phases, such as global code motion, conscious of their impact on resource allocation and thus the resulting execution time of the program being compiled. Performing the allocation of all resources simultaneously achieves a higher degree of integration than previous techniques proposed. The benefit of this degree of integration is that the impact of all allocation decisions can be assessed in terms of the overall resource allocation problem and thus the quality of code generated.

1.1 Previous Approaches

Approaches to integrating register allocation and instruction scheduling can be characterized by three properties: the degree of exploitation of ILP, the degree of integration, and the representations used. The degree of exploitation of ILP can be viewed as the extent to which the information about inherent parallelism is used during instruction scheduling and register allocation. At one end are heuristics that assume a complete ordering, typically the ordering provided by the programmer. At the other end are techniques that rely solely on the partial ordering determined by the minimal dependences needed to preserve semantic correctness. In the middle are techniques that use the partial ordering



(a) Program DAG

(b) Register Interference Graph

Figure 1.1: Intermediate program representations

for only some tasks, (*e.g.,* instruction scheduling) or use it for parts of a task (*e.g.,* during register allocation information about parallelism may be used for computing live ranges but not for selecting spill points). Such techniques exploit parallelism by selecting orderings of instructions to achieve better resource utilization without unnecessarily reducing available parallelism.

The degree of integration achieved between the heuristics can also vary significantly. At one end are approaches in which the tasks are performed independently in separate phases, resulting in no integration. At the other end is *unification*, that is, both tasks are performed simultaneously in a single phase. A unified phase gives equal consideration to both tasks during each allocation decision. In the middle of the spectrum are a number of approaches that use separate phases for each task but incorporate varying types and amounts of information from one heuristic into the other to add limited "awareness" of the heuristic's impact on subsequent heuristics. Closely related to the integration of phases is the types of representations used for each task. The use of vastly different representations used by instruction scheduling and register allocation presents a major obstacle to integration as incorporation of information in different forms is difficult.

The straightforward approach to generate code for ILP architectures is to first schedule the instructions using list scheduling and then allocate registers using traditional graph coloring algorithms. List scheduling naturally exploits inherent ILP as it traverses the program's instructions.

Coloring based register allocation has no concept of ILP and only takes advantage of parallelism if it happens to generate spill code near idle slots in the schedule. The separation of the tasks offers little opportunity for integration as each task uses a vastly different representation of the program. List scheduling uses a Directed Acyclic Graph (DAG) while coloring base register allocation uses an interference graph. Figure 1.1(a) shows a sample program DAG while Figure 1.1(b) shows the corresponding interference graph. There is no obvious method to incorporate the information in these two representations into one common representation. Thus, it is not obvious how to consider register interferences during instruction scheduling or instruction heights during register allocation.

An alternative to considering register interferences during scheduling is to track register pressure. Goodman and Hsu present such an algorithm [GH88]. In their algorithm the scheduler alternates between two states. In the first state register pressure is low and the scheduler selects instructions to exploit ILP. When the register pressure crosses a threshold the scheduler switches to a second state that selects instructions to reduce register pressure, possibly sacrificing opportunities to exploit ILP in the process. Additionally, no spilling of values is performed. When register pressure falls back below the threshold, the first state is reentered. In this manner scheduling and allocation are partially integrated using a single representation. A problem arises in that the scheduler cannot always select instructions to keep register pressure below the maximum allowed by the architecture. Furthermore, some programs have sufficiently complex register interferences such that some values must be spilled. As a result, a cleanup register allocation phase must be run subsequent to the integrated scheduler. This cleanup phase uses traditional coloring base register allocation, resulting in the degradation of instruction scheduling.

A problem with considering register allocation issues during instruction scheduling is that scheduling for ILP replaces the programmer's complete ordering of instructions with a partial order representing the parallelism available. As a result, live ranges that did not interfere because they were temporally ordered will interfere once the temporal ordering is removed. Recent research has extended register allocation techniques to address this possible overlapping of live ranges [Pin93, NP93]. However, these techniques still do not fully exploit the partial ordering information available to address the problems of which values to select for spilling and where to place the spill code.

A fundamental problem in integrating instruction scheduling and register allocation is the fact that heuristics for each problem use vastly different representations that do not provide uniformly adequate information for both problems being addressed. Consider the program DAG in Figure 1.1(a) and assume that there are three functional units and five registers available to execute the DAG. To exploit all available parallelism seven functional units and seven registers are needed to execute the

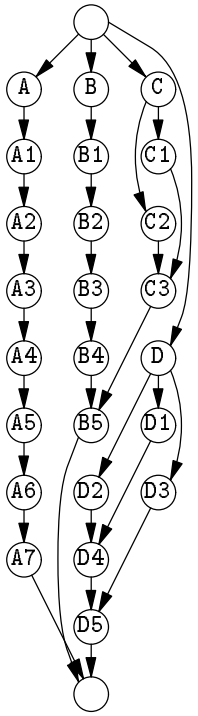


Figure 1.2: Resource conscious schedule

instructions `A1`, `B1`, `C1`, `C2`, `D1`, `D2`, and `D3` in parallel. List scheduling based on critical path lengths would select `A` and `B` for the first instruction issue slot and then would have to make a choice between `C` and `D`. Ideally, the scheduler would be able to "look ahead" and see that scheduling `D` with `A` and `B` would push register pressure over the limit when attempting to schedule subsequent cycles. As a result `C` would be scheduled with `A` and `B`, while `D` would be delayed until more registers are available. The result of such a scheduling decision is shown in Figure 1.2. By delaying `D` the length of the schedule has increased by only one instruction cycle and no spill code is needed.

Previous integration of register pressure into list scheduling is only able to compute register pressure based on the instructions already scheduled. After selecting `A` and `B` for the first slot the register pressure is only two, suggesting that there is no problem. Previous techniques for integrating scheduling information into register allocation guide the selection of which value to spill based on the relative slack times of the candidates for spilling. Unfortunately, slack time does not provide sufficient information to determine when spilling can be avoided.

1.2 Unification

This dissertation develops a new technique to integrate register allocation and instruction scheduling. The technique includes a new representation and algorithms that use the representation to alloca-

tion registers and functional units simultaneously. The representation, the Global Unified Resource Requirements Representation (GURRR), combines information about a program's requirements for both registers and functional units with scheduling information in a single DAG based representation. In this manner GURRR facilitates the determination of the impact of all scheduling and allocation decisions on the critical path length of the code affected.

GURRR is based on a key concept of computing resource requirement measurements, referred to as Unified ReSource Allocation (URSA), which computes sets of instructions that can safely share a single instance of a resource. From this information URSA computes the minimum number of instances of the resource needed to exploit all available parallelism. In terms of registers, this computation determines the maximum number of values that can be alive simultaneously. Although the specific computations of resource sharing differ for functional units and registers, the results of the computations for both types of resources are easily incorporated onto the program DAG.

A byproduct of the computation of how many instances of a particular resource are needed by a program is a set of elements that each represents a group of instructions that can share a single instance of the resource. This information is used to precisely identify the areas of the program where the resource is either over or under utilized. GURRR also incorporates this utilization information in its representation.

The availability of utilization level information suggests a new approach to resource allocation, the *Measure and Reduce* paradigm. This paradigm is based on the observation that allocation decisions are only required when the need for resources exceeds the number actually available. Under this paradigm, the resource usage information is used to select instructions to move from over utilized areas to under utilized areas. This process of moving instructions to the under utilized areas is referred to as *Resource Spackling*. The combination of utilization level information for all resources with direct access to scheduling information enables new unified resource allocation techniques. By allocating all resources needed by an instruction simultaneously, and making allocation decisions based on their impact on the critical path length, a unified allocation phase is achieved.

Advantageous application of Resource Spackling to individual instructions depends on heuristics that select instructions and determine the best place to spackle them. Several such heuristics are presented. The first heuristic applies Resource Spackling to local instruction scheduling and register allocation. This heuristic naturally performs global register allocation when applied to a hierarchical intermediate representation in a bottom up manner. Global register allocation and assignment include other issues such as assignment and copy placement. *Hierarchical Allocation of REgisters* (HARE) was developed as a part of this work to be such a heuristic. A heuristic for incorporating

Resource Spackling into global code motion is also presented. These heuristics must also consider architectural features such as pipeline interlocks and resources with special usage characteristics. Extensions to the heuristics to address these features are discussed.

1.3 Organization of the Thesis

The remainder of this dissertation discusses the realization of the techniques highlighted here, and addresses the practical considerations of target architectures and comparisons to previous techniques. Chapter 2 discusses previous work by other researchers related to the work presented in this dissertation. Chapter 3 gives an overview of the components developed in this work and how they fit together. Chapter 4 describes the measurement of resource requirements. Chapter 5 shows how the resource requirements are incorporated into an intermediate representation. Chapter 6 gives the theoretical basis for performing resource allocation using the information computed and provided in the intermediate representation. Chapter 7 describes heuristics for performing resource allocation for functional units and registers simultaneously in a local scheduler. Chapter 8 describes how unified resource allocation is performed during global code motion. Chapter 9 addresses issues related to global register allocation and assignment. Chapter 10 incorporates ILP architectural features and constraints into the resource measurement and allocation model. Chapter 11 describes important methods used in the prototype implementation. Chapter 12 presents and analyzes the experiments performed in this work. Finally, chapter 13 contains the conclusions of this work and directions for future research.

Chapter 2

Related Work

Register allocation and instruction scheduling are well known problems in compiler research and their interactions have been previously studied. This chapter discusses research in these areas as it relates to the dissertation, as well as the intermediate representations on which the heuristics for these problems depend.

2.1 Program Representations

The contribution of each in task in the compiler to the quality of code generated is dependent on the power of transformations enabled by the intermediate representation used. In particular, instruction scheduling and global code motion depend on specific properties of the representation to perform powerful transformations.

Traditionally, compilers have used the Control Flow Graph (CFG) and basic block dependence DAGs as the intermediate representations for instruction scheduling due to their straightforward computation and concise representation of the program. The CFG is also used to collect a variety of information, including dataflow information and value live ranges, used to construct register interference graphs [ASU86]. A number of extensions to the CFG have been created to aid the various tasks performed in the back end of the compiler.

In order to provide a larger scope for global code motion heuristics, basic blocks have been grouped together in a variety of ways. Traces [Fis81], Super Blocks [HMC+93], and Hyper Blocks [MLC+92] each form collections of basic blocks which satisfy properties required by specific scheduling techniques.

The Program Dependence Graph (PDG) is another representation which combines control and data dependence information in a way that simplifies many transformations [FOW87]. Control dependencies are used to identify *regions* of instructions that execute under the same conditions.

Regions support more powerful global code motion techniques than are possible on CFG based representations [GS90, BR91].

Another method for enabling global code motions is to convert the intermediate representation into Static Single Assignment (SSA) form, which uniquely assigns names to each definition of a variable [RWZ88]. The use of unique names simplifies constant propagation and other analysis [AWZ88]. Furthermore, SSA removes false dependences which would otherwise limit the range of motion for instructions, *i.e.,* due to anti and output dependencies. SSA was originally formulated on the CFG but has been incorporated in PDG based representations [BMO90].

The Program Structure Tree (PST) is a hierarchical representation that can be used by divide-and-conquer algorithms to speedup dataflow analysis and computation of SSA [JPP94]. A number of representations allowing direct interpretation have been proposed, including the Dependence Flow Graph [PBJ+91, JP93] and Value Dependence Graph [WCES94]. However, these representations do not directly identify the control dependencies desired by region-based global code motion algorithms. The Program Dependence Web [BMO90] is an interpretable representation that places a variation of SSA form on the PDG.

While many of the representations mentioned support powerful forms of instruction scheduling and global code motion, none of them provide resource usage information such as total register and functional unit demands. As a result, these representations cannot be used as is for unified resource allocation.

2.2 Global Code Motion

The movement of instructions between basic blocks is limited by both data and control dependences. In some cases architectural features can reduce these limitations. Speculative execution allows instructions to be moved above conditional branches by placing the results of the moved instructions in a shadow area [HP87, SHL92]. These results are then committed to the register file and memory only if and when the conditions for the instructions' execution are later met. Guarded execution associates a predicate with each instruction to be executed [DHB89, HD86]. The predicate represents the conditions under which the instruction should be executed. Instructions can be moved below a join in the control flow graph by using guarded execution to determine if the instruction should be executed or not. These techniques allow compilers more options when performing code motion.

Several compilation techniques that perform code motion to increase ILP have been developed. These techniques include Trace Scheduling [Fis81], Percolation Scheduling [AN88], and

Region Scheduling [GS90]. Each of these techniques has a different method of identifying instructions that may be moved and considers different approaches in selecting their potential destinations.

Trace Scheduling creates large basic blocks, called traces, consisting of sequences of basic blocks along a program path that has a high probability of being executed. Code reordering within a trace is used to generate a good schedule for the trace at the expense of decreased performance on off-trace blocks due to the insertion of compensation code in those blocks. Percolation Scheduling uses code motion operations on a control flow graph. The operations perform less code duplication than trace scheduling and place fewer restrictions on the movement of instructions. Region Scheduling's code motion operations operate on a Program Dependence Graph (PDG) [FOW87]. Region Scheduling identifies the largest set of potential destinations for an instruction and produces the least amount of code duplication.

Each of the above techniques uses list scheduling to schedule instructions and considers only functional units. There are several extensions to Percolation Scheduling to handle resource constraints. The first presents the idea of using a heuristic to control the application of operations in the presence of functional unit constraints [EN89]. Register constraints have also been addressed in a limited manner [ME92, Nor95]. However, these extensions do not consider using spilling to improve usage.

Another technique for increasing ILP across basic block boundaries is Shape Matching [MGS92]. This technique handles only functional units and attempts to overlap the ends of adjacent blocks. It is similar to Moon and Ebcioğlu's approach in that available resources in the middle of a block are not considered.

The scheduling techniques discussed so far have concentrated on using all available functional units, while mostly ignoring their impact on register allocation. Scheduling which consider the impact of scheduling on register allocation are discussed in section 2.4.

2.3 Register Allocation

Traditionally, register allocation is performed by coloring an interference graph, whose nodes represent live ranges and edges the interference between the live ranges [CAC+81, Cha82]. This process assigns colors representing registers to live ranges. When there are insufficient registers available a live range is spilled to memory. Such a live range must be loaded into a register before each use and written back to memory after each definition. A priority function is used to select which live range to spill. The goal of the priority function is to minimize the number of memory accesses, both as a

result of the number of live ranges spilled and the number of times each value must be accessed from memory. The priority function is based on the cost of a spill divided by the number of interfering live ranges.

Several enhancements to the basic coloring process have been proposed to reduce the number of spills generated by the coloring process. Briggs suggests several heuristics for simplifying the interference graph to increase the likelihood of coloring the graph without resorting to spilling [BCKT89]. Although the original proposed priority function included an execution estimate factor in the computation of the cost of a spill, most subsequent methods have favored a factor using the loop nesting depth instead. The use of the loop nesting depth is motivated by the observation that spills inside of nested loops are executed more frequently than those at a shallower nesting depth or outside loops. Bernstein *et al.* use a combination of three different priority functions to select values for spilling that will remove the most interferences from the graph [BGM+89].

Live range splitting was introduced in an effort to reduce the cost of spilling [CH90, KH93]. When spilling is required the live range selected for spilling is split into several smaller live ranges. The smaller live ranges are then treated as separate values requiring registers. The essential idea is that some of the smaller live ranges will not need to be spilled and so will not need to access memory. Memory accesses will only be required for those smaller live ranges that are spilled.

Hierarchical register allocation has been introduced as a method for reducing the number of dynamic memory references [CK91]. This technique creates *tiles* corresponding to basic blocks in the control flow graph. The tiles are colored from the inside out with respect to the nesting of control structures. The result is that values in loops have a better chance of remaining in registers. This results in spilling *pass-thru* values, those values that are alive at entry and exit and are not used in a block, [CAC+81]. More recently, hierarchical allocation has been extended to PDGs [NP94].

These hierarchical approaches suffer from two problems. First, they only consider one case of placing spills in less frequently executed locations, *i.e.,* outside of loops. They do not try to place more spills in conditionally executed code and fewer spills in unconditionally executed code. Second, once a tile or region has been allocated registers, its values are never candidates for future spilling. The relative execution frequencies may be such that spilling inside a previously allocated tile or region results in a lower overall execution time.

The RASE technique attempts to balance the use of registers by local and global values by computing a cost function for each basic block [BEH91]. The cost function estimates the increase in the block's critical path length for a given number of registers. The cost includes a factor of the execution frequency of the block. The interference graph represents the number of registers required

by each basic block and the cost function is used to select basic blocks that should have their number of allocated registers reduced. Although RASE considers both execution counts and resulting critical paths lengths, it has a limited hierarchical view of registers, by viewing them as either global or local.

Probabilistic register allocation takes a different approach to register allocation [PF92]. This technique is based on the principle that the probability that a value is still in a register when an instruction needs to use it is roughly proportional to the inverse of the distance from its definition or last use to the current use. The technique uses the distances between definitions and uses of each value to compute initial value probabilities. Probabilities are also assigned to each control branch in the program. These branch probabilities are then multiplied with the initial value probabilities to reflect the effect of different control paths on the final value probabilities. The final value probabilities are then used as priorities for allocating registers to the values. The technique iteratively selects the highest priority value and allocates a register to it and then recomputes the probabilities and priorities of the remaining values.

Probabilistic register allocation does address the cost of spills from the point of view of spilling values that are least frequently executed; it favors spills in shallowly nested loops over deeply nested loops and in conditionally executed code over unconditionally executed code. However, it does not consider the scheduling of the generated spill code.

All of the allocation techniques discussed so far have been developed for single issue architectures. Two recent techniques have considered the impact of multiple issue architectures on the interference graph. Pinter has developed the *parallel interference graph* to represent the additional interferences between live ranges that occur when instructions can be reordered and issued in parallel [Pin93]. Norris and Pollock use a similar interference graph and attempt to reduce register pressure by making some scheduling decisions during register allocation [NP93]. Neither of these techniques addresses the selection of values for spilling to minimize the spill code's impact on the program's execution time.

All of these techniques use some form of the register interference graph as their intermediate representation. Although this graph is typically constructed by performing live range analysis on the CFG, it is quite different from the CFG. As a result, the representations used for instruction scheduling and register allocation do not lend themselves well to incorporating information about the impact of one task on the other.

2.4 Integrated Register Allocation and Instruction Scheduling

Due to the known interaction between register allocation and instruction scheduling and its impact on resulting quality of generated code, previous research has investigated methods for integrating the two tasks. These techniques can be characterized as making one of the two tasks partially aware of its impact on the other. Two such approaches are possible based on the order in which the tasks are performed: *register sensitive scheduling* and *schedule sensitive register allocation.*

The register sensitive scheduling approach performs scheduling prior to register allocation so that the scheduler can make scheduling decisions that keep the demand for registers at or below the number available. Goodman and Hsu use a list scheduling approach that monitors the register pressure of the instructions scheduled in the form of the number of values live at each point in the schedule [GH88]. As each instruction is scheduled the register pressure value is updated by the number of live values defined and killed by the instruction. In its normal mode, the list scheduler selects instructions to schedule which are on the critical path of the block. However, when the register pressure rises above a preset threshold an alternate selection criteria is used. In this situation the scheduler selects instructions which reduce the register pressure by killing more values than it defines. Coloring based register allocation is subsequently performed to insert spills where scheduling was unable to restrict the number of registers needed to the number available. Extensions to this approach haven been made by Bradlee *et al.*[BEH91].

The schedule sensitive register allocator approach performs register allocation prior to instruction scheduling. One such approach is proposed by Bradlee *et al.*[BEH91]. A prepass scheduling phase is performed to construct a cost function for each basic block. These cost functions estimate the minimum number of registers that can be allocated to the block without significantly impacting its critical path length. Register allocation is then performed using register limits computed by the cost functions. A final instruction scheduling phase is then performed.

Another schedule sensitive register allocation approach is described by Norris and Pollock [NP93]. In their approach, the register allocator considers the impact of register allocations on the subsequent phase instruction scheduling. Since register allocation is performed first the instructions are not yet fully ordered. Thus the parallel form of the register inference graph must be used to represent live range interferences. The advantage of using only a partial ordering of the instructions is that register interferences can sometimes be removed by imposing temporal dependences on the instructions so that live ranges do not overlap. The disadvantage is that prior to this dissertation, no methods were known for performing live range splitting on a partial ordering of the instructions. Thus spilling was performed instead.

The integration techniques described were created to address the parallelism available in pipelined architectures. While they can be extended to target multiple issue architectures, they were not designed with the added complexities of these architectures in mind.

An integrated register allocation and scheduling method has also been introduced for software pipelining [NG93]. The goal of the technique is to minimize register requirements for a time optimal pipelined loop. It does not consider actual resource constraints and does not incorporate register spilling. In addition, the formulation of the problem as a linear programming problem limits its application to software pipelining of loops.

Chapter 3

Overview

The research presented in this dissertation consists of a number of components designed to incorporate unified resource allocation into a compiler. As a result, there are many dependencies and interactions among these components. This chapter gives an overview of the concepts and components developed.

First, a framework is presented to support unified resource allocation. Second, compiler back end phases are either replaced with phases that perform unified resource allocation or are modified to take resource allocation into account when making decisions. These new or modified phases achieve unified resource allocation or resource allocation awareness by using the presented framework.

3.1 Unified Resource Allocation Framework

The objective of Unified Resource Allocation is to support both the *Measure and Reduce* paradigm, and the ability to make back end phases aware of their impact on resource allocation. The Measure and Reduce paradigm is based on the observation that resource allocation decisions are only required when there is a demand for more resources than are available. When there are sufficient resources to meet the demand, only resource assignment must be performed, which must also consider its impact on the program's resulting execution time. In this paradigm, the task of resource allocation is viewed as reducing resource usage in the areas with excessive resource demands. These excessive areas must be located by measuring the resource demands of the program. The reductions are accomplished by performing transformations on the intermediate representation of a program. The URSA framework consists of three components: techniques for measuring the program's demands for resources, an intermediate representation of a program that indicates resource demands, and techniques to implement reductions in resource requirements.

The first component of the URSA framework is a set of techniques to compute resource requirements. When compiling a program to exploit ILP, the dependencies in an acyclic segment of the program are used to represent the set of all semantically correct ways of scheduling the segment. Different schedules may result in different resource requirements. The approach taken in the Measure and Reduce paradigm is to remove all schedules that result in excessive resource demands using the reduction techniques. The remaining set of schedules can safely be assigned the available resources. Thus, the measurement technique must consider the worst case schedule in terms of the number of resources required. Each type of resource may have one or more schedules that produces its worst case requirements.

In addition to computing the maximum number of resources required, the measurement techniques must identify the locations in a program where there are excessive resource demands and the locations where a resource is under utilized and available for additional allocations. These three types of resource measures are the result of the analysis techniques developed in this research.

Although resources like functional units and registers have different usage properties, their requirements are measured using a common technique. The technique uses a special relation that models the various usage properties to hide the details from the measurement technique. Issues of the precision of the measurements are addressed when dealing with register types of resources.

By operating on acyclic segments of a program, the measurement techniques can be used by a wide variety of compilers using different intermediate representations. This work concentrates on one particular intermediate representation which supports several advanced phases of interest for integration.

The intermediate representation used in URSA is called the Global Unified Resource Requirements Representation (GURRR). GURRR is based on an instruction level Program Dependence Graph (PDG). The PDG representation was chosen for several reasons. First, the dependencies represented in the PDG indicate which instructions in the program can be executed in parallel and which must be executed sequentially. Second, the PDG provides valuable control dependence information used by powerful global code motion techniques, which are important in exploiting ILP. Finally, the PDG is usable by many common optimization phases; therefore the phases do not have to be rewritten or designed for a new intermediate representation. A GURRR of a program is obtained from its PDG by adding the three types of resource requirements information computed by the first component of URSA.

The third component of the URSA framework is a set of techniques to perform the allocation of resources to instructions. The techniques refer to special nodes in GURRR that indicate resources

available for allocation as *Resource Holes*. The techniques are called *Resource Spackling* because they perform allocations by trying to fill the resource holes with instructions. The spackling techniques compute properties of the resource holes that affect how instructions can be placed in them. Cases are then identified as to whether or not the placement of instructions in a hole will increase the execution time of the program.

3.2 URSA Applications

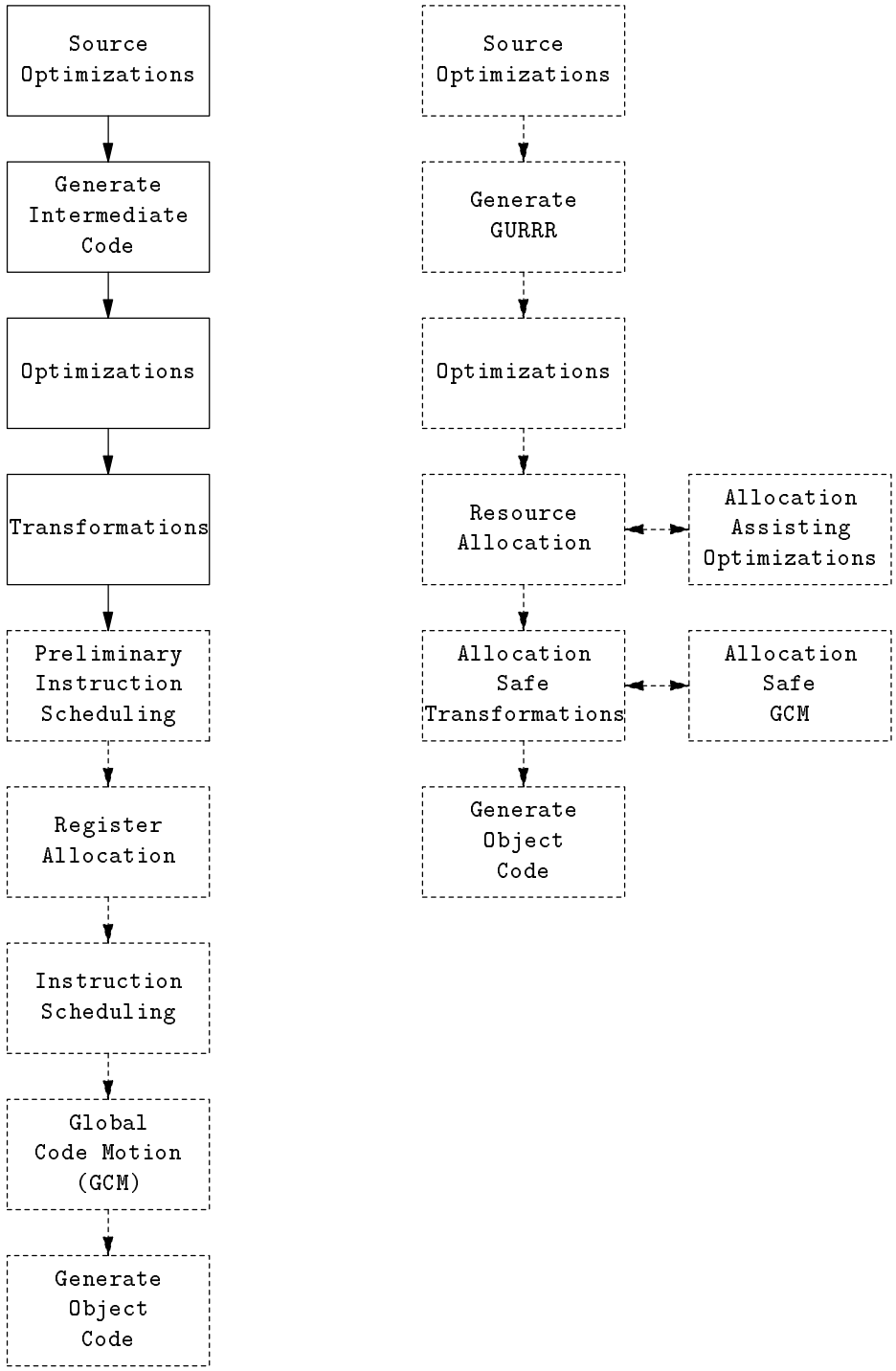
The URSA framework is used as the basis for the new and modified compiler back end phases. GURRR provides the resource requirements measurements in an intermediate representation usable by the phases. The spackling techniques are used in any phase that wishes to allocate resources. This research examines three areas of optimization in the compiler back end: resource allocation, application of transformations such as global code motion, and exploitation of architectural features and constraints.

URSA's spackling techniques are general enough to be used in several allocation schemes. This research develops a reduction phase which replaces the phases for local scheduling and global register allocation. It also presents a global scheduling phase modified to perform unified allocation.

The reduction phase implements the Measure and Reduce paradigm to produce an intermediate representation of a program that can be feasibly scheduled. Conceptually, the reduction phase removes excessive parallelism, either in the form of too many instructions that can be executed in parallel or too many values simultaneously alive in registers, by introducing additional sequentially between the offending instructions. The introduction of sequential dependencies is performed by the reduction transformations. The reduction transformations result in moving instructions from the locations with excessive requirements and placing them in resource holes. Simultaneous allocation is achievable by finding overlapping resource holes for all resources that an instruction demands.

Global scheduling consists of moving instructions between control dependence regions in GURRR to evenly distribute ILP on a larger scale than simply within individual regions. URSA techniques are used to ensure that such global code motions are only performed when there are available resources and would result in a reduction in a program's execution time. The availability of resources is ensured by spackling the moved instructions into resource holes in the destination region.

The application of code transformations serves several purposes. They can be used to introduce more ILP, ideally reducing the execution time of a program. They can also be used to



(a) Traditional Phases

(b) URSA Phases

Figure 3.1: Comparison of back end organizations

enable additional transformations, such as ILP exposing transformations, global code motions, and partial dead code elimination. Transformations interact with resource allocation in two ways. First, the resource requirement measurements can be used to drive the application of transformations. The measurements indicate pairs of areas where resource demands are unbalanced; one area demands too many resources while the other under utilizes them. In such situations either ILP exposing transformations or global code motion should be performed to redistribute the parallelism. When areas are not imbalanced neither redistributing or enabling transformations need to be considered.

The second interaction between transformations and resource allocation is predicting the effect of a transformation. For example, assume at a particular point in a program, resources maybe under utilized and several ILP exposing transformations maybe applicable. The different transformations may each expose a different amount of parallelism. The transformation selected should be the one that exposes an amount of parallelism that most closely matches the unused resources. If too much parallelism is exposed, then the resulting excessive resource requirements that must be reduced after the transformation. The incremental nature of the computation of the resource requirements information enables the effects of each candidate transformation to be estimated and the best one to be chosen.

A comparison of phases between a typical traditional compiler back end and a URSA based compiler is shown in Figure 3.1. In both cases source level optimizations are performed before the generation of their respective intermediate representations. Figure 3.1(a) shows the phases in a traditional compiler back end. Following the application of optimizations and transformations, local and global instruction scheduling is performed and then register allocation is carried out. A second local instruction scheduling phase is performed to schedule any inserted spill code. Finally object code is generated. Figure 3.1(b) shows the phases in an URSA based back end. Most optimizations are performed immediately after the GURRR intermediate form is generated. These optimizations are ones that will always reduce execution time (*e.g.,* strength reduction) or register demands (*e.g.,* copy propagation). Resource allocation is performed next using the spackling techniques of URSA to perform requirements reductions. The allocation phase may elect to perform optional optimizations and transformations that aid the reductions (*e.g.,* reordering commutative operations). Once all resources have been allocated, additional code improvements are performed. These improvements include transformations to uncover appropriate amounts of additional parallelism and global code motions to evenly distribute parallelism and reduce critical path lengths. Such improvements are only performed if there are sufficient resources available for the instructions either locally or in areas reachable through global code motions.

Chapter 4

Resource Requirements

The first step in developing techniques for the measure and reduce paradigm is performing the analysis of a program's resource needs. This chapter discusses the problems to produce such an analysis, including handling of resources with different usage characteristics, operating on segments of a program, and computing the different types of usage information needed.

A number of different intermediate representations are used by the back ends of compilers. All of these representations break up a program into small segments to simplify the tasks performed in the back end. Although different techniques are used to partition the segments and result in different sets of information available about the segments, all representations use a segment that is acyclic. Examples of acyclic segments include basic blocks in a Control Flow Graph [ASU86], superblocks [HMC+93], traces from Trace Scheduling [Fis81], and control dependence regions from a Program Dependence Graph [FOW87]. These different types of acyclic segments are generically referred to as blocks.

Acyclic blocks provide convenient pieces of a program to analyze for resource requirements, as instruction level parallelism is easily expressed and control flow can be considered as needed. Blocks are typically represented using the Directed Acyclic Graph (DAG)[1]. The nodes of the DAG represent the instructions or operations being scheduled. The edges in a DAG represent several types of dependencies, including data dependencies, preservation of semantic correctness, and scheduler imposed temporal dependencies. The temporal dependence edges are added to introduce sequentiality, that is, reduce parallelism where needed to reduce resource requirements. Thus, the DAG representation of dependence information represents a partial ordering of the instructions, *i.e.,* the DAG represents all schedules which honor the required dependencies.

To support the reduction of excess resource requirements, the measurement algorithm must

[1] This work uses the following conventions when representing and discussing DAGs: if node b is dependent on node a then the edge is drawn from a to b, and if a node is not dependent on any other node, it is called a root and is placed at the top of the DAG.

provide several types of information about resource usage in each block. In particular, three types of information must be determined:

1. Total number of resources required;
2. Locations where requirements exceed available resources; and
3. Locations where requirements are less than available resources.

The first item is compared to the number of resources available in the architecture to determine if there are areas with allocation problems. The second item identifies exactly what set(s) of instructions cause allocation problems. The third item indicates where there are additional resources available for allocation.

4.1 Measuring Resource Requirements

This section defines the resource measurements and presents techniques for computing them. The general framework for measuring the various types of resources in the work is described first. The details for the two primary categories of resources are then discussed.

4.1.1 Measurement definitions

The measurements of requirements for the various types of resources in a block are obtained using a single algorithm which operates on a data structure, called a *Reuse* DAG. A *Reuse* DAG indicates which instructions can reuse a resource used by an earlier instruction. The difference in the usage characteristics of the various types of resources is handled during the construction of the *Reuse* DAG for each resource. Resources are placed in one of two categories based upon their usage characteristics.

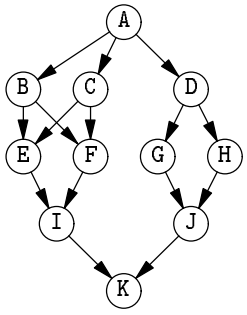
DEFINITION 1 *A resource R is a* **non-spanning** *resource if it is in use only during the execution of a single instruction.*

DEFINITION 2 *A resource R is a* **spanning** *resource if a use can start during the execution of one instruction and the use continues until the execution of a subsequent instruction. The beginning and ending instructions are called the* **defining** *and* **killing** *instructions, respectively.*

Since a functional unit is in use only while an instruction is being executed, it is a non-spanning resource. A register is in use from the time that one instruction, the defining instruction,

```
A: load v
B: w = v * 2
C: x = v * 3
D: y = v + 5
E: t1 = w + x
F: t2 = w * x
G: t3 = y * 2
H: t4 = y / 3
I: t5 = t1 / t2
J: t6 = t3 + t4
K: z = t5 + t6
```

(a) 3 address code



(b) DAG

Figure 4.1: Example code and corresponding DAG

places a value in the register until all instructions that need the value have used it from the register. The last instruction to read the value is the killing instruction.

Several different types of resources may belong to the same resource class. For example, an architecture may have both integer and floating point functional units. In this case there are two functional unit resource types, integer and floating point. Each register bank in an architecture is also a separate resource type.

The first kind of resource requirements information that is computed is the maximum resource requirements for each block. The maximum requirements for a resource, in a given block of a program, is the maximum amount of that resource required under any feasible schedule. It should be noted that a single schedule may not realize the maximum requirements for all resources, but instead different schedules may achieve maximum resource requirements for different resources. Thus the maximum resource requirements represent a worst case scenario.

The algorithm for obtaining the measurements of resource requirements operates on a DAG representing a partial order describing the dependencies. A chain in a partial order is a subset of elements such that every pair of elements in the subset are related. Every path in a DAG is a chain in the corresponding partial order, but a chain is not necessarily a path since it may be noncontiguous. Figure 4.1(a) shows a basic block of code and Figure 4.1(b) shows the corresponding DAG. In Figure 4.1(b), the sets of nodes {A, B, F, K}, {C, E, I}, {D, G, J}, and {H} are all chains.

DEFINITION 3 *Let relation Q be a partial order on set S*[2]*. A* **chain** *is a set $S' \subset S$ such that if $a, b \in S'$ then either $(a, b) \in Q$ or $(b, a) \in Q$.*

[2] *A relation Q on a set S is a* **partial ordering** *if and only if Q is reflexive, transitive, and anti-symmetric.*

DEFINITION 4 *A* **decomposition** *of a partial order P is a partition of P into chains. A decomposition is* **minimal** *if there is no other decomposition with fewer chains.*

If two nodes are independent, then they may be executed in parallel. The following theorem relates the maximum amount of parallelism to a minimal chain decomposition.

THEOREM 1 *The maximum number of independent elements in a partial order is equal to the number of chains in a minimal decomposition[Dil50].*

The DAG in Figure 4.1(b) can be minimally decomposed into a set of four chains, such as {A, B, E, I, K}, {C, F}, {D, G, J}, and {H}. Thus, at most four nodes at a time can execute in parallel.

If the resources needed can be represented as a partial ordering on the instructions, the task of computing maximum resource requirements can be performed using Theorem 1. The partial ordering on the nodes of a DAG, with respect to resource R, is defined as follows:

DEFINITION 5 *Let $CanReuse_R$ be a relation on nodes of the DAG for resource type R indicating if a resource instance r of type R used by a node can be reused by one of its descendants, i.e., $(a,b) \in CanReuse_R$ if and only if there is a node c that ends a's use of r and $c \in Ancestors(b) \cup \{b\}$.*

In other words, given that (a,b) belongs to the relation $CanReuse_R$, there is no schedule such that node b can execute while resource instance r is still in use as a result of executing a. The computation of the $CanReuse_R$ relation is different for spanning and non-spanning resources.

DEFINITION 6 *$Reuse_R$ DAG $(N, \bar{E})$ for resource type R is constructed from a program DAG (N, E). All edges $(a,b) \in \bar{E}$ must meet the following two conditions:*

1. *$(a,b) \in CanReuse_R$, and*
2. *$\nexists c \ni (a,c) \in CanReuse_R$ and $(c,b) \in CanReuse_R$.*

The second condition simply eliminates transitive edges from the $Reuse_R$ DAG. Although this condition is not necessary, it simplifies later discussions and techniques. $Reuse_R$ DAGs for functional units and registers are denoted as $Reuse_{FU}$ DAG and $Reuse_{Reg}$ DAG, respectively. The notation $Reuse_R$ DAG is used when a reference is not restricted to a particular resource. The DAG in Figure 4.1(b) is both a program DAG and a $Reuse_{FU}$ DAG.

DEFINITION 7 *An* **allocation chain** *for resource R is a chain $n_1, n_2, ...n_l$ such that $(n_i, n_{i+1}) \in Reuse_R DAG$ for any consecutive members n_i, n_{i+1} in the chain.*

```
function MeasureRequirements( Reuse_R DAG (N, E) )
   returns set of allocation chains
{
      /* build the bipartite graph */
   foreach n ∈ N
      add nodes s_n and t_n to N̂;
   foreach pair of nodes n,m ∈ N × N
      if ( m ∈ Ancestors(n) )
         add the edge (s_m, t_n) to Ê;

      /* find the maximum matching */
   M = BipartiteMatch(N̂, Ê)

      /* record the allocation chains in AC */
   numChains = 0;
   foreach n ∈ N such that t_n is not matched
   {  numChains = numChains + 1;
      i = n;
      add i to AC[numChains];
      while ( s_i is matched )
      {  i = j, where s_i is matched to t_j;
         add i to AC[numChains];
      }
   }
   return AC
}
```

Figure 4.2: Function *measureRequirements()*

After a $Reuse_R$ DAG has been decomposed into allocation chains each allocation chain can be assigned a different copy of the resource. However, if there are insufficient resources, these chains provide a measure of the resource requirements. Clearly, all chains in a $Reuse_R$ DAG are allocation chains. Therefore, by Theorem 1, a minimum decomposition of a $Reuse_R$ DAG into allocation chains gives the maximum resource requirements of R for the original DAG.

Ford and Fulkerson [FF65] have shown that the problem of finding a minimum chain decomposition can be solved by transforming it into a maximum bipartite graph matching problem. The bipartite graph represents all possible pairs of nodes $(a, b) \in CanReuse_R$. Since each node in the $Reuse_R$ DAG must participate in exactly one chain, a maximum matching finds a minimum number of allocation chains.

Figure 4.2 gives the Function *measureRequirements()*, which computes and returns the set of allocation chains. The algorithm first builds the bipartite graph from the $Reuse_R$ DAG by adding a node to each partition corresponding to each node in the DAG and then by adding an edge between each pair of nodes if the sink node can reuse the source node's resource. Next, the bipartite matching algorithm is applied. Finally, the allocation chains are constructed by traversing the matching edges.

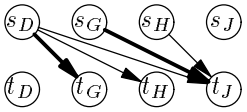


Figure 4.3: A bipartite graph matching

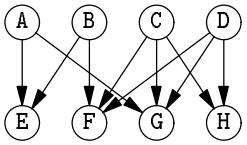


Figure 4.4: A complex case for defining *Kill()*

As an example, consider the subDAG {D, G, H, J} from the DAG in Figure 4.1(b). The corresponding bipartite graph is shown in Figure 4.3, with the bold arrows indicating a maximal matching. Since t_D is not matched, D is the head of a chain. The edge (s_D, t_G) indicates that the next node in the chain is G. Likewise, G is followed by J, which is the tail of the chain since s_J is not matched. Since neither t_H or s_H is matched, node H is the only node in a second chain. Thus, the two allocation chains are {D, G, J} and {H}.

4.1.2 Measurement specifics for functional units and registers

The definition of $CanReuse_R$ differs for spanning and non-spanning resources to reflect the different usage characteristics. Consider the non-spanning usage of a functional unit. A functional unit is only in use while an instruction is being executed; once the instruction has been executed the functional unit is available for reuse. In non-pipelined architectures if instruction b is dependent on instruction a, then b cannot begin execution until a's execution has been completed. Therefore, $CanReuse_{FU}$ is the partial order represented by the program dependence DAG, and the computation of the functional unit requirements and excess sets can be performed in polynomial time.

On the other hand, consider a spanning resource such as a register. A register is used to hold a value from the time that the defining instruction executes until the value is killed by the last instruction that uses it. Therefore, the definition of $CanReuse_{Reg}$ requires that the killing instruction be identified for each value defined, *i.e.,* the last use instruction to execute. However, URSA does not assume a specific schedule. Since the purpose of the resource requirements computations is to find the worst case scenario, the use instruction that would maximize the number of registers required is selected to be the killing instruction. Let $Kill(a)$ be the function that returns the node selected to kill node a's value. Then

$$CanReuse_{Reg} = \{(a,b) | b = Kill(a) \text{ or } b \in descendents(Kill(a))\} \tag{4.1}$$

In many cases the definition of *Kill()* is straightforward. However Figure 4.4 shows a case where defining *Kill()* is NP-complete. In this case all combinations of the nodes in the lower partition must be examined to find the smallest set that kills all of the nodes in the upper partition. In these cases the values of a set of nodes can be alive at the same time as a number of their dependents. *Kill()* must be defined to maximize the number of dependents that can be alive at the same time as their ancestors. This is accomplished by finding the minimum sized set of descendants that kills all of their ancestors.

THEOREM 2 *Defining Kill() for all nodes in the DAG is NP-Complete.*

PROOF: By reduction to the Minimum Cover problem, given in Appendix A. ■

Thus, a precise solution cannot always be expected when scheduling for parallel architectures or when code reordering is considered. However, it is useful to note that polynomial time algorithms exist when the maximum number of inputs is limited to two [Kar72]. Thus, for architectures whose instruction sets have no more than two physical inputs, precise maximum register requirements can always be computed.

The definition of *Kill()* can be broken into cases based on subDAGs. In several cases *Kill()* can be defined in linear time. In the remaining cases the problem of defining *Kill()* for the sub-DAG is equivalent to a minimum cover problem. A heuristic for the minimum cover problem, based on the greedy algorithm, can then be applied to each identified sub-DAG.

The sub-DAG {B, C, E, F} of the DAG in Figure 4.1(b) is an example of the difficult case. An optimal solution to the minimum cover problem for this sub-DAG will choose the same node to kill both B and C. Let the solution be F. Then, $Kill(\mathtt{B}) = Kill(\mathtt{C}) = \mathtt{F}$, so $(\mathtt{B},\mathtt{F}) \in CanReuse_{Reg}$, $(\mathtt{C},\mathtt{F}) \in CanReuse_{Reg}$, $(\mathtt{B},\mathtt{E}) \notin CanReuse_{Reg}$, and $(\mathtt{C},\mathtt{E}) \notin CanReuse_{Reg}$. Thus, three allocation chains are required to decompose this sub-DAG.

The impact of imprecise solutions to the minimum cover problems must be considered. If the cover found is not minimum, then fewer allocation chains are found than should be. Thus, the register requirements may be underestimated. An upper bound on the imprecision can be computed by finding the minimum number of nodes in T whose sum of edges from nodes in S is equal to or greater than the number of nodes in N. Since the requirements can be underestimated, the reduction of requirements may not produce a DAG whose requirements can be met by the target machine. Any remaining excess register requirements will be identified during the assignment phase. The assignment phase can use simple on-the-fly heuristics to resolve the conflicts. Due to the nature of the problem, the sub-DAG, and the minimum cover heuristic it is not expected that many conflicts will

be left unresolved for the assignment phase. Results in chapter 12 show that only simple heuristics were needed to compute precise measurements in all cases encountered in the benchmark programs.

4.2 Excessive Sets

The second step in measuring resource requirements is to identify all locations where a program needs more copies of a resource than are available. The blocks with excessive resource requirements contain sets of instructions that if executed concurrently would require more resources than are available. Each such set of instructions is called an *excessive set*. The excessive set information is used to determine which blocks must have their resource requirements reduced. URSA's transformations do not require enumeration of all excessive sets, but only the sets of allocation subchains that are independent of each other and whose size exceeds the number of resources available.

DEFINITION 8 *An* **excessive set** ES_R for resource type R is a set of instructions $\{n_1, n_2, \ldots, n_m\}$ which has the following properties.

1. $\forall n_i, n_j \in ES_R$, n_i and n_j are independent,
2. $m \geq |R|$, where $|R|$ is the number of copies of resource R available.

The enumeration of all excessive sets of instructions that occur in a block is a NP-Complete problem by virtue of the fact that there can be an exponential number of such sets. In practice, all that is needed is the set of all instructions that belong to at least one excessive set.

DEFINITION 9 *A* **summary excessive set** SES_R *for resource* R *is the union of all excessive sets* ES_R *in a given block.*

In practice, summary excessive sets are computed using a working list technique. Each instruction in the working list is added to the summary excessive set if it is independent of at least $|R|$ instructions on separate allocation chains for R. All unexamined instructions that are independent of the excessive instruction are then added to the working list. The initial instruction of the list is located by scanning all instructions until one that meets the excessive test is found. This growing process is graph linear in time.

Consider the minimal decomposition {{A, B, E, I, K}, {C, F}, {D, G, J}, {H}}, and assume that there are three functional units available. The instruction B is the first instruction that is independent of at least three other instructions, C, G, and H. Thus, B is added to the working list and the summary excessive set. The instructions C, D, G, H, and J are then added to the working list. Of

these instructions C, G, and H will be added to the summary excessive set while D and J will not. The final summary excessive set is {B, C, E, F, G, H}.

4.3 Resource Holes

The final piece of resource usage information that is computed are *resource holes*. These are locations in the program where a copy of a resource is under utilized and available for allocation to additional instructions. This section discusses how resource holes are found and identifies properties indicating how they can be allocated to additional instructions.

DEFINITION 10 *A* **resource hole** *is a location on an allocation chain where that resource is available for additional allocations. A resource hole, h, has the following properties:*

1. *The* **type** *of the hole indicates how the hole must be used.*
2. *The* **size** *of the hole, $size_h$, is the number of cycles that the hole is available for addition allocations.*
3. *The* **earliest available time**, $\mathtt{EAT}_h$, is the earliest time that the resource can be allocated to another instruction.
4. The **latest available time**, $\mathtt{LAT}_h$ is the latest time that the resource can be allocated to another instruction.

Resource holes and their properties are located by analyzing the allocation chains for the resource of interest. The properties of a hole are determined by the time of execution of the instructions surrounding it. The scheduling of instruction i in the program DAG is limited by the precedence constraints to a time frame in which it can execute. The time frame is delimited by the instruction's earliest start time, EST_i, and latest finish time, LFT_i. Let τ_i denote the execution time for instruction i. Then i's latest start time, LST_i, is given by $LST_i = LFT_i - \tau_i$. The *slack time* for scheduling instruction i is given by $slack_i = LST_i - EST_i$. The identification of resource holes is performed by examining the instructions' ESTs and LFTs on each allocation chain and recording the information for each hole.

Resource holes can occur in two different situations. The first, a free hole, occurs when an instance of a resource is unused in a section of a basic block. Free holes can occur because no instructions from an allocation chain can execute in this section of code. They can also occur at the beginning or end of a block before maximum demands are encountered.

type	size	EAT	LAT
Free	$\texttt{LST}_{n_2} - \texttt{EFT}_{n_1}$	$\texttt{EFT}_{n_1}$	$\texttt{LST}_{n_2}$
Slack	$slack$	$\texttt{EST}_{sn_1}$	$\texttt{LFT}_{sn_l}$

Table 4.1: Computation of hole properties

The second type of hole, a slack hole, occurs when resources that are already allocated may be temporally shared. If $slack_i = 0$ then i is on a critical path and has no flexibility for scheduling. If i is not on a critical path then there is some flexibility on when it can be scheduled. Thus, its resources may be available for allocation to another instruction.

DEFINITION 11 *If two consecutive instructions, i_j and i_{j+1}, on an allocation chain cannot be executed consecutively, i.e., $LFT_{i_j} < EST_{i_{j+1}}$, then there is a* **free hole**, *h, such that $EAT_h = LFT_{i_j}$, $LAT_h = EST_{i_{j+1}}$, and $size_h = LAT_h - EAT_h$.*

As an example, consider the basic block of code in Figure 4.5(a) and the corresponding DAG in Figure 4.5(b). Assume that the functional unit allocation chains are {B}, {A, C, E, G, H, I, K, L}, and {D, F, J}, and that the register allocation chains are {A, B}, {C, F, G, H, I, K, L}, {D}, and {E, J}, then Figures 4.5(c) and 4.5(d) show partial schedules for the resources, where each column represents one allocation chain. Thus the DAG requires three functional units and four registers to exploit all available parallelism. Assuming that all instructions require unit time, the DAG requires eight time units to execute. A free functional unit hole exists between instructions F and J, with size 2 and range (3, 4). Thus, two instructions could be allocated to that allocation chain between F and J. Another free functional unit hole exists between J and the end of the block.

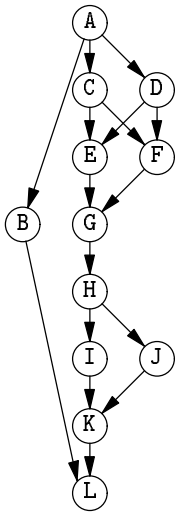
DEFINITION 12 *If there is a set of consecutive instructions $\mathcal{I} = \{i_1, i_2, ..., i_n\}$ and a constant s such that $\underset{i_j \in \mathcal{I}}{\forall} slack_{i_j} = s$, then there is a* **slack hole** *h, such that $EAT_h = EST_{i_1}$, $LAT_h = LFT_{i_n}$, and $size_h = s$.*

In Figure 4.5(d) there is a slack functional unit hole involving instructions A, B, and the end of the block. B has a slack time of 5, and a range of (1, 7). Thus, five instructions could be allocated to B's allocation chain. Any number of these five instructions can be allocated between A and B, with the remainder after B.

The computation of both the size and the availability of a hole is summarized in Table 4.1. Nodes n_1 and n_2 surround a free hole, and nodes sn_1 and sn_l are the first and last nodes in a slack hole. LST and EFT are the latest start time and earliest finish time of a node, respectively. *Slack* is

```
A: load a
B: b = 2 * a
C: c = a + 1
D: d = a - 3
E: e = c * d
F: f = c - d
G: g = e / f
H: h = g + 5
I: i = h * 2
J: j = h + 4
K: k = i / j
L: l = b + k
```

(a) Basic block of code



(b) Corresponding DAG

	A	
B	C	D
	E	F
	G	
	H	
	I	J
	K	
	L	

(c) Partial schedule of functional units

A			
B	C	D	
	F		E
	G		
	H		
	I		J
	K		
	L		

(d) Partial schedule of registers

Figure 4.5: Example DAG of a basic block

the slack time of each node in the hole. The hole nodes are annotated with these characteristics. The computation of the `LST` and `EFT` for the instruction and region nodes is graph linear in time. The location of the holes requires $O(N)$ time, and the worst case number of holes found is $2N$, where N is the number of instruction and region nodes in the region.

When Resource Spackling is performed it is preferable to find a resource hole large enough to hold the instruction or instructions being placed in it. However, it is possible that no such holes exist. In such cases the instructions are inserted in the most desirable hole despite its size, with the result that the length of the critical path through the DAG is increased. This technique is referred to as *wedged insertion*. In practice it is convent to place zero sized holes between all instructions where there would otherwise be no hole due to the instructions being on a critical path to facilitate wedged insertion.

Chapter 5

Global Unified Resource Requirements Representation

In this chapter an intermediate representation that incorporates the resource requirements information described in Chapter 4 is presented. The incorporation of the resource requirements enables the compiler back end to fully integrate phases that allocate and schedule different types of resources, such as registers and functional units. A single representation simplifies such integration by presenting all information in a consistent manner, including where and how all resources of interest are used and available, as well as factors that affect the execution time of the program, such as critical path lengths and execution counts of regions.

5.1 Integrated Resource Allocation Representation

Algorithms that integrate resource allocation need resource usage information to make effective allocation decisions that have a minimal impact on the execution time of the program. Resource allocation decisions only need to be made when there are locations in a program segment that require more instances of a resource than are available. Advanced resource allocation algorithms, such as those based on the Measure and Reduce paradigm and to some extent global schedulers, move instructions from locations where there are insufficient instances of a resource to locations where extra resource instances are available. Thus, the resource usage information must indicate all locations where resources are either over utilized or under utilized. Since the architectures targeted in this work exploit ILP, the representation used must take into account the ability to schedule instructions in parallel. This chapter identifies a set of properties for intermediate representations and shows how the work discussed in the previous chapter can be used in an intermediate representation to satisfy these properties.

5.1.1 Integrated Resource Allocation Properties

An intermediate representation should satisfy the following properties to support unified resource allocation. These properties are called the *Unified Representation Properties.*

PROPERTY 1 *[Integrated Representation] The representation can be used to determine the impact of a resource allocation or set of resource allocations on all resource demands in all segments and the execution time of the program.*

PROPERTY 2 *[Measurability] The representation enables measurement of all segments' demands for all resources. A resource measurement is* **precise** *if it indicates the minimum number of copies of the resource needed to exploit all parallelism uncovered in each program segment.*

PROPERTY 3 *[Resource Usage] The representation identifies all locations in each segment where resources are either over utilized or under utilized.*

PROPERTY 4 *[Executability] The representation indicates if each program segment in its current state can be executed using the available resources. A program segment is* **executable** *if and only if for each resource the number of copies required is less than or equal to the number of copies available. A program is executable if and only if all its segments are executable.*

Ideally, the representation should supply precise resource measurements. However, as discussed in Chapter 4 and Appendix A, the problem is NP-Complete for spanning resources. Thus, there is a trade-off between the precision of the measurements and the time taken to compute them. Appendix A also discusses fast heuristics developed for measuring spanning resource requirements that are demonstrably precise.

The measurability of the representation allows resource usage information for all resources to be computed. An intermediate representation that provides resource usage information for all resources enables unified resource allocation.

5.1.2 GURRR

GURRR is an intermediate representation that meets the unified representation properties and is used to investigate unified resource allocation algorithms. To support a variety of parallelization techniques, including powerful code motions, GURRR is based on a modified form of the Program Dependence Graph (PDG).

The Instruction Program Dependence Graph is used to represent instruction level parallelism not explicitly expressed in the traditional statement level PDG. Since a single program statement may result in several intermediate code statements, representing the program at the intermediate code level permits access to more ILP. To support a wider range of code motions the representation is converted to Static Single Assignment form (SSA) [RWZ88, AWZ88]. Special instruction nodes can be added to carry loop and array access information to enable the exploitation of medium grain parallelism as well. Compilers using PDG based representations perform resource allocations on a region by region basis. Thus, regions correspond to the program segments mentioned in the unified representation properties.

The *Instruction PDG* (IPDG) is a graph $G = (N, E)$, in which the set of nodes, N, is a union of the following node types.

1. Instruction nodes, $\mathcal{I}$, are similar to statement nodes found in traditional PDGs, but represent intermediate opcodes.
2. Region nodes, $\mathcal{R}$, in the PDG identify a unique set of execution conditions or control dependencies.

The set of edges, E, is a union of the following edges types.

1. Control dependence edges, $\mathcal{C} \subset \{\mathcal{I} \times \mathcal{R}\} \cup \{\mathcal{R} \times \mathcal{I}\}$, connect the region node to the instruction and subregion nodes that execute under the conditions that it identifies. Control edges are also added from the instruction nodes specifying those conditions to the region node.
2. Data dependence edges, $\mathcal{D} \subset \{\mathcal{I} \cup \mathcal{R}\} \times \{\mathcal{I} \cup \mathcal{R}\}$, connect the instruction nodes and represent the dependence of the instruction nodes on data values computed by earlier instruction nodes. In addition, data dependence edges are added from the instruction nodes defining values to the region nodes containing uses of the values to summarize the dependence of the region as a whole on data values computed by earlier instructions.
3. Transitive data dependence edges, $\mathcal{D}_T \subset \{\mathcal{I} \cup \mathcal{R}\} \times \{\mathcal{I} \cup \mathcal{R}\}$, indicate indirect dependencies between nodes due to a sequence of data dependencies. The addition of these edges simplifies the computation of the ordering of nodes within a region.

GURRR extends the IPDG to include the resource usage information required to meet the unified representation properties. In addition to summarizing control dependence information, region nodes in GURRR are used to summarize resource usage information. Since the regions are organized in a hierarchical manner, the summary for a region must include resource usage information

for both the instructions and subregions that it contains. Regions in GURRR also store execution counts, indicating how many times the region is expected execute during a run of the program. This information enables the allocation algorithms to make better decisions on how many resources to allocate in each region.

The Measure and Reduce allocation scheme presented in this dissertation explicitly decides which instructions should be delayed until all resources that it requires are available. To support these scheduling decisions GURRR must represent additional constraints placed on the ordering of nodes. GURRR must also contain information used to measure the resource requirements. A part of this information is identifying which instructions can share an instance of a resource. The following additions are made to the IPDG's nodes and edges to meet these requirements and obtain GURRR.

1. Resource hole nodes, $\mathcal{H}$, represent the resource holes found on the allocation chains. Each hole node is annotated with the resource availability characteristics.
2. Temporal dependence edges, $\mathcal{T} \subset N \times N$, are used to represent sequential dependencies. These edges are used to supply additional ordering constraints on the nodes, such as placement of hole nodes, and instruction and region node scheduling.
3. Reuse edges, $\mathcal{U} \subset \{\mathcal{I} \cup \mathcal{R}\} \times \{\mathcal{I} \cup \mathcal{R}\}$, connect nodes that can temporally share an instance of a resource under any schedule allowed by all of the dependencies in a region. A separate set of reuse edges is used for each resource type.

The stipulation that Reuse edges are added only when a resource can be shared under all semantically correct schedules allows for parallel execution and code reordering. An allocation algorithm, such as Measure and Reduce, can select any allowable schedule and determine the worst case resource requirements. When only a single schedule, such as the original order of the sequential source code, is used, the resource requirements measurements are less precise, since the schedule does not account for as many overlaps of uses of resources.

Figures 5.1(a) and 5.1(b) show a simple program and the corresponding GURRR. The target architecture has three functional units and three registers. Control, data, and temporal dependencies, and reuse edges are indicated by bold, normal, dashed, and dotted lines, respectively. To improve readability only the reuse edges for registers are displayed.

To be useful for unified resource allocation, GURRR must satisfy the unified representation properties. GURRR satisfies the Measurability property by using the reuse edges and allocation chains to compute each region's requirements for all resources. As discussed in section 5.2.1, the resource measurements are precise for functional units and usually precise for registers. GURRR

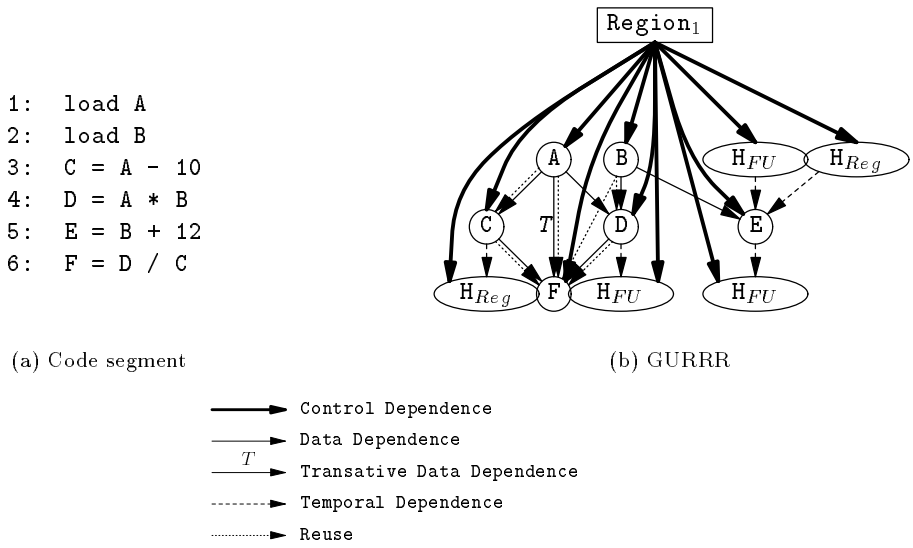


Figure 5.1: Example of GURRR

provides resource usage information for all resources on the IPDG. All types of dependencies are represented by the various types of edges in GURRR, allowing the execution time of a region to be computed. The combination of all of this information on a region by region basis and in hierarchical summaries satisfies the Unified Representation property. In each region the excessive sets and resource hole nodes identify all locations that over utilize and under utilize resources, respectively, satisfying the Resource Usage property. Finally, the number of instances of a resource required by a region is stored in the region node. This number can be compared to the number of instances of the resource available for allocation to the region to determine if the region is executable. Due to the hierarchical nature of GURRR, the program is executable if the root region is executable, satisfying the Executability property.

At times during the measurement of resource requirements and use of GURRR by the compiler back end, it is convenient to consider only subsets of the information provided by GURRR. Four combinations of subsets of nodes and edges commonly used are identified. Each combination is a subgraph composed of selected subsets of nodes and edges.

DEFINITION 13 *Given a graph $G = (N, E)$, the subgraph of G* **induced** *by $N' \subset N$* **with respect to** *$\hat{E} \subset E$ is the graph $G' = (N', E')$, where $E' = \{(u, v) \in \hat{E} : u, v \in N'\}$*

1. The **Control Dependence Graph**, CDG, is the graph induced by $\mathcal{I} \cup \mathcal{R}$ with respect to $\mathcal{C}$.
2. The **Data Dependence Graph**, DDG, is the graph induced by $\mathcal{I} \cup \mathcal{R}$ with respect to $\mathcal{D}$.
3. The **Region DAG** for a region R, $Region_{\mathtt{R}}$ DAG, is the graph induced by $\{n | n \in \mathcal{I} \cup \mathcal{R} \cup$

$\mathcal{H}$ and $(\mathtt{R}, n) \in \mathcal{C}\}$ with respect to $\mathcal{D} \cup \mathcal{D}_T \cup \mathcal{T}$. This graph provides all of the information needed to allocate all resources in the region.

4. The **Reuse DAG** for a region R and resource R, $Reuse_R$ DAG, is the graph induced by $\{n | n \in \mathcal{I} \cup \mathcal{R}$ and $(\mathtt{R}, n) \in \mathcal{C}\}$ with respect to $\mathcal{U}$.

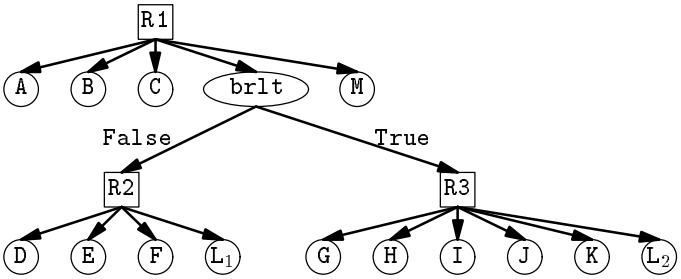
The CDG and DDG are the same as those found in the IPDG. The Region DAG contains all dependence and resource usage information required for performing local unified resource allocation. The CDG and Region DAGs for other regions may be used when performing various types of integrated global resource allocations. The *Reuse* DAG is typically used only by the resource usage computation algorithms. These algorithms measure the resource requirements, compute the excessive sets, and add the resource hole nodes.

As an example of the various subgraphs, consider the code segment of an `if-then` statement in Figure 5.2(a) and assume that the target architecture has a single type of functional unit resource and a single type of register resource. In the subsequent figures, edges representing redundant ordering information are removed to aid readability. The control and data dependence subgraphs are shown in Figures 5.2(b) and 5.2(c) respectively. The functional unit and register *Reuse* DAGs are shown in Figures 5.3(a) and 5.3(b) respectively. The region 2 node, `R2`, does not occur in the functional unit *Reuse* DAG since its instructions are not executed in parallel with region 1's instructions. The `R2` node occurs in the register *Reuse* DAG since the values it computes can be alive simultaneously with some of the values computed in region 1. Since the two values $\mathtt{D}_1$ and $\mathtt{D}_2$ share a register, the `R2` node represents the register demand of instruction `t`. The `brlt` predicate node does not occur in the register *Reuse* DAG since it does not write to a register. The functional unit *Reuse* DAG for region 1 can be covered by the four allocation chains {`C, brlt`}, {`A,` $\mathtt{D}_1$`, F, H`}, {`B, E`}, and {`G`}, indicating a maximum requirement of four functional units to exploit all parallelism in the region. The register *Reuse* DAG can be covered by the six allocation chains {`C, F`}, {`A,` $\mathtt{D}_1$`, H`}, {`B`}, {`E`}, {`R2`}, and {`G`}, indicating that it is possible for six values to be simultaneously alive.

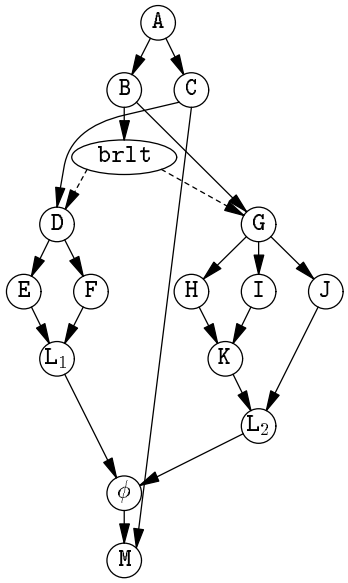
Figure 5.3(c) shows the partial schedule for functional units in region 1 imposed by the data and temporal dependencies. Each column represents an allocation chain. There are free resource holes before `C`, after both `brlt` and `E`, and before and after `G`. Instructions $\mathtt{D}_1$, `E`, and `G` have slack time in when they can be scheduled. Since the functional units are not needed for the entire time, these nodes exist in slack holes. Figure 5.3(d) shows the region DAG for region 1 with only the functional unit hole nodes. Free and slack hole nodes are marked with `FH` and `SH` respectively. A transitive data dependence edge has been added from node `C` to node `F` to indicate the transitive dependence caused

```
 1:  load A
 2:  load B
 3:  C = A * 4
 4:  D1 = A * B
 5:  E = B + 3
 6:  brlt C, 9
 7:  t = D1 + C
 8:  D2 = t * 5
 9:  F = D / E
10:  G = B + 10
11:  H = F + G
```

(a) Code segment

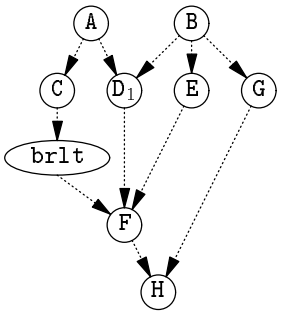


(b) Control Dependence Subgraph

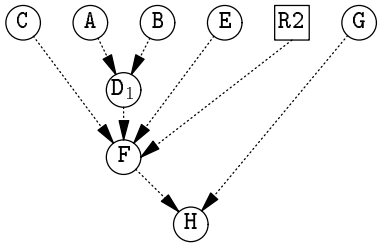


(c) Data Dependence Subgraph

Figure 5.2: Sample Code and GURRR Dependence Subgraphs



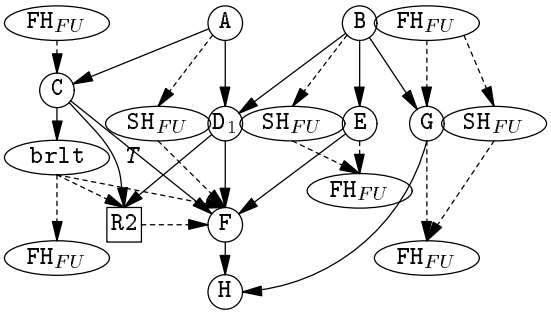
(a) Region 1 Functional Unit Reuse DAG



(b) Region 1 Register Reuse DAG

	A	B	
C	D1	E	G
brlt			
	F		
	H		

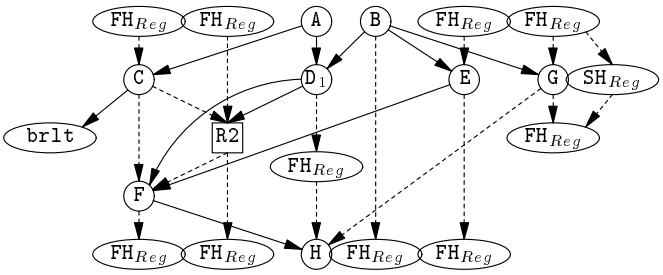
(c) Region 1 FU Schedule



(d) Region 1 DAG with FU Holes

	A	B			
C	D1		E	R2	G
F					
	H				

(e) Region 1 Register Schedule



(f) Region 1 DAG with Register Holes

Figure 5.3: GURRR Resource Usage Information

by nodes `t` and $\texttt{D}_2$, which are not in region 1. The largest set of instructions that can be executed in parallel is {`C`, $\texttt{D}_1$, `E`, `G`}, which would be excessive if the target architecture provided fewer than four functional units.

The partial schedule for registers is shown in Figure 5.3(e). The allocation chain containing `R2` does not have any instructions from region 1 and consists of two free holes separated by the node `R2`. Figure 5.3(f) shows the region 1 DAG without the functional unit holes.

5.2 Computing GURRR

GURRR is initially constructed and used as the input form for the integrated phases in the back end of the compiler. However, as the phases operate they transform the program. These transformations must be made in GURRR. Furthermore, The resource usage information must be updated to reflect the impact of the transformations. This section discusses both the initial construction and incremental updating of GURRR.

5.2.1 Construction of GURRR

The construction of GURRR begins with an IPDG and is performed in a hierarchical manner on the DAG of region nodes resulting from the forward control dependencies. The regions are visited one at a time in a bottom up order and the local components are constructed. A summary of the resource requirements of subregions is used during the construction in the parent region. The resulting global resource requirements are contained in the root region.

Special processing occurs when there are mutually exclusive subregions, such as the `then` and `else` subregions of an `if` statement. In this case, the region containing the `if` statement is only concerned with the maximum requirements of the set of mutually exclusive subregions. The subregions nodes are marked as mutually exclusive and the construction takes the maximum of the requirements for each resource.

The steps in the construction of GURRR for each region are performed as follows.

Add transitive data dependence edges: Transitive data dependence edges are added between all instruction and region nodes. The computation of the transitive data dependence edges can be done in graph linear time. In the worst case $O(N^2)$ edges are added. These edges are required for the proper computation of the *Reuse* DAGs.

Build *Reuse* DAGs: The $Reuse_R$ DAG is the instantiation of the relation $CanReuse_R$ for resource R. The $Reuse_R$ DAG is constructed by adding an edge from node a to node b for each

$(a, b) \in CanReuse_R$, where both a and b use resource R.

The sets of nodes whose resource can be reused by node n are computed in a forward topological traversal of the DAG using the equation

$$CanReuse_R[n] = avail[n] \bigcup_{P \in predecessors(n)} CanReuse_R[P].$$

$Avail[n]$ is at most all of n's immediate predecessors whose instances of R can be safely reused by n. The computation of $avail[n]$ is dependent on whether the resource is classified as spanning or non-spanning. For non-spanning resources, $avail[n]$ is the set of n's closest ancestors that use R. Computing $avail[n]$ for spanning resources requires a special component analysis. The identification and analysis of most components can be performed in graph linear time. However, for a few components the analysis is NP-Complete. The computations of $avail[n]$ and $CanReuse$ are graph linear and the resulting $Reuse$ DAGs contain $O(N^2)$ reuse edges.

Find allocation chains: The number of allocation chains for each resource is recorded in the parent region's node. Once the requirements have been measured, the allocation chains are used to compute excessive sets and resource holes.

Find excessive sets: The summary excessive sets are stored in the region node for use by the allocation phase. The process of growing excessive sets is graph linear in time.

Find resource holes and add hole nodes: Free hole nodes are added between the consecutive nodes surrounding the hole. Slack hole nodes are added between the predecessor of the first instruction or region node in the hole and the successor of the last node in the hole.

5.2.2 Incremental Updating of GURRR

GURRR is able to reflect changes in resource requirements resulting from the transformations applied to the program. The brute force approach is to recompute all information from scratch after each transformation is applied. This can be a costly approach, and it does not provide any support for predicting the impact of a transformation. It would be useful to be able to estimate the impact of a possible transformation on the resource requirements. This section discusses techniques for incrementally updating GURRR.

In previous work on specifying transformations a basic set of program edits to describe the transformations has been used [WS91, Dow94]. The following set of *Standard Edit Functions* (SEFs) is defined, which apply a transformation to the elements of a PDG.

`Add`*element* Create a new element

`Delete`*element* Delete an element

`Copy`*element* Create a new element and copy label information of an existing element

`Move`*element* Delete and recreate an element, preserving label information

`Modify`*element* Change the label information

There are two sets of the above operations, one for nodes and one for edges, giving a total of ten SEFs. Since nodes will never be added without corresponding edges, and the edge SEFs can be viewed as combinations of adding and deleting edges, only the `AddEdge` and `DeleteEdge` SEFs are considered.

The computation of the $CanReuse$ relation is graph linear. The updating of $avail[n]$ information is limited to the nodes affected by the `AddEdge` and `DeleteEdge` SEFs. The updated information is then propagated through the region DAG. The $Reuse$ DAG is updated by adding and deleting edges corresponding to the nodes inserted and removed in the $CanReuse$ relation.

The matching algorithm used to compute allocation chains is incremental in nature; each matching is a partial solution and new matchings are added by finding augmenting paths. Thus, the modified $Reuse$ DAG with edges deleted and added can be used as a partial solution. The complexity for this solution is $O(\sqrt{m}E)$, where m is the number of chains in the initial partial solution. An alternative approach can find only unit length augmenting paths in graph linear time, possibly introducing some imprecision.

Updating of the excessive sets is performed in two steps. First, the nodes in the existing excessive sets are tested to see if they are still in parallel with an excessive number of other nodes. This step can be limited to the nodes that have had edges added to them. Second, nodes not in the excessive sets are tested to see if they now should be added to the working set. The initial set of nodes considered in this step can be limited to those that have had edges removed.

Transformations can affect holes by creating new ones, removing existing ones, and by changing their characteristics. All of these changes can be found by examining each node whose `EST` and/or `LFT` has changed. However, the nature of the matching algorithm used to find the allocation chains can cause unchanged holes to migrate between allocation chains. The sequential edges used to place the hole nodes in the region DAGs can be updated to reflect the migrations in linear time in the number of hole nodes.

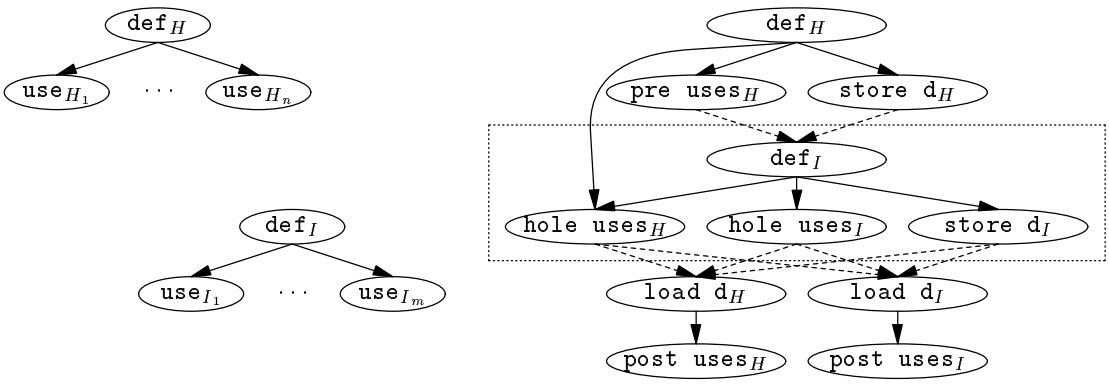
Chapter 6

Resource Spackling

In URSA's approach to the Measure and Reduce paradigm, instructions in excessive sets are allocated resources by placing them in resource holes. The placement is achieved by introducing additional temporal dependences between instructions. This process is called *Resource Spackling*. Spanning resources are more complex than non-spanning resources due the fact that spanning uses may have to be split, requiring insertion of store and load instructions. Since the mechanics of resource spackling differ for spanning and non-spanning resources, two different transformations are needed, spanning resource spackling and non-spanning resource spackling. When the type of transformation is clear from the context, the transformation is generically referred to as the *spackling transformation* in this dissertation.

Preconditions must be placed on the instruction and hole selected for the transformation to ensure that the program's semantics are preserved and that a reduction in resource requirements is achieved. In some situations the preconditions do not limit instructions to a single location. In these cases heuristics are used to select a specific location based on relative costs. Such heuristics are described in subsequent chapters.

This chapter presents the two spackling transformations as applied to GURRR, as well as the conditions required of the instruction and hole pair selected for transformation. It is shown that as long as there are excessive sets it is possible to find a hole into which an instruction from an excessive set can be spackled. Thus, all excessive resource requirements can always be removed. The spanning resource spackling transformation is presented first. The non-spanning resource spackling transformation is then shown to be a simplified form of the spanning transformation.



(a) Before (b) After

Figure 6.1: Filling a non-spanning resource hole

6.1 Filling Spanning Resource Holes

The spanning resource spackling transformation arranges for two spanning uses of a resource to temporally share a single instance of the resource in an interleaved manner. The transformation deals with two groups of nodes. The first group consists of the node at which the hole starts and all nodes that use its value. Together these nodes represent the spanning use defining the hole. The second group consists of the node selected for insertion into the hole and all nodes that use its value. Together these nodes represent the spanning use to be inserted. These nodes are combined into several useful subsets.

DEFINITION 14 *Let* `H` *be a spanning resource hole and* `I` *be the use to be inserted into* `H`*. The following sets of nodes are used during spackling spanning resource uses.*

1. *The hole* `H` *is delimited by the nodes where the spanning value is defined by the instruction* $\texttt{def}_H$*, and the end of the hole, node* $\texttt{H}_{end}$*.*
2. *The instruction* $\texttt{def}_I$ *defines the value* `I` *to be spackled into* `H`*.*
3. *The set* $\texttt{use}_H$*, consisting of nodes* $\texttt{use}_{H_1} \ldots \texttt{use}_{H_n}$*, is the set of all uses of* $\texttt{def}_H$*. In cases where* $\texttt{def}_H$*'s live range must be split, these nodes are partitioned into three sets,* $\texttt{preUses}_H$*,* $\texttt{holeUses}_H$*, and* $\texttt{postUses}_H$*.*
4. *The set* $\texttt{use}_I$*, consisting of nodes* $\texttt{use}_{I_1} \ldots \texttt{use}_{I_m}$*, is the set of all uses of* $\texttt{def}_I$*. In cases where* $\texttt{def}_I$*'s live range must be split, these nodes are also partitioned into two sets,* $\texttt{holeUses}_I$ *and* $\texttt{postUses}_I$*.*
5. *The 7-tuple* $\mathcal{T} = (\ \texttt{def}_I, \texttt{holeUses}_I, \texttt{postUses}_I, \texttt{def}_H, \texttt{preUses}_H, \texttt{holeUses}_H, \texttt{postUses}_H\ )$ *is referred to as a* **S-spackling tuple***.*

To achieve a reduction in resource requirements the spackling transformation places the node `I` into the hole `H`. Circumstances may require either or both of the live ranges to be split. When

$\texttt{def}_H$'s live range must be split its uses are partitioned into the sets $\texttt{preUses}_H$, $\texttt{holeUses}_H$, and $\texttt{postUses}_H$, which are placed prior to, in, and after the hole, respectively. When $\texttt{def}_I$'s live range must be split its uses are partitioned into the sets $\texttt{holeUses}_I$ and $\texttt{postUses}_I$, which are placed in and after the hole, respectively. Figure 6.1 shows the affected portion of the program graph before and after the transformation. The dashed box in Figure 6.1(b) represents the resource hole into which `I` is spackled.

Figure 6.1 depicts the various sets of nodes as being disjoint. However, the S-spackling transformation must handle cases when nodes exist in more than one set. These cases arise when either a node belongs to both live ranges, or $\texttt{def}_I$ uses the value defined by $\texttt{def}_H$.

CONDITION 1 *[Overlapping Sets] The following cases describe when two sets in a S-spackling tuple may overlap.*

1. $\texttt{def}_I \in \texttt{preUses}_H$
2. $\texttt{postUses}_I \cap \texttt{postUses}_H \neq \emptyset$
3. $\texttt{holeUses}_H \cap \texttt{holeUses}_I \neq \emptyset$

All other pairs of sets in the S-spackling tuple must be disjoint.

The procedure `spackleSpanning()`, shown in Figure 6.2, performs the actual modifications of GURRR depicted in Figure 6.1. For clarity, the code to handle the special cases mentioned in the Overlapping Sets Condition is omitted. The procedure `spackleSpanning()` performs a single spanning resource spackling transformation and is called from the driver procedure `reduceSpanning()`, shown in Figure 6.3. For each call to `spackleSpanning()` the driver first finds a pair of definitions for the transformation and creates the S-spackling tuple by splitting their uses. After calling the spackling transformation the driver updates the resource requirements information. This process is repeated as long as there is an excessive set for the spanning resource under consideration.

The remainder of this section is dedicated to proving two important properties of the algorithms given. The first property is that `spackleSpanning()` is a safe transformation, that is, it does not produce a representation of the program that violates the properties of GURRR and it preserves semantic correctness. The second important property of the algorithms is that `reduceSpanning()` will terminate.

Since GURRR is a graph based representation of dependences, several standard graph functions are useful in this discussion.

```
procedure spackleSpanning( defI, holeUsesI, postUsesI,
                           defH, preUsesH, holeUsesH, postUsesH )
{
      /* spill the inserted value if needed */
   if ( postUsesI != ∅ )
   {  storeI = newStore();
      loadI = newLoad();

      foreach n ∈ postUsesI
      {  delEdge( defI, n, Data );
1:       addEdge( loadI, n, Data );
      }

2:    addEdge( defI, storeI, Data );
   }
   else
      loadI = NULL;

      /* spill the spanning value if needed */
   if ( postUsesH != ∅ )
   {  storeH = newStore();
      loadH = newLoad();

      foreach n ∈ postUsesH
      {  delEdge( defH, n, Data );
3:       addEdge( loadH, n, Data );
      }

      if ( storeI )
4:       addEdge( storeI, loadH, Temporal );

5:    addEdge( defH, storeH, Data );
6:    addEdge( storeH, defI, Temporal );
   }
   else
      loadH = NULL;

      /* constrain hole uses into hole */
   foreach n ∈ holeUsesI ∪ holeUsesH
   {  if ( loadI )
7:       addEdge( n, loadI, Temporal );
      if ( loadH )
8:       addEdge( n, loadH, Temporal );
   }

      /* constrain pre uses to before hole */
   foreach n ∈ preUsesH
9:    addEdge( n, defI, Temporal );
}
```

Figure 6.2: Procedure *spackleSpanning()*

```
procedure reduceSpanning( block, resource )
{
   computeRequirments( block, resource );

   while ( block has excessive sets )
   {
      (defH, defI) = selectSpackle( block, resource );
      (holeUsesI, postUsesI, preUsesH, postUsesH) = splitUses( defH, defI );
      spackleSpanning( defI, holeUsesI, postUsesI, defH, preUsesH, holeUsesH postUsesH );

      updateRequirements( block, resource );
   }
}
```

Figure 6.3: Procedure *reduceSpanning()*

DEFINITION 15 *The following functions take two sets as parameters and return the status of dependences between them.*

1. `depends(A, B)` *returns true if and only if there is a direct or indirect dependence of any type from any node in the set* `A` *to any node in the set* `B`.
2. `noDeps(A, B)` *returns the negation of* `depends(A, B)`.

After all spackling transformations have been performed the resulting GURRR of the program is used to determine the order of the instructions during code emission. Thus, the GURRR for a region must be acyclic and `spackleSpanning()` must not introduce cycles. Existing dependences between the nodes of interest are used in the selection of $\texttt{def}_H$ as well as in the splitting of the uses of $\texttt{def}_H$ into their respective sets. Based on the temporal dependences added by the transformation preconditions are placed on the sets in the S-spackling tuple to ensure that no dependence cycles are introduced.

CONDITION 2 *[Acyclic] An S-spackling tuple* $\mathcal{T}$ *is* **acyclic** *if the following conditions hold:*

1. *noDeps(* $\texttt{postUses}_I$, $\texttt{holeUses}_I$ *)*
2. *noDeps(* $\texttt{postUses}_H$, $\texttt{def}_I \cup \texttt{holeUses}_I$ *)*
3. *noDeps(* $\texttt{def}_I$, $\texttt{def}_H \cup \texttt{preUses}_H$ *)*
4. *noDeps(* $\texttt{holeUses}_H$, $\texttt{preUses}_H \cup \texttt{def}_I$ *)*
5. *noDeps(* $\texttt{postUses}_I \cup \texttt{postUses}_H$, $\texttt{holeUses}_H$ *)*

The above preconditions on the splitting of the uses are necessary and sufficient for the routine `spackleSpanning()` to avoid introducing cycles. Furthermore, it is easily seen that the algorithm preserves semantic correctness. The transformation `spackleSpanning()` performs two sets of graph manipulations. The first set generates spill code, which consists of adding `store` and

`load` instructions and moving the post subsets of uses from the original defining instruction to a new `load`. When the S-spackling tuple satisfies the Acyclic Condition all newly added direct and indirect dependences between pairs of existing nodes will not introduce dependence cycles. The sequentializations introduced by spill code are lines 1 through 6 in Figure 6.2. Examination of the algorithm shows that all values are properly spilled. That is, uses of values which are delayed by spilling still access the same values after they are reloaded.

Thus, the transformation `spackleSpanning()` satisfies the property that semantic correctness is preserved. The second required property is that `reduceSpanning()` terminates. Since the procedure loops until there are no more excessive sets, termination is guaranteed if two requirements are meet:

1. each application of `spackleSpanning()` reduces the number of interfering spanning instances.
2. while an excessive set can be found, an application of `spackleSpanning()` can also be found, and

Requirement 1 implies that progress is made during each iteration of `reduceSpanning()`. Requirement 2 is a perquisite to applying `spackleSpanning()`. The satisfiability of each requirement is proved in turn.

Further examination of the procedure `spackleSpanning()` is required to demonstrate the termination property. To this end it is convenient to introduce the notion of *spanning instances*, which are analogous to register live ranges in that they both describe a spanning use of a resource. A live range is described as extending from the definition of that value to the last instruction to use that value. On the other hand, a spanning instance is described as extending from a use of a value back to the definition of a value. Thus, each use of a given value forms a separate spanning instance. This viewpoint is useful because different uses of a single value may be handled differently by the spackling transformation. A more formal definition is given as follows.

DEFINITION 16 *A* **spanning instance** *is a pair of instructions* (def, use)*, where* def *is a* **definition node** *which defines a value, and use is a* **use node** *which uses that value. A single definition may be a member of multiple spanning instances. Let* (def_1, use_{1_i}) *and* (def_2, use_{2_j}) *be two spanning instances, then they are said to be* **unique** *if and only if* def_1 *and* def_2 *are not the same node. A pair of unique spanning instances are said to be* **fully ordered** *if and only if*

$$\texttt{depends(}\ \texttt{use}_{1_i}\texttt{,}\ \texttt{def}_2\ \texttt{)} \vee \texttt{depends(}\ \texttt{use}_{2_j}\texttt{,}\ \texttt{def}_1\ \texttt{)} \tag{6.1}$$

A pair of unique spanning instances that are not fully ordered are said to be **interfering**.

Traditional coloring based register allocation two live ranges are said to overlap one another if the later value is defined before the earlier value is killed. This condition can be restated in terms

of spanning instances. Let def_1 and def_2 be two different instructions defining values. Then their definitions interfere if and only if there exists spanning instances (def_1, use_{1_i}) and $(def_2, use2_j)$ which interfere. This definition of interference naturally extends live range analysis to DAG based representations of a program and is therefore useful in describing spanning resource spackling.

In addition to partitioning the uses in such a way that the transformation does not introduce cycles, the splitting routine must also ensure that the partitioning will achieve a reduction in resource demands. A reduction occurs when the interfering spanning instances are placed in sets of their respective partitions such that they become fully ordered. The cases where the interfering spanning instances become fully ordered after the spackling transformation are given below.

CONDITION 3 *[Reducing] Let $\mathcal{T}$ be a S-spackling tuple. $\mathcal{T}$ is* **reducing** *if and only if there exists a pair of interfering spanning instances* $(\mathtt{def}_H, \mathtt{use}_{H_i})$ *and* $(\mathtt{def}_I, \mathtt{use}_{I_j})$*, and at least one of the following statements is true*

$$\mathtt{use}_{H_i} \in \mathtt{preUses}_H \wedge \mathtt{use}_{I_j} \in \mathtt{holeUses}_I \cup \mathtt{postUses}_I \tag{6.2}$$

$$\mathtt{use}_{I_j} \in \mathtt{holeUses}_I \wedge \mathtt{use}_{H_i} \in \mathtt{postUses}_H \tag{6.3}$$

$$\mathtt{use}_{H_i} \in \mathtt{holeUses}_H \wedge \mathtt{use}_{I_j} \in \mathtt{postUses}_I \tag{6.4}$$

It is straightforward to show when the spackling transformation is given a S-spackling tuple which satisfies the Reducing Condition the interfering spanning instance is removed.

LEMMA 1 *Let $\mathcal{T}$ be a S-spackling tuple that satisfies the Reducing Condition. Then the spanning resource spackling transformation will affect a reduction in the number of interfering spanning instances.*
PROOF: If the $\mathtt{def}_I$ spanning instance is placed in $\mathtt{use}_{hole_I}$ then the $\mathtt{def}_H$ spanning instance is ordered either before or after it by the transformation. If the $\mathtt{def}_H$ spanning instance is placed in $\mathtt{use}_{pre_H}$ then the $\mathtt{def}_I$ spanning instance is ordered after it by the transformation. If the $\mathtt{def}_H$ spanning instance is placed in $\mathtt{use}_{hole_H}$ then the $\mathtt{def}_I$ spanning instance must be placed in $\mathtt{use}_{post_I}$, which is always ordered after all nodes in $\mathtt{holeUses}_H$. In all cases the interference is removed, resulting in fewer interfering spanning instances. ■

To show that a satisfactory S-spackling tuple always exists it is necessary to consider all possible partial orderings between the four nodes of a interfering spanning instance. Let the interfering spanning instance be composed of the pairs (D_1, U_1) and (D_2, U_2). A partial ordering can be identified by a unique string of the form $wxyz$, where each variable represents a particular component of a partial ordering:

pattern	cases	reason
`A***`	27	symmetry between D_1 and D_2
`*A**`	18 (27)	no interference
`**B*`	12 (27)	no interference
`**NA`	4 (9)	U_1 `A` U_2 implies U_1 A D_2
`BN**`	5 (9)	D_1 `B` D_2 implies D_1 B U_2
`*N*B`	4 (9)	U_1 `B` U_2 implies D_1 A U_2
`NBNN`	1	symmetrical to `NNAN`
`NNAA`	1	symmetrical to `NBNB`
`NBAA`	1	symmetrical to `NNAB`

Table 6.1: Invalid and symmetrical orderings removed from consideration

w - the relation between D_1 and D_2

x - the relation between D_1 and U_2

y - the relation between U_1 and D_2

z - the relation between U_1 and U_2

Each component of the identifier can take one of three values:

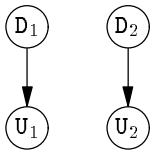
`B`: node n is ordered before node m

`A`: node n is ordered after node m

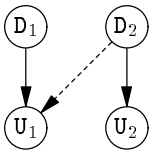
`N`: there is no order between node n and node m

There are a total of 81 combinations of the ordering values for the components of the ordering descriptors. After removing the illegal and symmetrical combinations, there are only 10 cases left, which are shown in Figure 6.4. The sets of partial orders removed are given in Table 6.1, where the character `*` indicate any of the values `A`, `B`, or `N`. The first column gives the pattern of cases removed. The second column gives the number of unique cases covered by the pattern and the total number of cases in parentheses. The final column explains the reason for removal.

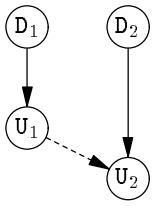
The ground work has now been laid to show that a S-spackling tuple can always be found which satisfies the Overlapping Sets, Acyclic, and Reducing Conditions. An algorithm exists which can find such a partition in two passes over the use nodes. In the first pass, all use nodes that must come before the hole are placed there and remaining use nodes are placed after the hole. The second pass then moves use nodes that follow the hole into the hole to ensure a reduction will occur. This movement is performed using a topological traversal. Thus the algorithm operates in graph linear time. Feasible placements of the uses for each unique case are given in Table 6.2.



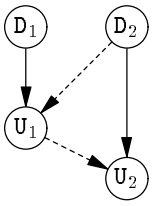
(a) Case nnnn



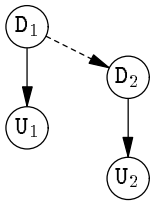
(b) Case nnan



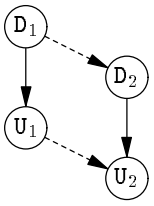
(c) Case nbnb



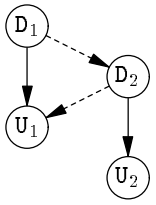
(d) Case nbab



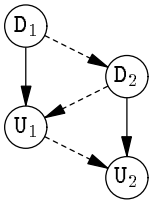
(e) Case bbnn



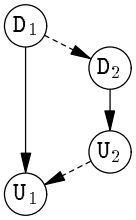
(f) Case bbnb



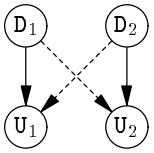
(g) Case bban



(h) Case bbab



(i) Case bbaa



(j) Case nban

Figure 6.4: All unique orderings of two interfering spanning instances

case	U_1	U_2
NNNN	$\texttt{preUses}_H$	$\texttt{holeUses}_I$
NNAN	$\texttt{postUses}_H$	$\texttt{holeUses}_I$
NBNB	$\texttt{preUses}_H$	$\texttt{holeUses}_I$
NBAB	$\texttt{holeUses}_H$	$\texttt{postUses}_I$
BBNN	$\texttt{preUses}_H$	$\texttt{holeUses}_I$
BBNB	$\texttt{preUses}_H$	$\texttt{holeUses}_I$
BBAN	$\texttt{holeUses}_H$	$\texttt{postUses}_I$
BBAB	$\texttt{holeUses}_H$	$\texttt{postUses}_I$
BBAA	$\texttt{postUses}_H$	$\texttt{holeUses}_I$
NBAB	$\texttt{postUses}_H$	$\texttt{holeUses}_I$

Table 6.2: Splittings of interfering spanning instances

THEOREM 3 *If there is a spanning excessive set then a S-spackling tuple $\mathcal{T}$ exists that satisfies the Overlapping Sets, Acyclic, and Reducing Conditions.*
PROOF: by construction.

Given that there is an excessive set, there must exist at least two interfering live ranges. If there is a dependence between them then let $\texttt{def}_H$ be the definition on the source of this dependence and $\texttt{def}_I$ be the other definition. Otherwise arbitrarily choose $\texttt{def}_H$ and $\texttt{def}_I$ from the two definitions. Let H be the hole following $\texttt{def}_H$.

The simplest partitioning of the uses of the two live ranges to satisfy the Acyclic Condition is the following. If $\texttt{def}_I$ is dependent on any uses from the set $\texttt{use}_H$, place those uses in $\texttt{preUses}_H$. Place all remaining uses of $\texttt{def}_H$ and all uses of $\texttt{def}_I$ in the sets $\texttt{postUses}_H$ and $\texttt{postUses}_I$, respectively. This partitioning ensures all nodes on which $\texttt{def}_I$ is dependent on are ordered before it. All remaining nodes are either dependent on $\texttt{def}_I$ or independent of it, and so can be safely ordered after it.

The partitioning must now be adjusted to satisfy the Reducing Condition. Let the set *post* be the union of the sets $\texttt{postUses}_H$ and $\texttt{postUses}_I$ from the step above. Since there are no cyclic dependencies between the nodes, at least one of the uses in *post* must have no ancestors in *post* and can be moved into its respective $\texttt{holeUses}_H$ or $\texttt{holeUses}_I$ set without violating the Acyclic Condition. Let use_{move} be this use and let use_{stay} be the other use of the pair of interfering spanning instances. Then allowing for the situations where a either or both of use_{move} and use_{stay} may be in both $\texttt{postUses}_H$ and $\texttt{postUses}_I$ as identified by the Overlapping Sets Condition, there are three cases.

1. There is a $\texttt{use}_{H_i}$ in $\texttt{preUses}_H$ as a result of the first partitioning step. Then equation 6.2 of the Reducing Condition is already satisfied.
2. $use_{move} \in \texttt{use}_I$ and $use_{stay} \in \texttt{use}_H$. Then placing use_{move} in $\texttt{holeUses}_I$ satisfies equation 6.3.
3. $use_{move} \in \texttt{use}_H$ and $use_{stay} \in \texttt{use}_I$. Then placing use_{move} in $\texttt{holeUses}_H$ satisfies equation 6.4.

Thus, after moving at most one use into the hole, all conditions are satisfied as required. ■

The above theorem shows that if there is an excessive for a spanning resource, then a reducing transformation can always be performed. Since every program contains a finite number of spanning instances, only a finite number of interfering spanning instances in excessive sets can exist. By Theorem 3 if there is an excessive set then an application of the transformation `spackleSpanning()` can be applied, which will reduce the number of interfering spanning instances. Therefore routine `reduceSpanning()` must eventually terminate and it satisfies the termination property.

(a) Before (b) After

Figure 6.5: Filling a non-spanning resource hole

6.2 Filling Non-spanning Resource holes

The non-spanning spackling transformation arranges for two instructions to temporally share a single instance of a non-spanning resource. The transformation simply introduces a temporal dependence between the two instructions. For the sake of completeness this section outlines the transformation and the necessary conditions required of the instructions.

DEFINITION 17 *Let* `H` *be a non-spanning resource hole delimited by the nodes* $\texttt{H}_{start}$ *and* $\texttt{H}_{end}$. *Let* `I` *be the instruction to be inserted in this hole. The 3-tuple* ($\texttt{H}_{start}$, $\texttt{H}_{end}$, `I`) *is referred to as a NS-spackling tuple.*

To achieve a reduction in resource requirements the instruction `I` must be independent of at least one of $\texttt{H}_{start}$ and $\texttt{H}_{end}$. With out loss of generality, $\texttt{H}_{start}$ is chosen to be the node.

CONDITION 4 *[Independence] Let T be a NS-spackling tuple. T is* **independent** *if and only if*

$$\texttt{nodep(I, H}_{start}\texttt{)} \wedge \texttt{nodep(H}_{start}\texttt{, I)} \tag{6.5}$$

The portion of the program graph affected by the non-spanning spackling transformation is shown in Figure 6.5. The non-spanning spackling transformation, `spackleNS()` simply consists of two calls to `addEdge()` to add a temporal edges from $\texttt{H}_{start}$ to `I` and from `I` to $\texttt{H}_{end}$. A driver procedure similar to `reduceNS()` is used to select NS-tuples, perform the transformation, and update the resource requirements information.

Instead of interfering spanning instances, non-spanning spackling deals with the number of other instructions in the excessive set which are independent of `I`.

DEFINITION 18 *Let E be a non-spanning excessive set and let* $\texttt{I} \in E$. *Then the set* $excessive_I = \{n | \texttt{nodep}(I, n) \wedge \texttt{nodep}(n, I)\}$ *is I's* **excessive set**, *and* $numEx_I = |exessive_I|$.

LEMMA 2 *Let T be a NS-spackling tuple that satisfies the independence condition. Then the following statements are true:*

1. *if there is a non-spanning excessive set then a NS-spackling tuple T which satisfies Condition 4 can be found,*
2. `spackleNS()` *reduces $numEx_I$, and*
3. `spackleNS()` *does not introduce any dependence cycles*

PROOF: All statements follow immediately from the combination of the independence condition and the definition of non-spanning excessive sets. ■

Using the concept of an instruction's excessive set it can be seen that `reduceNS()` terminates. Since every program contains a finite number of instructions, there must be a finite number of instructions in non-spanning excessive sets and thus each $excessive_I$ is also finite. By Lemma 2 if there is an excessive set then an application of the transformation `spackleNS()` can be applied, which will reduce the number of non-spanning interferences. Therefore routine `reduceNS()` must eventually terminate.

Chapter 7

Unified Allocation

The previous chapter described the mechanics of performing Resource Spackling. However, the application of these techniques in a unified resource allocation scheme must address several additional issues. Allocation of functional units has only a local impact on the program, while allocation of registers has both local and non–local effects due to the nature of spanning uses. Unified resource allocation is achieved by the simultaneous allocation of multiple resources. This simultaneous allocation performs resource spackling using a set of resource holes. To preserve semantic correctness additional constraints must be placed on the holes as a set. Finally, a method is needed to compare the cost of several allocation alternatives to prioritize them in terms of cost. This chapter describes heuristics to handle all of these issues.

7.1 Local Scheduling and Register Allocation

In the Measure and Reduce paradigm, local resource allocation is performed by introducing sequentiality between instructions whose resource demands exceed available resources. The sequencing places two instructions, which are on separate allocation chains, onto a single allocation chain. The result is that the two instructions are allocated a single instance of the resource and they share it temporally. Sequencing must be performed when there are excessive sets in a block.

Sequentialization is performed by selecting an instruction, i, in the excessive set that has the greatest slack time to be moved into holes. The slack time is used to prioritize the instructions since it indicates flexibility in finding a place to move the instruction. If there is a set of overlapping holes for all resources that i excessively uses within i's execution range, then i can be inserted in those holes without increasing the critical path length of the block.

If there is no set of holes within i's execution range, then an increase in the critical path length is unavoidable. There are two options. First, there may be a set of holes close to i's execution

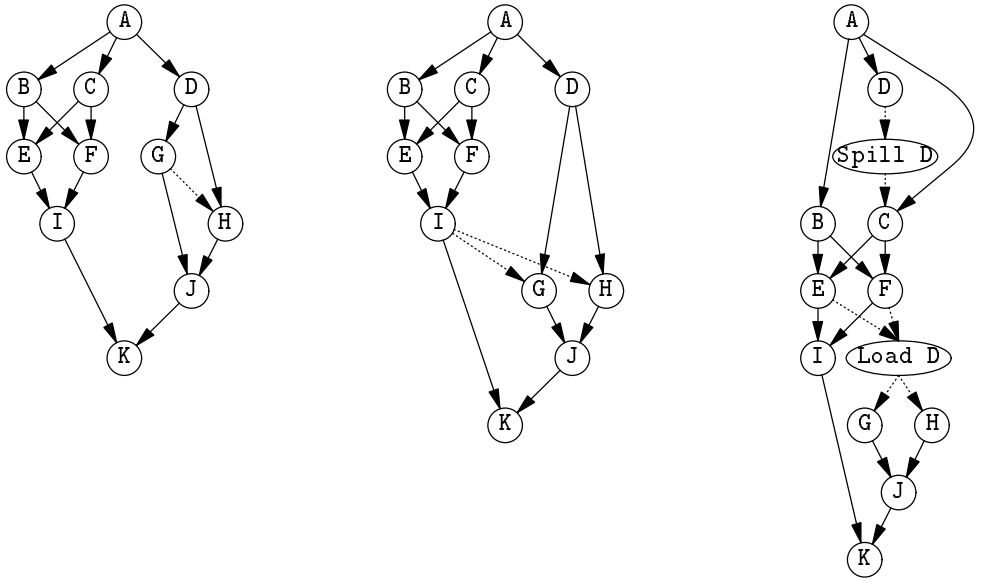
```
Procedure reduceBlock( block )
{
   While block has excessive sets do
   {  I = all instructions in all excessive sets for all resources;
      select i ∈ I with maximum slack_i;
      R = the set of resources that i excessively uses;

      if ( ∃ ∀ hole h_r whose ranges overlap with each other and i's execution range)
           r∈R
         holes = this set of holes;
      else
      {  close = the set of holes h_r s.t. r ∈ R whose ranges overlap and are closest
             to i's execution range;
         wedge = the set of holes created by wedged insertion for R;
         holes = the set, either close or wedge that minimally increases the critical path
             length of block;
      }
        ∀     place i in h_r by adding sequentialation edges;
      h_r∈holes
      if ( excessive spanning uses remain )
         spill uses between the excessive set and the hole containing i;
      remove i from excessive set information;
   }
}
```

Figure 7.1: Function *reduceBlock()*

range to which i can be moved. Second, wedged insertion (section 4.3) can be performed to create a set of holes for i's excessive uses. Both options can be considered and the one that minimizes the increase to the critical path length can be selected. A function based on this approach is given in Figure 7.1. Pseudo code for the algorithm for finding overlapping resource holes is presented in the next section.

As an example, consider the DAG in Figure 4.1(b). First assume that the target architecture has at least five registers and three functional units. Then the nodes `B`, `C`, `E`, `F`, `G`, and `H` are all members of at least one functional unit excessive set. Nodes `G` and `H` each have a slack time of one. There is a functional unit slack hole around each of `G` and `H`, so `G`'s hole overlaps with `H`'s execution range. Figure 7.2(a) shows the result of inserting `H` in `G`'s hole. Dashed arrows indicate sequentializing dependencies, *i.e.,* dependencies due to reuse of resources rather than data values.

Now assume that only four registers are available and `F` is selected to kill both `B`'s and `C`'s values and `H` is selected to kill `D`'s value. Then nodes `B`, `C`, `E`, `G`, and `H` are in both functional unit and register excessive sets. Node `G` has slack time but there are no holes in its execution range. Therefore the algorithm must increase the critical path length. There are a functional unit and a register hole available after `F` executes since it kills two values and only needs one register for itself. Inserting `G` in the hole following `F` would increase the critical path by one instruction. Wedged insertion would



(a) Functional unit sequencing (b) Register sequencing (c) Register spilling

Figure 7.2: Local reductions of resource requirements

increase the critical path length more because the pseudo-hole must be large enough to spill and reload a value. Therefore the algorithm chooses the hole close to `G` instead of performing wedged insertion. The resulting DAG is shown in Figure 7.2(b).

Although the creation of some live values may be delayed by sequencing, the instructions that compute the live values may need input values. These input values remain alive from where they are computed to where the excessive instructions are moved. In Figure 7.2(b) the value computed by `G` was delayed until there was a register available for it. However, `D`'s value remains alive until after both `G` and `H` execute. In this example it is impossible to reduce the register requirements below four using sequentialization alone. When such a situation occurs sequentialization must be combined with register spilling. There are two options for selecting what values to spill. Either the values in the excessive set may be computed and spilled, or the input values may be spilled. The option selected depends on what holes are available. Computing and spilling the excess values prior to the excessive set requires finding additional functional unit and register holes, while spilling the input values requires additional functional unit holes to where the values are moved. An additional criterion to consider is the number of values that must be spilled in each case.

Continuing with the above example, assume the same killing instructions and that the target architecture has three registers and two functional units. As before, the nodes `B`, `C`, `E`, `G`, and `H` are in a register excessive set and only instructions `G` and `H` have slack time. Free functional unit and register holes become available after `F` executes, and another set of free functional unit and register holes

```
Function findOverlappingHoles( DAG, i, resourcesNeeded ) returns holeSet
{
      /* find all usable holes */
   foreach resource r ∈ resourcesNeeded
   {  viable[r] = ∅;
      foreach hole h of type r
         if ( r is spanning )
         {  if ( canSplitUses(h, i) )
               viable[r] ∪= h;
         }
         else
         {  if ( canInsert(h, i) )
               viable[r] ∪= h;
         }
   }

      /* find best set of holes */
   bestCost = MAXINT;
   bestSet = ∅;
   foreach holeSet hs ∈ {h1 ... h|resourcesNeeded|}
   {  if ( noCycles(hs) )
      {  cost = insertCost( i, hs );
         if ( cost < bestCost )
         {  bestCost = cost;
            bestSet = hs;
         }
      }
   }

   return bestSet;
}
```

Figure 7.3: Function *findOverlappingHoles()*

become available after `I` executes. `G` and `H` are placed in the free holes. However, the sequentialization would still leave the excessive set {`B`, `C`, `D`, `E`}. Thus a spill must be performed. To minimize the number of spills, the algorithm spills the input value, `D`. The resulting DAG is shown in Figure 7.2(c).

The algorithm presented in Figure 7.1 uses the framework to simultaneously allocate registers and functional units. More advanced algorithms, such as one based on first sequencing instructions in order of number of excessively used resources can also be developed using the framework.

7.2 Finding Overlapping Holes

Unification of resource allocation is achieved by performing resource spackling for all resources needed by an instruction simultaneously. This process requires that a set of resource holes be found that are independent of each other, i.e., no cycles will be introduced if the instruction is

inserted into all of them. A straightforward approach is to consider all combinations of holes for all needed resources. This approach first examines all holes of each needed resource type and records those which are viable. All combinations of these viable holes are then generated. For each combination the algorithm first checks that the holes are mutually independent and if so, computes the cost of using them. The combination with the least cost is then selected for unified allocation of resources needed by the instruction. A sketch of such an algorithm is given in Figure 7.3. The function `canSplitUses()` checks the dependences and splits the uses of the hole and inserted instruction live ranges as described in Chapter 6 for spanning resources. The function `canInsert()` checks the dependences for inserting instructions in non–spanning holes as also described in Chapter 6. Function `noCycles()` checks if the holes are mutually independent. More elaborate heuristics can be constructed which prioritize the search for holes and use more intelligence in determining which holes are independent of each other.

7.3 Selection Heuristics

The `spackleSpanning()` transformation depends on heuristics to select a particular instruction and hole pair and a partitioning of their uses into the required sets. The requirements placed on the partitioning of uses of spanning resources for the transformation leave some latitude to the partitioning heuristics. The design of these heuristics needs to weigh the impact of potential 6-tuples on the resulting quality of code generated. Several criteria for consideration are suggested in this section.

Since the application of the `spackleSpanning()` transformation sequentializes independent nodes, the length of the critical path of the scheduling unit may increase. Using the following definitions the impact on the critical path length can be accurately predicted.

DEFINITION 19 *Assume the 7-tuple given to* `spackleSpanning()`.

1. *The* $\texttt{def}_H$ **store time**, τ_{storeH}, is the execution time of a `store` instruction if $\texttt{postUses}_H$ is nonempty and zero otherwise.
2. The $\texttt{def}_H$ **load time**, τ_{loadH}, is the execution time of a `load` instruction if $\texttt{postUses}_H$ is nonempty and zero otherwise.
3. The $\texttt{def}_I$ **store time**, τ_{storeI}, is the execution time of a `store` instruction if $\texttt{postUses}_I$ is nonempty and zero otherwise.
4. The $\texttt{def}_I$ **load time**, τ_{loadI}, is the execution time of a `load` instruction if $\texttt{postUses}_I$ is nonempty and zero otherwise.
5. The function `height(subtree)` return the height of the tree `subtree`, which is defined to be the length of the critical path of the tree.

There are five subtrees of interest:

1. $before = \{\texttt{def}_H \cup \texttt{use}_H \cup \texttt{def}_I \cup \texttt{use}_I\}$
2. $preH = \{\texttt{def}_H \cup storeH \cup \texttt{preUses}_H\}$
3. $postH = \{loadH \cup \texttt{postUses}_H\}$
4. $holeI = \{\texttt{def}_I \cup storeI \cup \texttt{holeUses}_I \cup \texttt{holeUses}_H\}$
5. $postI = \{loadI \cup \texttt{postUses}_I\}$

The length of the critical path of the local tree before the potential transformation is given by

$$cpl_{before} = height(before) \tag{7.1}$$

The length of the critical path of the local tree after the potential transformation is given by

$$cpl_{after} = height(preH) + height(holeI) + max(height(postH), height(postI)) \tag{7.2}$$

If the height, I_{est}, and depth, I_{lft}, of each node I is precomputed, then the values of $height(before)$, $height(preH)$, and $height(preI)$ can be computed in a single pass of the respective subtrees. Furthermore, if the topological ordering of the nodes in the scheduling region is saved, the values $height(postH)$ and $height(postI)$ can also be computed in linear time. These values are then used to compute the difference in local critical path lengths.

$$cpl_{\delta} = cpl_{after} - cpl_{before} \tag{7.3}$$

To determine if the potential transformation has any impact on the overall critical path length of the scheduling region, Equation 7.3 can be compared to the slack time of the subtree $before$.

The impact of two candidate ($\texttt{def}_H$, $\texttt{def}_I$) pairs can be compared using Equation 7.3. However for any given ($\texttt{def}_H$, $\texttt{def}_I$) pair more than one partitioning may exist. Simple heuristics can be designed to partition the uses in an attempt to minimize the resulting cpl_{δ}, given whether zero, one or two values are spilled.

Orthogonally, when given appropriate latitude, the heuristics can select which values should be spilled. If the partitioning restrictions allow, either or both of $\texttt{postUses}_H$ or $\texttt{postUses}_I$ to be empty, the need for spilling the respective value is removed. The size of the resulting $preH$ or $holeI$ tree must be compared to the cost of spill code to determine if such a case is beneficial.

Two special cases exist when $\texttt{def}_H$ is a `load` instruction. Such a case may occur as a result of an earlier application of `spackleSpanning()`. Minimally, the *storeH* instruction is not needed as the value is already available in memory. In addition, the $\texttt{def}_H$ instruction itself can be removed if the set $\texttt{preUses}_H$ is empty. Such a case can occur if a previous `spackleSpanning()` transformation does not delay $\texttt{postUses}_H$ until there are sufficient resources. The subsequent application of `spackleSpanning()` then increases the delay time.

Chapter 8

Global Code Motion

Global code motion is a technique used to redistribute ILP among different basic blocks and regions of a program. The goal of this redistribution is to reduce the program's execution time through more efficient use of resources. Reduction of a program's execution time depends on an accurate assessment of the usage of all resources. This chapter presents heuristics to use resource spackling to handle the particular problems encountered when performing resource allocation during global code motion.

8.1 Resource Conscious Global Code Motion

The goal of global scheduling is to move instructions from a *source* block to a *destination* block to decrease the execution time of the source block. A decrease in execution time is achieved when the critical path length is reduced in the source block while the critical path length of the destination block is not increased. The instructions moved are called *fill instructions* since they are inserted in holes in the destination block.

Fill instructions may be found in several source blocks, some of which depend on the types of execution supported by the target architecture, *e.g.,* speculative or guarded execution. Consider the control flow graph of an `if` statement in Figure 8.1. Instructions can be moved between blocks that share the same set of control dependencies, such as blocks `B1` and `B2`. Instructions can be moved above conditional branches, from `Bthen` and/or `Belse` to `B1` if either the moved instructions do not violate data dependencies or if the architecture supports speculative execution [HP87, SHL92]. Instructions can be moved below join points or above split points, from `Bthen` and/or `Belse` to `B1` or `B2`, if the architecture supports guarded execution [DHB89, HD86]. Lastly, instructions can be moved either below conditional branches or above join points by duplicating the moved instructions on each branch, *e.g.,* from `B1` or `B2` to both `Bif` and `Belse`. The individual cases are not considered while

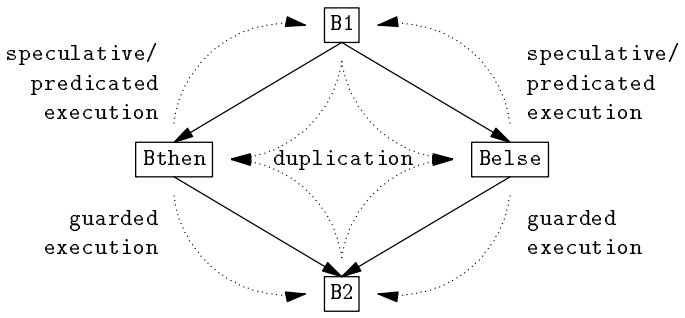


Figure 8.1: Code motion techniques and architectural supports

describing the mechanics of filling holes. It is assumed that the features of the target architecture are known to the algorithm and only instruction moves supported by the architecture are considered. Without loss of generality only the upward movement of fill instructions is discussed.

If the direction of code motion is upward, the instructions are taken from the beginning of the source block. If the direction of code motion is down, the instructions are taken from the end of the source block. There are, as discussed later, special situations where it is desirable to move instructions that are not on a critical path, even though these will not have a direct impact on reducing the overall execution time. In these situations the moved instructions make additional resources available which can be used by subsequent moves, resulting in a reduction in execution time.

Existing global code motion techniques [Fis81, AN88, GS90] can be adapted to use Resource Spackling based heuristics to unify functional unit and register allocation, and determine which code motions are beneficial. Two properties of code motion must be satisfied to realize a benefit.

1. The critical path length of the destination block must not be increased.

2. The critical path length of the source block must be reduced.

To satisfy the first property, it must be ensured that there are sufficient resources available for all instructions being moved. If this is not the case wedged insertion would have to be performed, negating the reduction in the critical path length in the source block.

To satisfy the second property, all instructions which are at one end of a block and are on a critical path must be moved together; otherwise the critical path length will not be reduced.

DEFINITION 20 *A* **critical set** *of instructions with* **length** L *is the smallest set of instructions that must be removed from the end of a block to reduce the block's critical path length by* L *cycles.*

Consider removing nodes from the top of the DAG in Figure 4.1(b). The first critical set is {`A`}. When `A` is moved the length of the DAG is reduced by the execution time of `A`. Then the next

```
Function fill( dest, source )
{
   reduce = 0;
   While dest has holes do
   {
      cs = next set of critical instructions from source;
      foreach instruction i in the critical set
      {  compute EST_i based on its dependencies in the destination block
          LFT_i = LFT of the last instruction in the destination block
      }
         /* find overlapping resource holes */
      foreach instruction i in the critical set, in decreasing
         order of EST_i
      {  forall resources r required by i
          {  select holes h_r such that they overlap with the other holes
                selected and with i
             if no such holes exist
             {  undo all moves from the current critical set;
                return reduce;
             }
             Insert i into h's allocation chain;
          }
          Update the hole description information;
      }
      reduce = reduce + min_{i ∈ cs} ( τ_i );
   }
   return reduce;
}
```

Figure 8.2: Function *fill()*

critical set is {C, D}. Although B can now be moved, it is not in the critical set since moving it would not affect the length of the critical path.

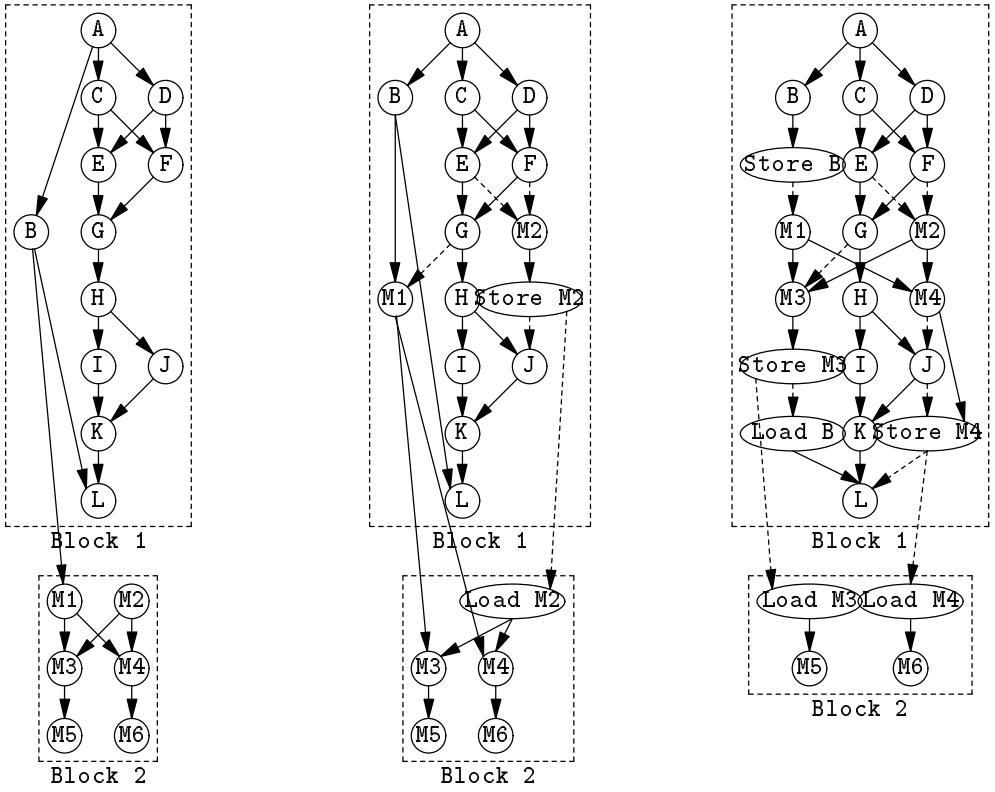
Three methods for performing resource conscious global code motion are suggested:

1. Move individual instructions from minimal critical sets;

2. Apply reduction techniques to minimal critical sets; and

3. Apply reduction techniques to an estimated maximal critical set.

In the first approach one instruction at a time from the critical set is moved to the destination block. Thus, one instruction at time from a critical set is allocated its resources. If all instructions cannot be allocated their resources, none of the instructions in the critical set are moved. The allocation of resources is similar to that in local schedulers. Overlapping resource holes are found for all resources required by each instruction. However, the holes must be within the instruction's execution range, and wedged insertion is not performed, since the goal is to avoid increases to the critical path length of the destination block.



(a) DAG of source instructions

(b) After moving two instructions

(c) After moving four instructions

Figure 8.3: Example of global code motion

The set of fill instructions must be inserted into the identified holes. Insertion requires each instruction to be placed in the proper allocation chains. This modification to the allocation chains will change the size and shape of the holes in which the fill instructions are inserted. For non-spanning resources, fill instructions may be interleaved with instructions already in the destination block. Spanning resources should ideally have all fill instructions inserted consecutively, since any gaps between inserted instructions can only be used if there is enough room to accommodate an instruction to spill the use. Slack holes that have instructions inserted will have their size reduced, and may have their range reduced due to dependencies of the fill instructions.

Consider moving instructions from block 2 to block 1 in Figure 8.3(a) using method 1. Assume that there are three functional units and four registers available. The first critical set consists of instructions `M1` and `M2`. Instruction `M2` can be inserted in the functional unit hole following `F` and the register hole following `D` since the holes overlap. `M2`'s value must be spilled since the register hole is not available to the end of the DAG. `M1` can be inserted in the functional unit hole following `B` and the register hole following `G`, which results from killing `F`. The value computed by `M1` need not be

spilled. The resulting DAGs are shown in Figure 8.3(b). Next `M3` and `M4` are moved up. Since `M4` is selected to kill both `M1` and `M2` it can use the same functional unit and register as `M2`. Instruction `M3` can use `M1`'s functional unit, and it will also take `M1`'s register, forcing `M1` to use `B`'s register. `B`'s value must now be spilled around the inserted instructions and `M3`'s value is spilled before `B`'s value is reloaded. The resulting DAGs are shown in Figure 8.3(c).

The second and third methods remove critical sets from the source block and place them in the destination block. The reduction techniques discussed in chapters 6 and 7 are used to remove any resulting excessive sets. The transformation is made permanent if there is no increase in execution time of the destination block. The difference between methods 2 and 3 is in the size of the critical set moved. Method 2 uses the same minimal length critical set as method 1. Method 3 uses heuristics to compare the resources available in the destination block and requirements of candidate critical sets to estimate the largest critical set that can be moved and supported by the destination block.

The methods use available spanning resources in the destination block to different degrees of efficiency. The methods are listed in increasing order of efficiency. There are two ways in which spanning resources may be inefficiently used. In the first case, consider moving two instructions, `a` and `b` that use the same value, `v`, already computed in the destination block. Method 1 may move instruction `a`, allow it to reuse `v`'s register, spilling `v` in the process. When instruction `b` is moved, it must be placed after `v` is reloaded, possibly causing the movement of the critical set to fail due to an increase in the schedule. Methods 2 and 3 would attempt to schedule both `a` and `b` before the spill, leaving more room to move subsequent instructions that are dependent on either `a` or `b`.

In the second case, consider moving instructions `c` and `d`, where `d` uses the value computed by `c`. If the register used to hold `c`'s value is not available from the execution of `c` to the end of the block it must be spilled. In methods 1 or 2 when the subsequent critical set containing instruction `d` is moved, it may fail because `c`'s value must be reloaded before it can be used. Method 3 would determine ahead of time if both instructions `c` and `d` can be moved. If so, the reduction transformations will avoid spilling `c`'s value before it can be used by `d`. As can be seen, the overhead for more efficient use of spanning resources is increased bookkeeping and complexity of determining the appropriate length of critical set to be moved.

Traditional global schedulers, based on list scheduling, are able to identify available functional units, *i.e.,* functional unit holes. However, since the scheduler is separate from the register allocator, it does not know if there are registers available for the instructions that are moved up. Similarly, these schedulers cannot recognize when instructions from other blocks should be moved up above instructions in the block with slack time, since these schedulers usually schedule all instruc-

tions in the current block first. Resource Spackling can move instructions from other blocks above instructions with slack time in the current block when overlapping resource holes are available.

The methods for identifying holes and performing code motion allow Resource Spacklingto fill holes that occur in the middle of a block, even if registers are not available from the hole to the origin of the fill instructions. Such capability is an improvement over techniques that do attempt to consider resource demands [ME92]. When holes occur only at the beginning or end of basic blocks, the problem degenerates to Shape Matching [MGS92]. However, Shape Matching considers only functional units, while in this case Resource Spackling performs the equivalent of shape matching across all resources at once.

When fill instructions have live uses of spanning resources at the end of the hole, the uses must be spilled to free the resource for uses that follow the hole. Thus, a spill instruction is inserted as the last instruction in the hole and a corresponding `load` instruction must be inserted in the source block. The insertion of `store` and `load` instructions may increase the critical path length if the `load` instruction takes longer to execute than the instructions removed from the block. This scenario suggests that instructions should not be moved if `load` instructions must be inserted in the source block, but this limits the reductions that can be made. A better approach is to allow temporary increases in the source block's critical path. The movement of the first set of critical instructions allows subsequent critical sets to be moved. If several sets of instructions are moved their total reduction may offset the cost of inserting `load` instructions for the last (live) spanning uses of the moved instructions. In this case there is a net reduction in the source block's critical path, as desired.

If instructions are moved below a conditional branch or above a join point, the instructions must be duplicated on each branch. In this situation the execution time along the selected path of execution is reduced. In the best case the duplicated code can be placed in holes on the duplication branches. In the worst case, the execution time along the duplicated path(s) will not change. The worst case occurs when there are not enough holes in the duplication branches to absorb the duplicated instructions. In this case they are simply added to the end of the block. The duplicated instructions must be executed anyway and they were already on a critical path created by concatenating the two blocks. Thus, moving them from one block to another cannot increase the execution time.

8.2 Selection of Fill Sets

When several different sources of fill instructions are available for consideration, a decision must be made as to which set is most beneficial. Only sets for which the destination block has room are considered, although the techniques mentioned in the previous section can be used to try to sets that apparently exceed the currently available resources. In practice, the code motion must be performed tentatively in case the whole critical set cannot be moved without increasing the critical path.

A technique commonly used for selecting instructions to move in other code motion schemes is to choose instructions from the source block with the highest branch probability. This Resource Spackling based heuristic generalizes the approach by using execution counts of the basic blocks. This approach allows more direct handling of moving instructions across several conditional branches or join points. There are several other possible factors that can be considered, such as the amount of wasted hole space, the number of `store` instructions inserted, the amount of duplicated code, and how far the instructions can be moved.

The techniques presented here permit fill instructions from several different source blocks to be moved to the same destination block. However, due to the nature of spanning resources, once fill instructions from one source have been selected it is more likely that the next set of fill instructions will be selected from the same source block than from a different source block. The reason is that fill instructions from the same source block may be able to reuse some of the spanning resources used by the first set of fill instructions, while fill instructions from a different source block will require additional resource instances. Thus, choosing the first fill instruction set for a set of holes may be a more critical decision than subsequent fill instruction sets.

The above observation, along with the problem of inserting `load` instructions in the source block suggests an alternative approach to selecting fill instructions. In this approach each source block is considered independently and as many critical sets as possible are tentatively moved. Then the overall impact can be assessed in terms of the criteria of wasted space and number of `store` instructions inserted. An additional benefit of such an approach is that it allows a better criterion than selecting the block with the highest branch probability. Instead, the execution count can be multiplied by the reduction in the critical path. As an example, consider two branches, A and B, with execution counts equivalent to branch probabilities of .7 and .3 respectively. Assume that the critical path length of A can be reduced by 2 while the critical path length of B can be reduced by 5. Then the overall reduction in execution time if A is reduced is $2 \times .7 = 1.4$ while the overall reduction in execution time if B is reduced is $5 \times .3 = 1.5$. Thus, block B should be the source block, even though block A has a higher branch probability.

Chapter 9

HARE

The previous chapters have described how the resource requirements of a program can be measured and incorporated into an intermediate representation, as well as the mechanics of performing resource allocation by placing instructions in resource holes. This chapter presents an application of the URSA framework that integrates instruction scheduling with global register allocation and assignment, called Hierarchical Allocation of REgisters (HARE). HARE addresses several register allocation issues, such as coalescing, spill code placement, SSA copy placement, and register assignment, by enhancing the previously known algorithms by making them resource conscious.

9.1 Overview of HARE

HARE consists of a phase for the allocation and a phase for assignment of registers. The allocation phase is a hierarchical application of the Measure and Reduce paradigm to GURRR that unifies instruction scheduling and global register allocation. The primary task of the allocation phase is the selection of values to spill and the placement of spill code to minimize the execution time of the program. The assignment phase hierarchically assigns registers to the values computed. In addition, the assignment phase inserts register copy instructions.

The goal of HARE is to improve the quality of code generated by the allocation and assignment of registers. To accomplish this, each phase considers both the availability of registers and functional units and the overall cost of a register allocation decision using the region execution counts. The major steps of the algorithm are shown in Figure 9.1.

Register coalescing is performed on the GURRR of a program prior to allocation. Traditionally, coalescing is performed in instances where the source and destination live ranges of a copy instruction do not interfere. The two live ranges interfere when there are other uses of the source value that can execute in parallel with the copy instruction. However, by using GURRR, HARE ex-

```
Procedure allocateAndAssign( PDG )
{
      /* compute GURRR */
   buildPDG();
   convertToSSA();
   coalesceValues();
   addReuseDags();

      /* do hierarchical register allocation */
   foreach region reg in bottom-up order
   {  while ( reg has excessive sets )
      {  find all resource holes in reg;
         call estimateRegion(reg) which selects nodes from the
            excessive sets and places spill code in FU holes;
           consider the cost of reduction transformations that
              1) use holes in reg
              2) place spill code in a dependent if region
              3) reduce register demands of a dependent region
         call allocateRegion(reg) to perform allocations selected by
            estimateRegion();
      }
   }

      /* do register assignment
      in each step the number of registers assigned in a
      region is minimized */
   foreach region reg in bottom-up order
      foreach value def in reg reaching other regions
         assignDef(def);

   foreach region reg in bottom-up order
      foreach φ node phi in reg
         assignSsaDef(phi);

   foreach region reg in bottom-up order
      foreach unassigned value def in reg
         assignDef(def);
}
```

Figure 9.1: Top level register allocation algorithm

tends coalescing by identifying cases when the live ranges can be ordered to remove the interference without increasing the execution time. Temporal dependencies can be added from the other uses to the uses of the copy instruction if the length of the critical path is not increased as a result. The two live ranges are then combined and the copy instruction is removed. After coalescing, the initial register reuse information is computed.

Consider the program DAG of a region in Figure 9.2(a) and assume that instruction `J` is a copy instruction of the value computed by instruction `G`. Instructions `H` and `I` are the other uses of `G`. The only use of `J` is instruction L_2, which is already dependent on instructions `H` and `I`. Since L_2 can be executed after both `H` and `I` without increasing the execution time of the region, `J` can be coalesced with `G`. Instruction `J` is removed and a data dependence is added from `G` to L_2. The resulting program DAG is shown in Figure 9.2(b).

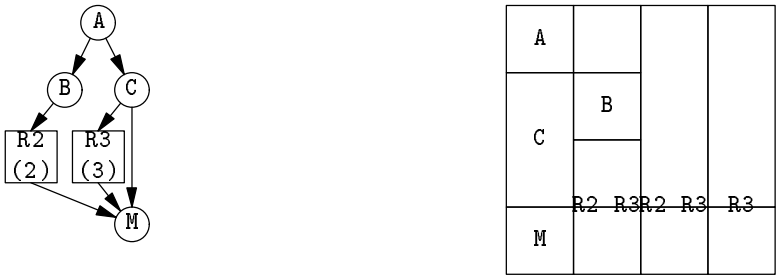
Register allocation is performed in a bottom-up traversal of the regions. In each region the allocation is performed by measuring and reducing excessive register requirements. When an excessive register use is encountered, several options for its reduction are considered, including delaying the defining instruction, different resource holes for spill code, rematerialization of the values, and additional register reductions in subregions. For each option any increase in the critical path of the region or subregion is multiplied by the execution count of the region or subregion to give a total cost in terms of execution time. The minimum cost option is then selected. The result of the allocation phase is that each region is transformed by the reductions and allocated a sufficient number of registers for all values computed by the dependent instructions and child regions.

The assignment of registers must also be performed hierarchically to ensure that no more than the allocated number of registers are assigned in each region. The assignment phase consists of three bottom-up traversals of the regions where each pass assigns all values of a particular type. The types are: 1) global variables, 2) values used or defined by SSA ϕ nodes, and 3) values local to a single region. The second traversal determines when register copy instructions are required and uses the functional unit resource hole and execution count information to place the copy instructions to minimize any increase in the program's execution time.

The hierarchical nature of the GURRR supports the summary of the resource requirements of the instructions in a region in the region node. These requirement summaries are then used in computing the resource requirements for the parent region(s). Figure 9.3(a) shows the register *Reuse* DAG for region 1. Summary nodes for regions 2 and 3 are included in region 1's $Reuse_{Reg}$ DAG. These nodes represent the register requirements of the child regions, which are indicated by the numbers in parenthesis. Since regions 2 and 3 are the branches of an `if-then-else` statement,

(a) Region $Reuse_{FU}$ DAG (b) After coalescing

Figure 9.2: Example of coalescing



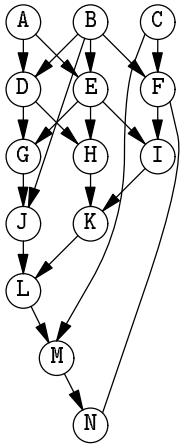
(a) Region 1 $Reuse_{Reg}$ DAG (b) Region 1 Register schedule

Figure 9.3: Region 1 Register usage

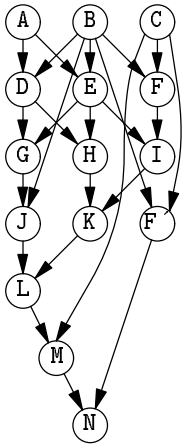
the maximum of their requirements is used to compute the register requirements of region 1. Thus, region 1 requires a total of four registers. One of these registers is used only for values computed in region 1, two registers are used only for values computed in the child regions, and one register is used in both region 1 and region 3. The schedule for registers in region 1 is shown in Figure 9.3(b). Regions 2 and 3 are shown on two and three register allocation chains, respectively. Regions 2 and 3 are drawn on the lines to indicate that they identify boundaries for the register holes, *i.e.,* the last two registers are available for use in region 1 both before and after the branches of the `if-then-else`, but are used in the branches.

9.2 Allocation and spill code placement

A goal of HARE during allocation is to minimize the cost of spill code introduced. Three factors affect the cost and thus the selection of a reduction transformation: 1) placement of spills in less frequently executed regions, 2) selection of values that permit spill code to use under utilized resources, and 3) less costly alternatives to memory accesses, such as rematerialization. When several reduction transformations are possible, HARE estimates the cost of each transformation and selects the one that will cause the least increase in the overall execution time of the program.



(a) Program DAG

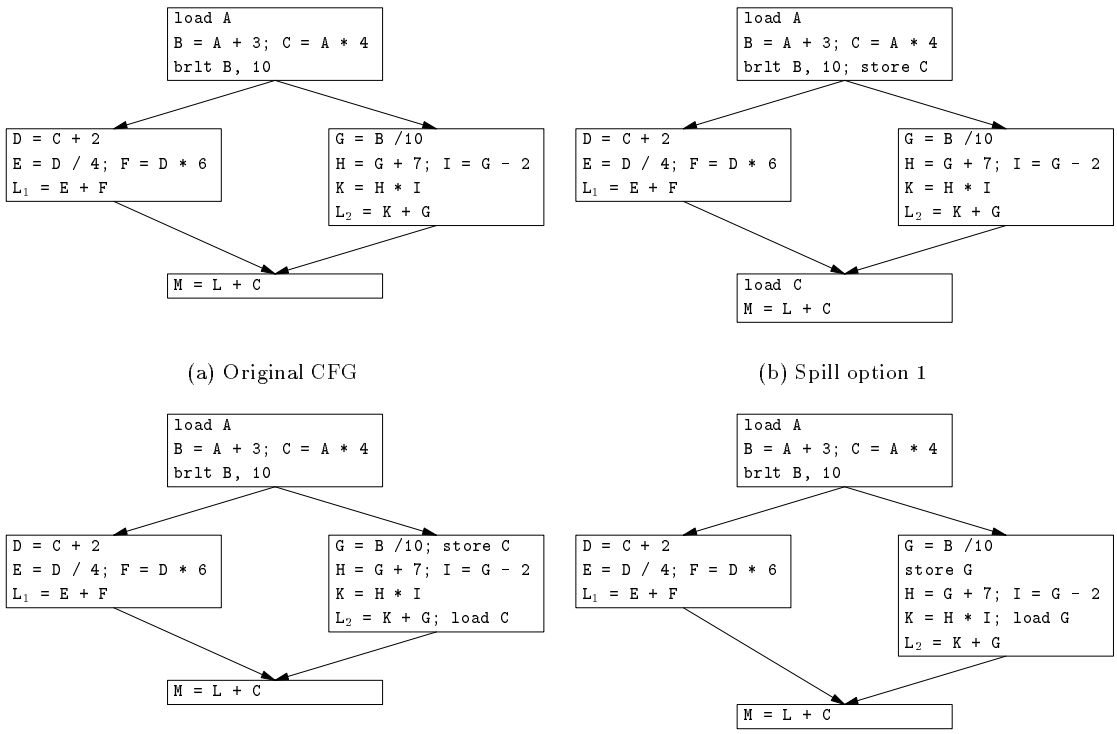


(b) Program DAG after rematerialization

Figure 9.4: Example of rematerialization

Besides spilling values to reduce register demands, HARE also considers rematerialization of values. In addition to the traditional candidates for rematerialization, such as loading constant values and computations using system registers and constants [BCT92], HARE examines the selected instructions and the data dependence graph to identify values that can be recomputed from other values already in registers. As an example, consider the program DAG in Figure 9.4(a) and assume that the architecture has three functional units and five registers. The set of the six instructions {`B, C, F, G, H, I`} is an excessive set. The instruction `F` is selected for reduction since there is distance between its first and second uses. While functional units are available to execute the `load`, they are not available to execute the `store` before instructions `G`, `H`, and `I`, and thus the execution time would be increased if `F` were spilled. However, HARE can determine that since the values computed by `B` and `C` are still in registers after the execution of `G`, `H`, and `I`, the value computed by `F` can be rematerialized. The resulting program DAG is shown in Figure 9.4(b). Since HARE considers the availability of functional units to perform the rematerialization, arbitrarily large computations can be candidates for rematerialization.

The different types of reduction transformations considered are listed in the algorithm in Figure 9.1. Assume that HARE is being applied to region 1's $Reuse_{Reg}$ DAG in Figure 9.3(a) and that the architecture provides three registers. The value `C` is selected for spilling since value `B` is used in both region 1 and region 2. Option 1 considers placing the spill code in region 1, with the `store` being executed currently with the branch instruction and the `load` being executed after the branches return and before `M`. Option 2 considers placing the spill code in region 3. Spill code is not required for region 2 in this case since the total register requirements of regions 1 and 2 do not exceed the



(a) Original CFG

(b) Spill option 1

(c) Spill option 2

(d) Spill option 3

Figure 9.5: CFGs for the spill options

architecture's resources. Option 3 considers reducing region 3's register requirements from three to two. Again, region 2 does not need to be reduced in this example since it does not appear in an excessive set. The control flow graphs resulting from each option are shown in Figure 9.5.

The cost of a potential reduction transformation is determined by multiplying the estimated increase in the length of region's critical path by the region's execution count. There are several possible methods for estimating the impact of a reduction transformation on the length of the critical path. A simple estimation technique is presented.

The estimation technique determines the execution time of all spill code introduced and compares it to the total time of the availability of the functional units used to perform the spilling or rematerialization. Pseudo code for the estimation algorithm is given in Figure 9.6. The availability of the functional units is found by summing the size of all of the functional unit holes in the region. The instructions that compute values that are in the register excessive set are grouped by their distance from the beginning of the excessive set. In each group enough values are selected so that the group will no longer have an excessive number of values. To provide the most flexibility in using available functional units, the instructions with the most slack time are selected.

Each instruction selected is checked to see if its value can be rematerialized instead of

```
Function estimateRegion( region, goal )
   returns estimated cost of reducing region to use only goal registers
{
      /* compute availability of holes in region */
   holeTime = 0;
   foreach functional unit hole h in region
      holeTime = holeTime + h.size;

      /* determine requirements for holes*/
   reduce = ∅;
   group the nodes in the excessive set for region;
   foreach group grp
      if ( |grp| > goal )
         reduce = reduce ∪ |grp| - goal instructions with most slack time;

      /* use hole time for instructions spilled in region */
   foreach node n ∈ reduce that must execute in region
      if ( canRematerialize(n) and
           rematerializeTime(n) < spillTime(n) )
         holeTime = holeTime - rematerializeTime(n);
      else
         holeTime = holeTime - spillTime(n);

      /* determine where to spill other nodes */
   foreach node n ∈ reduce that may be spilled in a dependent region child
   {  if ( canRematerialize(n) and
           rematerializeTime(n) < spillTime(n) )
         need = rematerializeTime(n);
      else
         need = spillTime(n);
      if ( child.holeTime > 0  ||  holeTime <= 0 )
         child.holeTime = child.holeTime - need;
      else
         holeTime = holeTime - need;
   }

      /* total costs */
   if ( holeTime < 0 )
      cost = -holeTime * region.execCount;
   else
      cost = 0;
   foreach child region child where spill code was added
      if ( child.holeTime < 0 )
         cost = cost + (-child.holeTime * child.execCount);

      /* compare against reducing a dependent region */
   if ( there is a child region child in region.excessiveSet )
   {  subGoal = child.requirements - (region.requirements - goal);
      childCost = estimateRegion( child, childGoal );
      if ( cost < childCost )
         return cost;
      else
         return childCost;
   }
}
```

Figure 9.6: Allocation cost estimation algorithm

spilled. The function *canRematerialize(n)* determines if and how the value computed by n can be rematerialized. The time to rematerialize n is then compared to the time to spill n and the transformation for the smaller of the two is used.

The placement of spill code is performed in two steps. In the first step only values whose spill code is constrained by dependencies to the current region are examined. The cost of the spill code for these values is subtracted from the available time of the functional units. In the second step the remaining values, which can have their spill code placed in either the current region or a child region, are examined. The spill code is placed in the child region if there are sufficient functional units available in it, or if there are insufficient functional units available in the current region.

The cost of inserting spill code is determined by multiplying each region's increase in critical path length by its execution count. Finally, if the excessive set contains child regions, a reduction of these regions is estimated. The cost of these recursive reductions is compared to reducing the current region and the least expensive one is selected. The values and places for spill or rematerialization code selected by *estimateRegion()* are recorded and used to perform the actual reduction transformations.

A balance between applying local and recursive reductions is achieved by reducing the size of the excessive set in increments of 1. In this way the spill code may be distributed in all regions that have available functional units. More advanced estimation techniques consider the functional unit holes usable by each instruction, instead of assuming all holes in the region can be used by any instruction. These estimations consider only resource holes that have sufficient available time within the execution range of the instructions being moved.

Consider region 1 in Figure 9.3(a) and the corresponding code in Figure 9.5(a). Assume that regions 1, 2, and 3 have execution counts of 10, 3, and 7, respectively and that the architecture provides two functional units and three registers. Option 1 selects `C` for spilling and places the spill code in region 1. The `store` can be placed in the functional unit hole containing `C`. However, there is no hole to hold the `load`, since it must be executed after the `brlt` and before `M`. Thus the execution time of region 1 is increased by 1, giving option 1 a cost of 10. Option 2 places the spill code for `C` in region 3. There are sufficient functional unit holes in region 3 for both the `store` and `load`. Thus there is no increase in the length of region 3's critical path and option 2 has a cost of 0. Option 3 reduces region 3's requirements from three to two by spilling `G`. Since there is a hole available for the `load` but not for the `store` or `load`, the length of region 3's critical path increases by 1, giving option 3 a cost of 7. Option 2 has the lowest cost and is thus selected in this example.

Special handling must be given to the `then` and `else` regions of an `if-then-else` statement when placing spill code in them or considering recursive reduction transformations. Spill code must

be duplicated in both regions. When recursive reductions are considered the amount that each region must be reduced is different if they have different total requirements. Assume that the region containing the `if-then-else` statement has requirements of R, and that the `then` and `else` regions are in the excessive set and have requirements of R_{then} and R_{else}, respectively. The requirements of the region are given by the equation $R = P + max(R_{then}, R_{else})$, where P is the number of values in R that are alive in parallel with the child regions. The child regions must reduced by the amount $subgoal_i = goal - (P + R_i)$.

9.3 Assignment and placement of copy instructions

The allocation phase allocates a sufficient number of registers for each region for all values that it and its subregions compute, but does not indicate which particular registers should be used by each region. The assignment phase must assign the registers so that the total number of registers assigned to all subregions does not exceed the number of registers allocated to the parent region. The values are divided into three types, based on their effect on the overall assignment. The three types are: 1) non-SSA global values, 2) SSA global values, and 3) values local to a single region. Global values are assigned registers first since their assignment affects multiple regions. The SSA global values are assigned after the other global values due to the nature of the ϕ nodes. The ϕ nodes may require more registers than are allocated to the region containing the `defs` and `uses` of the ϕ nodes. When this situation occurs, register copy instructions must be inserted. The insertion of these copies is performed after the other global values are assigned registers so that the instances where copy instructions are required are properly identified. Finally, the values local to each region can be assigned the registers not used by global values.

The assignment phase operates in a bottom-up manner to maximize the sharing of registers whenever allowed. Each region node records several pieces of information about the registers being assigned. `Assigned` is the set of registers that have been assigned to an instruction in the region or in one of its subregions. `Interfere` is the set of registers in ancestor regions that interfere with instructions in the current region.

A previously unassigned register is assigned to a value in a region only when there are no other previously assigned registers that can be used. This approach minimizes the number of registers used by each region and its child regions.

Pseudo code for selecting a register is given in Figure 9.7. The value being assigned a register is identified by the instruction that computes it, say n. Any register in use by an instruction

```
Procedure assignDef( n )
{
      /* find all registers that n's value interferes with */
   used = ∅;
   foreach i ∈ {n} ∪ all uses of n
      used = used ∪ i→region.interfere  ⋃ (j is independent of n)  j.defAssign ∪ j.useAssign;
      /* look for previously registers */
   wRegion = region;
   avail = wRegion.assigned - used;
   while ( avail == ∅  &&  wRegion != NULL )
   {  wRegion = wRegion.dependsOn;
      avail = wRegion.assigned - used;
   }
      /* select a register */
   if ( avail == ∅ )
   {  register = any register not in used;
      add register to region.assigned for all ancestor regions;
   }
   else
      register = any register in avail;

      /* assign the register to the value */
   n.def = n.def ∪ register;
   foreach use u of n
      u.use = u.use ∪ register;

   update region.interfere for all affected regions;
}
```

Figure 9.7: Assignment algorithm

that interferes with n is determined. Values that interfere with n may either be independent of n or be alive in an ancestor region at the time that n's region is executed. The set of registers already assigned in the region is checked to see if it contains any registers that would not interfere with n. If not, the ancestor regions are each checked in turn. If such a register is available it is selected for the assignment. If no such register is available a previously unassigned register is selected. The selected register is assigned to the instruction and recorded in all of the instruction's uses, as well as the affected regions.

The procedure *assignDef()* can be used as given for non-SSA global values and local values. For SSA values the assignment algorithm must determine if register copies are needed and if so, place them to minimize the increase in the overall execution time. This algorithm is summarized in Figure 9.8. The algorithm first determines which registers are available for each def reaching the ϕ node and use reached by the ϕ node. The function *findAvail()* is similar to the availability calculation in *assignDef()*. If all of the defs and uses have a common register available, that register is selected and used for the assignment.

```
Procedure assignSsaDef( n )
{
      /* find all registers that interfere the φ node n */
   used = ∅;
   foreach i ∈ {n} ∪ all uses of n
      used = used ∪ i→region.interfere  ⋃_{j is independent of n}  j.defAssign ∪ j.useAssign;
      /* look for a previously used register */
   wRegion = region;
   avail = wRegion.assigned - used;
   while ( avail == ∅  &&  wRegion != NULL )
   {  wRegion = wRegion.depends_on;
      avail = wRegion.assigned - used;
   }
   foreach i in all defs reaching n
      avail_i = findAvail( i );

      /* select a register */
   ok = ⋂_i avail_i ;
   if ( ok != ∅ )
   {  register = any register in ok;
      add register to region.assigned for all ancestor regions;
   }
   else
   {  foreach i in all defs reaching n
         cost_i = copy_cost( d_i , n );
      possible = ⋃_i avail_i ;
      register = register in possible that minimizes the sum of
                 cost_i for avail_i not containing it;
   }

      /* assign the register to the value */
   n.def = n.def ∪ register;
   foreach use u of n
      u.use = u.use ∪ register;

   update region.interfere for all affected regions;
}
```

Figure 9.8: SSA assignment algorithm

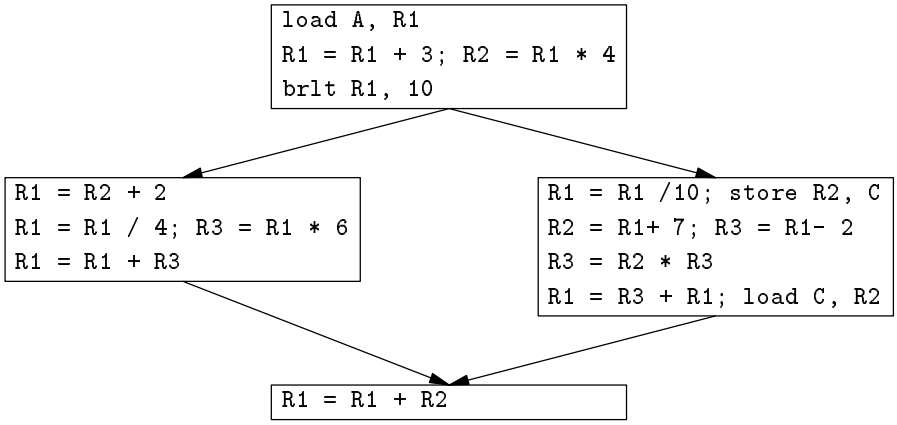


Figure 9.9: Final register assignment

If no such register is available, register copy instructions must be inserted. The cost of inserting a register copy for each def reaching the ϕ node is computed. A register that minimizes the copy costs for those defs that cannot use it is selected and used for the assignment. The defs that cannot use the selected register are assigned alternate registers from their available register sets and copy instructions are inserted at the appropriate locations.

The cost of inserting a copy instruction is computed by the function *copyCost()*, which checks for available functional units in each region between the def and the ϕ node. If an available functional unit is found, the location is recorded, and there is no cost. If no functional unit is available the copy instruction is inserted in a region where both the source and destination registers are available, and the cost is the execution time of the copy instruction times the execution count of the region containing the copy instruction.

Consider assigning the three registers R1, R2, and R3 to the example program in Figure 9.5(c). The first phase assigns registers to the non-SSA global values B and C. There are no previously used registers available, so R1 is assigned to B and R2 is assigned to C. The second phase assigns registers to the SSA global values L_1 and L_2. Both L_1 and L_2 interfere with C but can use R1 or R3. Since R1 has been previously assigned, it is assigned to L_1 and L_2. Since the same register is assigned to both values, no copy instructions are needed. The third phase assigns all remaining values to registers. In region 2, D, E, and F must be assigned registers and interfere with C. D and E are assigned R1 since it was previously assigned in the region, and F is assigned R3. In region 3 the values G, H, I, and K must still be assigned registers. They are assigned R1, R2, R3, and R3 respectively. Finally the values A and M in region 1 must be assigned values. Since neither interferes with any other value they can both be assigned R1. The result is that region 2 uses the two registers R1 and R3, and both regions 1 and 3 use all three registers. The resulting code is shown in Figure 9.9.

Chapter 10

Architectural Considerations

In addition to exploiting ILP, compilers for fine grain parallel architectures must consider other architectural characteristics as well. These characteristics include pipelined functional units, implicit uses of specific registers by some instructions, and instructions and values that may use any of several types of resources. When methods to handle such characteristics are incorporated into a compiler better quality code can be generated through more aggressive resource allocation and assignment. This chapter describes a set of extensions to URSA to exploit these architectural features.

10.1 Instruction Pipelining

The two primary factors that limit the effectiveness of an instruction pipeline are data dependency hazards and collisions, which occur when two instructions attempt to use a pipeline stage at the same time. Data dependence hazards are best addressed during the allocation of registers to data values. Collisions must be considered when assigning instructions to a particular functional unit's pipeline. The problem posed by pipelined multiple issue architectures is that given a set of instructions currently assigned to functional units, is the best assignment of the next set of instructions to functional units must be determined. The algorithm presented in this section first minimizes any increase in the length of the critical path through the DAG, and then minimizes idle time due to waiting for a safe time to issue each instruction. The values computed in this algorithm are also used by URSA's allocation phase when decisions are made on how to reduce an excessive requirements set.

The limitations placed on issuing instructions to the pipeline to preserve the program's semantics are represented as interlocks between instructions in this dissertation. These interlocks delay instructions until they can be executed without altering the semantics of the program. Interlocks are required to prevent improper access to data values, such as trying to read a register before its

value is available, or writing a new value to a register before a previous instruction has had time to read the old value. In addition, an interlock occurs between instructions `A` and `B` if `B` cannot be issued on the first cycle after issuing `A` due to `A` still using a portion of the pipeline when instruction `B` would need it. This permits architectures supporting instructions of different execution times to be considered. The interlock problem requires more analysis in multiple issue architectures, since a ready instruction may experience data related interlocks with several currently executing instructions. In such a situation the instruction must wait for all of the interlocks to be resolved.

DEFINITION 21 *The function $Interlock_D(A, B)$ returns the number of cycles after issuing instruction A that instruction B must wait before it can be issued, to prevent register inconsistencies for registers shared with A. The minimum value returned is 0, indicating that no data dependencies exist between A and B.*

DEFINITION 22 *The function $Interlock_F(A, B)$ returns the number of cycles after issuing instruction A that instruction B must wait before it can be issued on the same functional unit as A to guarantee that all pipeline stages will be available to B when needed. The minimum value returned is 1, indicating that B can be issued on the next cycle after issuing A.*

The problem of assigning of functional units to a set of instructions can be modeled as a weighted bipartite matching problem [FF65]. The source partition of nodes represents the available functional units, while the destination partition of nodes represents the instructions to be assigned. Edges are added between all pairs of functional unit and instruction nodes. Each edge is weighted with the cost of assigning the instruction to the functional unit. The cost represents the increase in the critical path length if the corresponding assignment is made. Let S be the last set of instructions assigned to functional units and T be the next set of instructions to be assigned. Let $length(s, t)$ be the length of the longest path from the beginning of the DAG to the end that executes s followed by t. The function $length(s, t)$ is defined as follows

$$length(s,t) = max(s.start\_time + Interlock_F(s,t), data\_delay(t)) + t.longest\_remaining\_path \tag{10.1}$$

where $s.start\_time$ is the time that s was issued, $t.longest\_remaining\_path$ is the length of the maximum path from t to the end of the DAG, and $data\_delay(t)$ is the earliest time that t can be executed and still honor all data dependencies with other executing instructions. The first parameter of the maximum function computes the earliest time that t can be issued after s on s's functional

unit. The second parameter, $data\_delay(t)$, computes the earliest time that t can be issued after waiting for all of its data dependencies, including any from s, and is given by

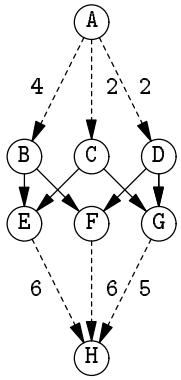
$$data\_delay(t) = \max_{p \in Parents(t)} (p.start\_time + Interlock_D(p,t)) \tag{10.2}$$

As an example, consider the path A, B, E, H in Figure 10.1(a). Notice that $s.start\_time = 4$, $t.longest\_remaining\_path = 6$. Assuming that $\text{Interlock}_D(\texttt{B, E}) = 1$, and all other interlocks for E are less than or equal to 1, then $length(\texttt{B},\texttt{E}) = 4 + 1 + 6 = 11$. The function $data\_delay()$ can easily be extended to include data dependence interlocks from earlier instructions that may reach far enough to affect the instructions currently being assigned.

It should be noted that the length computed by $length(s,t)$ cannot be precise, as all of the instructions both before s and after t would have to be assigned to functional units first to determine the effects of interlocks on the lengths of the paths. Thus, the assignment must proceed from one end of the DAG to the other. The approach taken in this work is to perform assignments from the top to the bottom of the DAG as this aids URSA's handling of value live ranges. Each subsequent matching attempts to balance the lengths of the paths through the DAG; therefore, the impact on the length of the critical path from having to compute assignments in a successive manner is not expected to be significant.

A weighted bipartite graph is created by adding an edge from each node in S to each node in T and weighting each edge (s,t) by $length(s,t)$. A matching of a node s_i to a node t_j represents an assignment of s_i and t_j to the same functional unit. The goal of a *minimized weight* matching algorithm is to find a maximum matching that has a minimum sum of weights on the matching edges. However, the goal of this work is to find a maximum matching that also minimizes any increase in the length of the critical path through the DAG. Since the weights in this work represent the length of the path resulting from the matching, this problem is referred to as the *minimum path increase* maximum matching problem. Well known algorithms exist for finding a minimum weight maximum matching. However, these algorithms do not necessarily find a matching that has a minimum largest weight; it is possible that the minimum weighted matching includes the largest weighted edge.

A new algorithm that solves the minimum path increase maximum matching problem has been developed as a part of this work. The solution has the same form as the minimum weighted matching solution; only a supporting procedure and cost function need to be modified. In these algorithms, the matching problem is represented as a network flow problem with each edge in the bipartite graph having unit flow [Tar83]. The function $length()$ as defined above is used as the basis for the cost function for the edges.



(a) Portion of a DAG

	B	C	D
E_5	1, 1, 11	1, 0, 11	0, 2, 11
F_5	1, 2, 12	0, 0, 11	1, 2, 11
G_3	0, 4, 13	1, 6, 13	1, 5, 12

(b) Table of interlocks and lengths

Figure 10.1: Pipeline Example

```
procedure RelaxShortest( u )
{  foreach v ∋ (u, v) ∈ E
   {  newLen = u.shortest + cost(u, v);
      if ( v.shortest > newLen )
      {  v.shortest = newLen;
         v.shortestPredecessor = u;
      }
   }
}
```

(a) shortest paths

```
procedure RelaxMinimumIncrease( u )
{  foreach v ∋ (u, v) ∈ E
   {  newMax = max( u.max, mcost(u,v) );
      if ( v.max > newMax )
      {  v.max = newMax;
         v.maxPredecessor = u;
      }
   }
}
```

(b) minimum increase paths

Figure 10.2: Relaxation procedures

The minimum weighted matching algorithm operates by finding a shortest path to any unmatched node $t \in T$ for each node $s \in S$. The shortest path algorithm computes the shortest path from s to all other nodes in the graph. The algorithm maintains estimates of the shortest path to each node. These estimates are updated when shorter paths are found. Once the shortest path from s to u has been determined, the algorithm checks each v that is adjacent to u, to determine if the path through u to v is shorter than the previous shortest path to v. The procedure that checks this condition and updates the path information as needed is shown in Figure 10.2(a), where $cost(u, v) = legnth(u, v)$. This process is referred to as *relaxing*. The shortest path length estimate to `v` is recorded in `v.shortest`.

The minimum path increase matching problem requires the maximum weight of an edge in the path to be found, as opposed to the sum of all edges in the path used for the minimum weight

matching problem. The cost function used for the minimum path increase algorithm is defined as

$$mcost(u,v) = \begin{cases} length(u,v) - cp & \text{if } length(u,v) > cp \\ 0 & \text{otherwise} \end{cases} \tag{10.3}$$

where `cp` is the length of the current critical path through the DAG. The corresponding relaxation procedure for the minimum path increase algorithm is shown in Figure 10.2(b). The maximum increase estimate is recorded in `v.max`. With these modifications to the cost function and relaxation procedure, the minimum weight maximum matching algorithm can be used to solve the minimum increased path maximum matching problem. The minimum increased path algorithm operates similarly to the shortest path algorithm, requiring a priority queue for selecting the next minimal node to relax and updating the priority of each `v`. If Fibonnaci heaps are used, the minimum increase path matching algorithm requires $O(nlogn + m)$ time per source node s. This gives an overall time of $O(n^2logn + nm)$ since a minimum path must be found for each $s \in S$.

As an example, consider the portion of a DAG in Figure 10.1(a). The dashed lines represent arbitrary paths. The labels on the edges from `A` represent the earliest start times of the respective nodes, while the labels on the edges to `H` represent the longest paths to `H` for the respective nodes. The solid lines represent data dependencies between the two sets of nodes. The table in Figure 10.1(b) shows 3-tuples for each possible edge in the bipartite graph constructed from the nodes currently being considered. The subscripts on `E, F`, and `G` indicate the earliest start times of the instructions as computed by $data\_delay()$ from the data dependence interlocks. The first number of each tuple is the $\text{Interlock}_D()$ for the possible matching. The second number of each tuple is the $\text{Interlock}_F()$ value for the edge, and the third number is the resulting length if the matching is used. Thus, `G` should be matched with `D` to minimize the longest path. Both nodes `E` and `F` can be matched to either of nodes `B` and `C` since they cannot affect the critical path length. However, the matchings `E-B` and `F-C` result in a lower total weight, which is advantageous to other phases that may interact with URSA.

It should be noted that it is still advantageous to minimize the sum of lengths in the matching for edges that do not increase the critical path, as this represents less time that the functional units are idle due to interlocks. The increased useful idle time may be used by URSA's reduction transformations or migration of parallel instructions across DAG boundaries as performed by percolation scheduling [AN88], region scheduling [GS90], Global Resource Spackling, or other global scheduling methods [BR91, SHL92]. The minimum increase path algorithm can be augmented to minimize the weights of non-increasing edges by using $length(u,v)$ as secondary key in the priority queue and adding the shortest path relaxation logic to `Relax_Minimum_Increase()` when `v.max = new_max`.

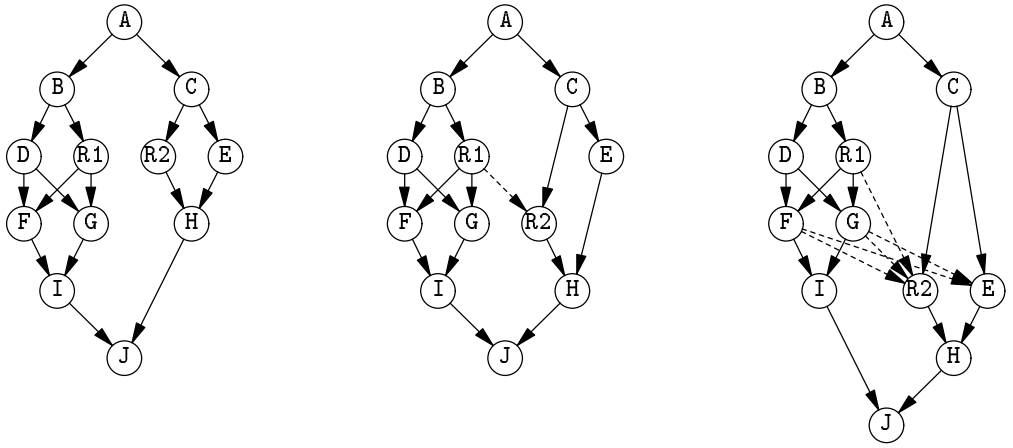
In the description of URSA the allocation and assignment of resources are treated as separate phases. However, the assignment of resources may impact allocation decisions. In practice available assignment information is used during allocation to determine the best spackling transformation to apply. As an example, consider three equal instructions, `A`, `B`, and `C`, to be allocated to two functional units. If `C` has a larger interlock times for both functional units, then `C` should be the instruction that is sequenced after the others, allowing useful instructions to be scheduled during `C`'s interlock time. Therefore, when considering spackling transformations the interlock information is used to select instructions for non–spanning resources and to decide how to split the sets of uses for spanning resources. Interlock delays are factored into the path lengths under consideration.

10.2 Modeling Architectural Constraints

While the integration of pipelining into URSA requires the computation of additional path length information, the integration of implicit uses and generic requirements requires additional resources. In this section two methods for representing restrictions on instruction scheduling as additional resource types are discussed. The methods handle requirements for two cases: specific copies of a resource required by an instruction, and selection from several types of resources for generic resource requirements. Both methods use additional classifications of resources to represent the restrictions as resource requirements, and thus allow URSA to allocate them as required. The first method uses a special subclass resource for architectural components, such as registers, which are sometimes implicitly required for some instructions, but may be used as general purposes registers when such instructions are not being executed. The second method uses a generic resource that includes all resource types that may be used to meet a requirement.

10.2.1 Reserved Resource Copies

In some architectures, particular instances of a resource are always used by some instructions and can also be used for general purposes. For example, the VAX instruction set includes the instruction `MOVC3`, which copies a string from one location to another. The `MOVC3` instruction uses registers `R0-R5` during its operation, overwriting any previous values held in them. The particular instances of a resource required by such an instruction are referred to as *reserved copies* and the instructions as *reserved copy instructions*. The values in `R0-R5` are results that may be used by subsequent instructions. These uses of the results are marked as reserved copy instructions as well. In the absence of instructions such as `MOVC3`, registers `R0-R5` can be used as a general purpose registers.



(a) before reductions (b) after reserved reduction (c) after register reduction

Figure 10.3: DAG with reserved copy instructions

To address the problem of reserved copy uses, an instruction scheduler must guarantee that the scheduled code has two properties: 1) there are never conflicting reserved copy instructions scheduled concurrently, and 2) no live values are destroyed by reserved copy instructions. The first property can be satisfied by adding a new resource to URSA that represents the reserved copy and indicating which reserved copy instructions require the resource. This technique can be generalized to multiple sets of reserved copies. The second property can be handled simplistically by never allocating general purpose uses to the reserved copies. However, this approach makes inefficient use of the resource.

To efficiently use resources that contain reserved copies, the technique presented treats the reserved copy as a subclass of the general purpose resource class. URSA first performs allocations for just the reserved copy subclass. URSA then performs allocations for the general resource class, which includes the reserved copies. In addition to their normal requirements, the reserved copy instructions are considered to require one copy of the general resource for each reserved copy. The assignment phase then considers the reserved copy to be available for assignment to all instructions except those that will execute in parallel with a reserved copy instruction, *i.e.,* if an instruction, `I`, can be executed concurrently with a reserved resource instruction, `I` is assigned any copy from the set of available non-reserved copies of the resource; otherwise `I` is assigned any copy from the set of all available copies of the resource.

As an example, consider the DAG in Figure 10.3(a) and a target architecture with four registers, one of which is a reserved copy register. The DAG has two reserved copy instructions, `R1` and `R2`, which can be executed concurrently according to the data dependencies. Since no increase in

critical path length occurs, the reserved copy register resource reduction transformation sequences R2 after R1, as shown in Figure 10.3(b). After this reduction, the set {D, R1, F, C, E} is an excessive register set of size five. Thus, a general register resource reduction transformation is applied, with the result shown in Figure 10.3(c).

10.2.2 Generic Instructions

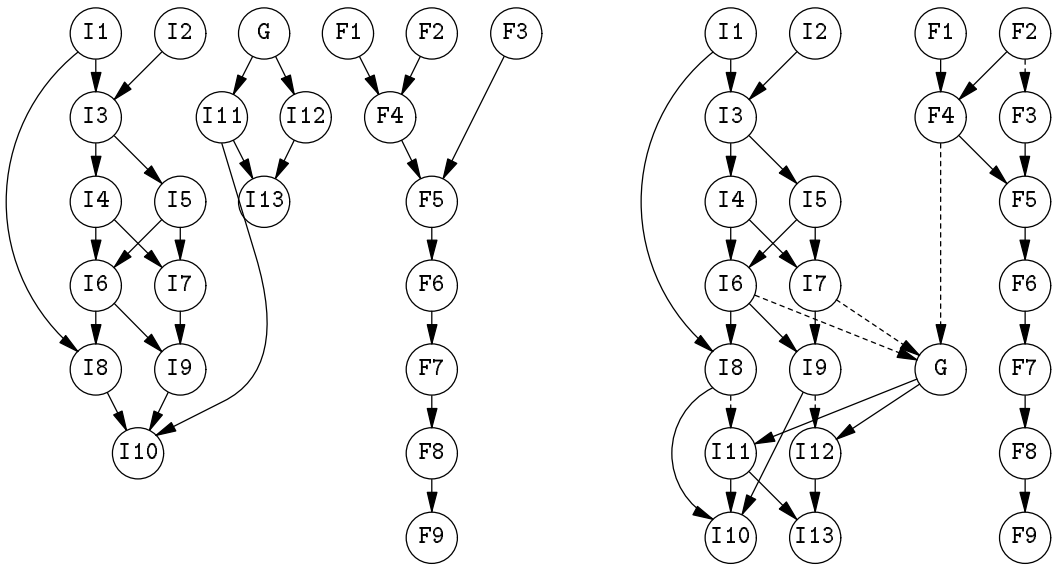
In architectures with multiple types of a resource, such as integer and floating point functional units, some resource requirements can be fulfilled by any one of several types of a resource. For example, some architectures allow moves to be performed by either integer or floating point functional units. Instructions that can be executed by one of several types of functional units are called *generic* instructions [ATGLR93]. The technique presented generalizes this concept to *generic requirements* for resources; an instruction may have specific register type requirements but generic functional unit type requirements, or vise versa.

Generic requirements should be scheduled on whatever compatible resource is available. When there are no compatible resources available, the generic requirement is a member of an excessive set. In this case, the reduction transformations select the resource type that the generic requirement will use. The reduction transformation should select the resource type that, when its uses are sequentialized, results in the least increase to the length of the critical path through the DAG.

The approach taken to extend URSA is to create a new resource type for each type of generic requirement. A requirement's generic resource type is the union of all of the resources that can fulfill the requirement. Instructions that require a number of a specific member resource are considered to also require the same number of the generic resource. The generic requirements of an instruction are represented in the generic *Reuse* DAG and not in any of the member *Reuse* DAGs.

The assignment of an instruction with generic requirements to resources usually occurs in the assignment phase. However, if the instruction is in an excessive set, URSA's reduction transformations effectively make the assignment decision. The reduction transformation selects the member resource that results in the least increase in path lengths when introducing sequentiality.

As an example, consider the DAG in Figure 10.4(a) and a target architecture with two integer and two floating point functional units, and four integer registers. Nodes labeled In require an integer functional unit, nodes labeled Fn require a floating point functional unit, and the node G is a generic instruction that may use either type of functional unit and generates an integer value. The set {I1, I4, I5, I6} requires four integer registers concurrently, so the values in the subDAG rooted at G cannot straddle this set without causing excessive integer register requirements. The



(a) before reductions

(b) after reductions

Figure 10.4: DAG with a generic requirement

instruction `F3` must be delayed until either `F1` or `F2` has been issued, so that both floating point functional units are in use for the first two issuing cycles. A list based scheduling method may try to schedule `G` on an integer functional unit in parallel with `I3`; however, as noted, this would create an integer register excessive set that would require spills to resolve it. URSA's reduction transformation is able to determine that delaying `G` until after the register requirements of {`I1, I4, I5, I6`} and assigning `G` to a floating point functional unit avoids the need for spills. This reduction does not increase the length of the schedule. The result of the floating point reduction followed by the generic reduction is shown in Figure 10.4(b).

It is possible that the different resources that can fulfill a generic requirement will have different costs in terms of execution time. If there are several available resources, the assignment phase can select the resource with the minimum cost. If the generic requirement is in an excessive set, then the reduction transformation selects a member resource and uses the cost of the selected resource when determining the impact on path lengths. If the reduction transformation considers several possible resources for the generic requirement, it can use the respective costs when comparing resulting path lengths to make a final assignment determination.

There are two possible approaches to applying reduction transformations. The first is to reduce only one resource per transformation. The second is to reduce multiple resources in a single transformation. The first approach may result in less efficient schedules since the separate transformations may impact each other. The first transformation may sequentialize one instruction

that a second instruction in the second excessive set is dependent on, resulting in a larger than necessary increase in path length after the second transformation.

In the second approach a single reduction transformation reduces several resources at once. The selection of the subsets used for sequentialization considers each instruction's impact on all of the resources being considered. The selection process can then look for instructions that when sequentialized, would free up any of the resources that can fulfill the generic requirements.

Chapter 11

Implementation

As a part of this research a prototype implementation was performed. This chapter discusses the design and implementation of the measurement and spackling algorithms and intermediate representations used in URSA and Resource Spackling.

URSA and Resource Spackling require several new data structures to represent the additional information they compute and use. These data structures have a symbiotic relationship with the host compiler's intermediate representation. To adequately maintain this relationship proper interfaces between the compiler and URSA are required. Since the implementation is a research prototype, data structures and algorithms were designed to ease the task of porting the implementation.

One of the goals of the implementation was to identify the algorithms and structures that are critical to creating a useful and practical system. As a result, specialized graph data structures were created to support efficient traversals and compute and store GURRR's information in a hierarchical manner. One of the algorithms most critical to URSA's performance is the bipartite matching algorithm used to compute allocation chains. Extensions to the intermediate graph representation were created to support a specialized implementation of this algorithm.

11.1 URSA Interfaces

The allocation and assignment phases addressed by URSA interface and interact with a number of other components in a compiler. The GURRR computed and used by URSA must be constructed from the preliminary intermediate representation generated by the front end. The result of URSA must be passed on to the code emission phase. URSA must also interface with other portions of the compiler, such as the symbol table and the application of transformations not written to use GURRR. Finally, it is intended that URSA should target a variety of architectures with different features and characteristics.

The implementation of URSA was designed to be modular with respect to the rest of the compiler in two different respects. First, URSA is designed to be a module within the compiler, *i.e.,* URSA can be inserted in a host compiler to provide the allocation and assignment of resources. The host compiler used for the prototype implementation is `pdgcc`, the University of Pittsburgh's research compiler which generates a PDG based intermediate representation from C source code [Fie92]. Second, URSA is not dependent on a particular host compiler. A formal interface is used between the host compiler's and URSA's structures and algorithms. Thus URSA is not directly dependent on the host compiler's structures, and in fact, large portions of the implementation can be used on alternatives to a PDG based GURRR. There are several advantages to this approach. First, the implementation is more portable; only a small piece of interface code must be written to rehost URSA to a different compiler. Second, the implementation of the algorithms can exploit a set of structures designed for the efficiency of URSA.

URSA requires detailed information about the target architecture to determine the cost of potential allocations and assignments, as well as what types of global code motion can be performed. To support flexible targeting of URSA, configuration files are used to describe the architecture. The configuration file contains the following types of information

1. Resource types, *e.g.,* functional units and register banks
2. Generic resource classes
3. Number of each type of resource
4. Architectural features, such as support for predicated and/or speculative execution
5. Description of the instruction set

The description of each instruction includes the resource types used, including implicit uses, and the number of cycles required to execute it. The grammar used for the configuration file is easily extensible to allow for additional characteristics in the future.

The implementation was designed to load the configuration information at run time. While this approach implies slight execution inefficiency due to some extra levels of dereferencing and more dynamic memory allocation, it is expected that the benefits outweigh the cost. This approach allows testing and experiments to be run on a variety of of target architectures without having to rebuild the compiler each time. In addition, compile time configuration mechanisms are more complex to implement, as well as cumbersome to maintain and extend.

11.2 Representation

This section describes the implementation of the Global Unified Resource Requirements Representation (GURRR). Fundamentally, GURRR is a hierarchical graph consisting of nodes and edges, each labeled with information. Several types of nodes and edges exist to represent the instructions, regions, and special summary information. In addition, the representation contains structures for resource holes and allocation chains. Each structure and design decision is described in turn.

The implementation of GURRR is designed to provide efficient support for the major operations performed on it. The first operation performed by URSA is the computation of the resource requirements information, and accounts for the majority of execution time. The second operation performed by URSA is the set of graph transformations to reflect the allocations made. Many of URSA's algorithms involve graph traversals and computing information using bit vector operations. In addition, information about the program represented is placed in conveniently accessed locations.

In GURRR, as in the PDG, groups of instruction nodes that have common control dependencies are identified by *region* nodes that summarize the control conditions. However, in GURRR, region nodes are also incident on summary data and temporal dependencies. With respect to the various types of edges present in the graph, region nodes are no different than instruction nodes. While region nodes must carry label information describing the region it represents, they must also carry nearly all of the label information on instruction nodes. Thus, in the implementation there is a single node structure used for both instruction and region nodes. The node structure contains all label information needed for instruction nodes. Two additional fields are added: a type tag and a generic data pointer. The tag field indicates whether the node is a region or summary node. The data pointer points to the additional label information required when the node is a region node.

The algorithms as described and implemented assume that each node in the graph begins the use of at most one spanning resource. This assumption is critical to the proper computation of excessive sets as the live ranges for multiple spanning uses begun by a single node may have different live ranges. Such defining nodes occur in two situations: 1) instructions defined by the target architecture, and 2) region summary nodes representing regions that need multiple instances of the spanning resource. To enable the algorithms to handle these situations special nodes, called *single definitions*(SDEFs), are inserted in the representation. An SDEF node, S_{I_j}, is inserted in the graph for each spanning use started by an instruction I. All dependence edges sinking on I are copied to S_{I_j}. All temporal dependences sourcing from I are also copied to S_{I_j}. However, only the data dependences for the j_{th} unique spanning use began by I are copied to S_{I_j}. The result is that the live ranges are properly captured by the S_{I_j} nodes. The measurement algorithms ignore instructions

I that begin multiple spanning uses while the Resource Spackling algorithms ignore the SDEF nodes.

Instructions that use multiple instances of a resource must also be addressed in the *Reuse* DAG approach to decomposition. In a bipartite representation an additional pair of nodes can be added for the instruction for each additional instance of the resource needed. The *Reuse* DAG approach keeps track of the maximum number of matching edges that can be incident on a node. For example, if an instruction uses three registers, that instruction's node can have up to three incoming and three outgoing matching edges. This technique trades off the cost of node creation during the decomposition for the cost of some extra bookkeeping.

Because many computations on GURRR can be done efficiently as bit vector operations, each node is assigned a unique integer id. A global indexing array of pointers to nodes is used to convert an integer identifier to a pointer to a node. Region nodes have two integer ids: 1) an identifier assigned to the representing node, 2) an identifier is is assigned for the region. Each type of identifier has a separate index array of pointers to type specific label information. Although not really necessary, a separate region id simplifies handling region information, tracking of regions through transformations, and reduces memory demands.

This approach gives great flexibility to the representation of edges. There are a number of types of edges used in GURRR, including, data, control, and temporal dependencies. In addition, there is a unique reuse edge type for each resource being measured, producing multiple reuse subgraphs. For discussion purposes these edges are commonly grouped in several types of subgraphs. However, in the implementation of the representation, all edges are treated in a similar manner, allowing a single set of edge manipulation routines to be used. Each type of edge is assigned a unique number, called an *edge set*. Each node has an array of edge set information. The edge routines are informed of which edge set to operate on by passing the appropriate array index. An alternate approach would be to declare a separate object in the node structure for each edge type. Using an object oriented language, such as C++, stronger type checking could be enforced. Such an approach has two drawbacks. First, some loss of generality is incurred. There are several situations where traversals of multiple types of edges must be performed, *e.g.,* a topological traversal of data and temporal dependencies to compute the full partial ordering of instructions in a region. The representation implemented easily supports this requirement by iterating over a bit vector set of the desired indices. Second, an array for reuse edge sets must still be used to support run time configuration of the number of types of resources.

In the implementation there are two ways in which an edge can be represented, depending on whether or not labels are present. The first edge representation, is a bit vector, and is used

when there are no labels on the edges. Each node has a pair of vectors for each edge set; one for vector for incoming edges and one for outgoing edges. A bit vector edge is added by inserting the corresponding node's identifier in the respective bit vectors for both the from and to nodes. Since several computations used in constructing GURRR required node ancestor and descendant information, the bit vectors in the appropriate edge sets double as data fields for these computations. This representation incurs less overhead by avoiding unnecessary memory management for storing edge characteristics.

The second edge representation is for labeled edges. For all edges incident on the same pair of nodes a linked list of label information is created. Each entry in the list is a data structure containing the label information. Both nodes incident on the edge contain a pointer to the shared data in their linked lists, allowing consistent access of the label data from either node. To support the node relative computations, labeled edges are also recorded in the edge set bit vectors.

In practice, all edges of a particular edge set type will either be labeled or not. Thus, one set of edge manipulation routines is used for both labeled and unlabeled edges. The routines automatically determine the type of representation to use based on the edge type. Optional additional parameters to the routines allow overriding of the label type.

In addition to using edge sets to represent dependence information, edge sets are also used to represent *Reuse* DAGs. However, additional information is needed for each resource used by the node. This information includes which immediate ancestors of the node are reusable, and which values the node kills for spanning resources. An array of resource usage information is allocated for each node and indexed by the same edge set index used for the edge set array.

Allocation chains also use a dual representation to facilitate several different access methods. Bit vector sets representing the nodes on each allocation chain are used for computing excessive sets and determining the number of allocation chains covering interesting collections of nodes. Linked lists of nodes on each chain are used by algorithms that need to traverse all nodes in particular chain in the order dictated by their dependences. Such algorithms include the identification of resource holes.

In the description of GURRR in Chapter 5 resource holes are as a normal graph nodes, which are incident on edges. Conceptually holes are a separate type of node from instruction/region nodes; holes have different label information and are incident only on temporal edges. Thus, in the implementation, hole nodes are not inserted in the graph. Instead, each allocation chain contains two linked lists, a list for instruction nodes and a list for hole nodes. The nodes in each list are stored in order from the beginning to the end of the region's DAG. Each hole records which instruction nodes

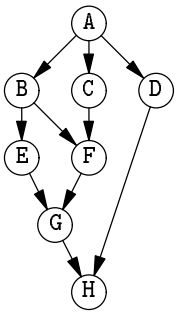
it occurs between. Instruction nodes contain ancestor and descendant resource hole bit vectors to aid the computation of holes available for a fill instruction.

11.3 *Reuse* DAG Decomposition

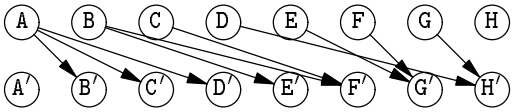
The computation of allocation chains depends on the decomposition of the partial ordering of a *Reuse* DAG, which is performed using a bipartite matching algorithm. The construction of a separate bipartite graph each time allocation chains must be computed is too time consuming to provide a practical benefit. Therefore, the matching algorithm is adopted to the existing graph based representation of dependences.

In the original description of using bipartite graphs to compute chain decompositions a pair of nodes i, i' is created for each instruction i in the *Reuse* DAG. Let P and P' be the two partitions, then i is placed in P and i' is placed in P' [FF65]. For each pair of related instructions (i, j) in the *Reuse* DAG an edge is added from i to j' in the bipartite graph. Edges sourcing in P and sinking in P' represent nonmatchings. Edges in the other direction, sourcing in P' and sinking in P, indicate matchings between the nodes. The matching algorithm then adds matchings by finding *augmenting paths* using these edges, *i.e.,* a path is found that starts in the first partition, ends in P', and alternates between the partitions. The direction of the edges used in the path are then reversed, changing the unmatched edges to matching ones and the matched edges to nonmatching ones. As a result of the properties of the path, the number of matching edges increases by exactly one when the directions are reversed. An example is shown in Figure 11.1. The original *Reuse* DAG is shown in Figure 11.1(a) and the corresponding bipartite graph is shown in Figure 11.1(b). Two unit length augmenting paths are found from A to B' and from B to F'. The bipartite graph resulting from the reversal of the edge is shown in Figure 11.1(d). Now the alternating path C, F', B is found. The result after reversing these edges is shown in Figure 11.1(f).

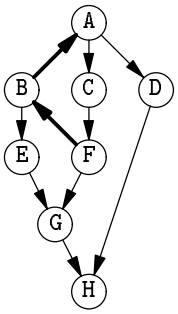
After the *Reuse* DAG is constructed the transitive closure of the edges is computed in the resource's edge bit vectors to obtain the full partial order, *i.e.,* all transitive edges are added to the *Reuse* DAG. The resulting DAG has the same set of edges as the bipartite graph. The pairs of nodes in the bipartite graph are represented using the single node in the *Reuse* DAG. The matching algorithm as implemented finds augmenting paths on the *Reuse* DAG and records the matching information in the resource information structure instead of constructing a separate bipartite graph. Instead of physically reversing edges indicating matches, the matches are recorded in a separate field in each edge set structure. Augmenting paths are found using a special bidirectional traversal



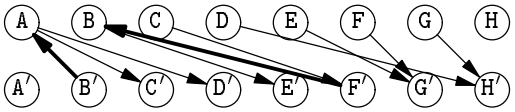
(a) *Reuse* DAG



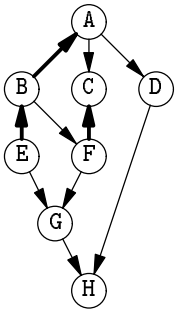
(b) Corresponding bipartite graph



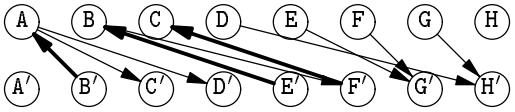
(c) Path B to F



(d) Path B to F



(e) Path C to F to B



(f) Path C to F to B

Figure 11.1: Chain decomposition steps

algorithm that consults both the reuse edges and the match fields. Alternating paths on the *Reuse* DAG can start at any node, must start and end with nonmatching edges, and alternate between nonmatching and matching edges. Figure 11.1 contains both the *Reuse* DAGs and bipartite graphs for each step of the example discussed above. The stipulation that valid augmenting paths must alternate between nonmatching and matching edges preserves the bipartite nature of the problem. In Figure 11.1(c) the path from C to A, C, F, B, A. However, the path traverses two edges of the same type in a row and thus is not an alternating path.

Chapter 12

Experimentation

To assess the practical application of resource requirements measurements and spackling concepts in the allocation of resources, several sets of experiments were performed as a part of this work. The experiments compared the performance of a variety of base and hybrid resource allocation techniques. The techniques differ in the type of integration used, as well as in the methods used to measure requirements and perform reductions. The base techniques consist of register sensitive scheduling, scheduling sensitive register allocation, and the unified resource allocation method described in the previous chapters. The hybrid techniques incorporated URSA's resource requirements information into the register sensitive scheduling and scheduling sensitive register allocation techniques. In addition to comparing the quality of code generated by the various techniques, statistics were collected to assess the practicality and benefit of computing register requirements information.

This chapter describes and discussed the various experiments performed. First the design of the experiments performed are described. Descriptions of the base techniques are given next. The presentation of each set of experiments, including variations and hybrids of the base techniques, are given next. Finally, conclusions are drawn from the data collected.

12.1 Experimental Design

A total of 22 benchmark programs were used in the experiments performed. The programs and some statistics concerning their resource requirements are listed in Table 12.1. The loops listed are thirteen of the first fourteen loops from the Livermore Loops benchmark suite.

Profile information was collected for each benchmark by instrumenting each region with a unique counter and compiling the benchmark using the `gcc` compiler. The resulting execution counts were recorded to a file and used in the performance analysis of the techniques implemented. In addition, the execution counts were used as annotations in the source code compiled by the prototype

	functional units		registers	
benchmark	maximum	average	maximum	average
bubble	5	2.12	6	2.73
hanoi	9	4.12	7	3.50
heapsort	13	3.67	26	5.88
intmm	5	2.38	7	2.86
nsieve	27	4.29	24	5.35
perm	6	2.23	6	2.17
puzzle	25	2.68	10	3.32
queens	5	1.96	5	2.19
quick	6	2.13	7	2.85
loop1	5	3.00	7	2.86
loop2	11	3.71	12	3.57
loop3	4	2.57	4	2.43
loop4	6	2.78	6	3.12
loop5	13	3.71	14	4.00
loop6	13	3.11	14	3.38
loop7	11	3.33	13	3.75
loop9	12	3.78	31	6.50
loop10	21	4.00	32	5.75
loop11	4	2.71	5	2.83
loop12	4	2.29	4	2.17
loop13	16	2.88	27	4.00
loop14	11	3.09	16	3.80

Table 12.1: Benchmarks used for experimentation

compiler. Thus, these annotations were made available to the techniques during compilation in the GURRR data structures.

To gauge the performance of the various allocation heuristics under varying levels of resource pressure, eleven different architecture configurations were targeted. The architectures consisted of eleven of the twelve combinations of 2, 4, and 6 functional units and 4, 8, 16, and 32 registers. Since each operation performed requires at least one operand, and most typically use two operands, the architecture with 6 functional units and only 4 registers is considered impractical. In addition, architectures with a single functional unit and the four sizes of the register file were used as a base for comparisons. For all architecture configurations memory access instructions are assumed to execute in 2 cycles, while all other instructions are assumed to execute in one cycle.

Estimated execution times for a given compilation method were computed using the formula

$$totalCycles = \sum_{r \in Regions of B} r.cpl \times r.executionCount \tag{12.1}$$

where B is the benchmark program, $r.cpl$ is the length of the region's critical path, and $r.execCnt$ is the region's execution count. The speedup for a heuristic is computed by dividing the base estimated execution time by the heuristic's estimated execution time. In most cases the base execution time is the execution of the benchmark program on a single issue architecture compiled using the base

P-RIG technique. When comparing variations of a particular technique, all variations are compiled for the same architecture and one is selected as the base for the speedup calculations. The graphs plot the unweighted average speedup for all benchmarks in the test suite. The individual speedups for each combination of benchmark, heuristic, and architecure are listed in appendix B.

12.2 Overview of the Algorithms

The experiments performed in this work considered three approaches to integrated resource allocation and instruction scheduling. The first approach is register sensitive instruction scheduling, based on Goodman and Hsu's integrated prepass scheduler [GH88]. The second approach is schedule sensitive register allocation, based on Norris and Pollock's parallel register interference graph [NP93]. The final approach is the unified resource allocation technique developed in this work. All of these techniques were implemented using GURRR as the intermediate representation and explicitly exploit its hierarchical properties. This section highlights the features of each major technique in terms of the Measure and Reduce paradigm. Implementation details of these base and hybrid techniques considered are discussed in the subsequent sections.

Integrated Prepass Scheduler (IPS): This algorithm performs on-the-fly detection and reduction of excessive resource demands. Instruction allocation is handled by list scheduling. Excessive functional unit demands are detected when the size of the ready list exceeds the number of functional units available. Excessive functional unit demands are reduced by delaying instructions not selected for execution in the current cycle by the priority function. The detection of register excessive demands is performed by tracking register pressure during scheduling. When excessive register demands are detected, the priority function favors instructions which reduce the register pressure. The effect is that new live values are sequentialized after previous values have been killed. However, sequentialization alone cannot always prevent register pressure from exceeding the maximum number of registers available. Therefore, a subsequent register allocation pass is used to perform spilling to further reduce register demands.

Unified List Scheduler (ULS): This algorithm was developed in this work and is a modification of IPS to eliminate the need for a separate spilling postpass. ULS uses an on-the-fly *live range splitting* reduction technique. This technique is invoked when IPS would otherwise schedule an instruction that would cause the register pressure to exceed the number of available registers. This approach requires a hierarchical representation of the program and performs resource allocation in a bottom up manner. In this way, the register requirements of lower levels

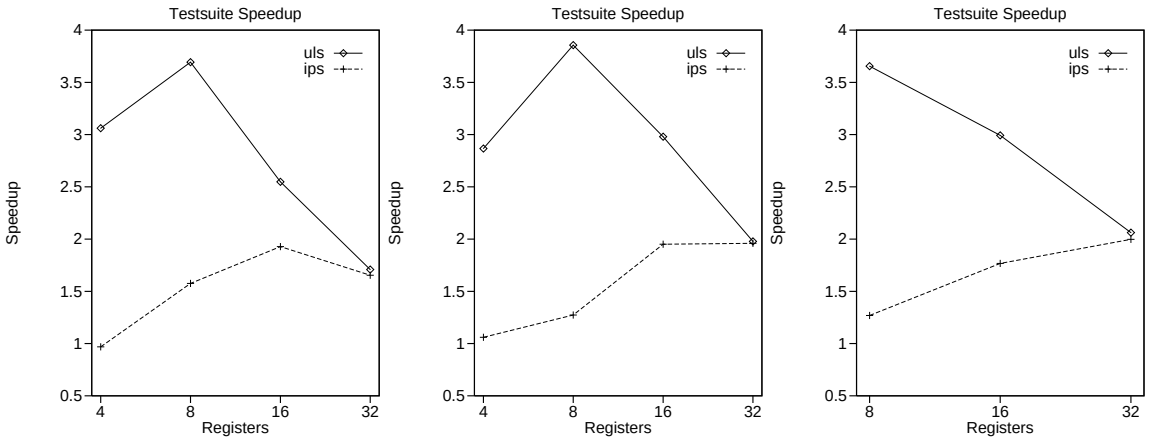
are accounted for in summary nodes and global register allocation is achieved. Because this technique operates hierarchically, global allocation is achieved and the need for a postpass register spilling phase is eliminated.

Parallel Register Interference Graph (P-RIG): When register allocation is performed prior to instruction scheduling for ILP architectures the instructions are only partially ordered. If there are no dependence between instructions incident on separate live ranges, it can not be determined if the live ranges will overlap in the final schedule. This problem does not occur when the instructions are fully ordered. Therefore, an alternate version of the register interference graph, the parallel interference graph (P-RIG) is computed to capture the additional potential live range conflicts. The P-RIG is then used to detect excessive demands for registers. The simplified P-RIG is used to detect excessive register demands. The reduction of these demands is achieved through sequentialization of instructions and spilling of live ranges. Separate cost computations are used in the priority function for the respective reduction methods. As a result of the sequentializations introduced by both register reduction methods some of the functional unit excessive demands also may have been reduced. Any remaining excessive functional unit demands are identified and reduced by a postpass instruction scheduler.

Unified Resource Allocation (URSA): This approach uses the techniques developed in this dissertation to detect and reduce excessive resource requirements. Excessive resource requirements are identified by computing excessive sets for both registers and functional units. Reductions in resource demands are performed using the resource spackling transformations. Like ULS, resource allocation is unified in that all resources are allocated simultaneously. Resource spackling reductions for registers perform both sequentialization and live range splitting, depending on how the uses of the interfering definitions are partitioned into the sets used by the transformation.

12.3 Register Sensitive Schedulers (RSS)

The first set of experiments performed investigated the performance of the register sensitive instruction scheduling (RSS) techniques of IPS and ULS. This section describes implementation details and experimental results of both the base and hybrid algorithms.



(a) 2 issue architecture (b) 4 issue architecture (c) 6 issue architecture

Figure 12.1: Comparison of base RSS techniques

12.3.1 Base Techniques

As described above, IPS and ULS are based on a list scheduling algorithm. As in typical list scheduling algorithms, the height of ready instructions are used in the priority function that selects the next instruction to schedule. However, in IPS and ULS register pressure is also used. The register pressure is measured by computing the number of values alive as each instruction is scheduled. The change in register pressure, P_δ, for a candidate instruction, I, is given by

$$P_\delta = I.defs - I.kills \tag{12.2}$$

where $I.defs$ is the number of new values defined by I and $I.kills$ is the number of live values killed by I. A live value is killed by I if I is the last use of the value to be scheduled. By adding P_δ for the instruction selected to the current register pressure the algorithms have a precise measure of the number of registers required at each point in the committed schedule.

In both IPS and ULS the register pressure value is compared to threshold values to determine which of several priority functions are used to select the next instruction to schedule. In IPS a single threshold, 80% of the number of available registers, is used. When register pressure is below this threshold the priority function selects the instruction with maximum height. If several instructions have the same height, the priority function selects the instruction with the minimum P_δ value. When register pressure is at or above the threshold the priority function reverses the ordering of the two values. That is, the function selects the instruction with the minimum P_δ value. In the event of a tie the instruction with maximum height is selected.

ULS differs from IPS in that it adds a second threshold for register pressure and splits a

live value when this threshold is reached. The splitting threshold is set at the number of available registers, since exceeding this value results in generated code that cannot be executed. When the register pressure reaches this threshold an alternate reduction technique is used. The priority function invoked in this case selects a value to split rather than an instruction to schedule. The value selected is the one whose earliest remaining use has a minimum height. Thus, this priority function selects the value that can be delayed for the longest time. A store instruction for the value being split is injected into the ready list. A corresponding load instruction is injected into the not ready list, with a temporal dependence on the store instruction. All unscheduled instructions which require the split value are delayed to use the load's definition of the value instead. Since the injected store instruction reduces register pressure, it is guaranteed to be the next instruction selected by the pressure reducing priority function on the next iteration of the list scheduler.

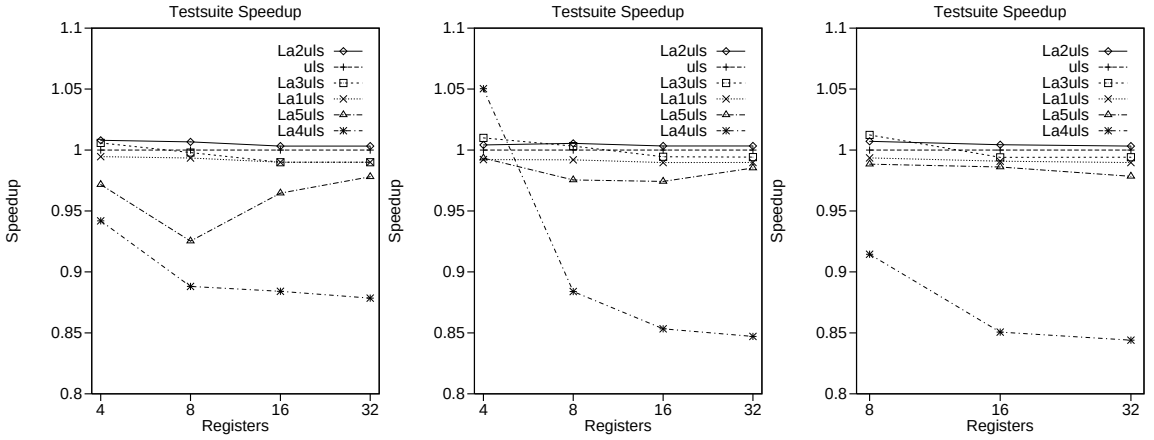
The results of using the two algorithms are shown in Figure 12.1. As can be seen, the ULS algorithm performs much better than the IPS algorithm when register pressure exceeds register availability. When sufficient registers are available the two algorithms perform similarly. Analysis of the code generated by the two algorithms showed that the difference in performance was attributable to greater amounts of spill code introduced by IPS. Thus, it is concluded that the fewer loads of a value performed by live range splitting is of significant benefit.

12.3.2 Hybrid Register Sensitive Schedulers

Although ULS performs better than IPS, it is still limited by its lack of ability to lookahead and consider the impact of its scheduling decisions on resource demands later in the schedule. In an attempt to circumvent this limitation a set of hybrid ULS algorithms were implemented. All of these algorithms incorporate the resource requirements information computed by URSA into their priority functions.

As in the original implementation of ULS, the register pressure value determines which priority function is used. The instruction priority function considers the values in the following order: future pressure estimate, instruction height, P_δ. The register sequentializing priority function considers the values in the following order: P_δ, future pressure estimate, instruction height. Lower values for the future pressure estimate have higher priority. The functions are describe below.

- ULS - is the original priority function described in the previous subsection.
- La1ULS - computes the number of allocation chains covered by the descendants of all scheduled instructions and the instruction under consideration.



(a) 2 issue architecture (b) 4 issue architecture (c) 6 issue architecture

Figure 12.2: Comparison of hybrid RSS techniques

- La2ULS - is similar to La1ULS, except that other instructions in the cycle currently being scheduled are not considered.
- La3ULS - computes only the number of new allocation chains that would be added by scheduling the instruction under consideration.
- La4ULS - computes the same priority as in La3ULS, but does not schedule the instruction if it has slack time and there is at least one other instruction already scheduled in the current cycle if the register pressure would cross a threshold.
- La5ULS - computes only the number of new allocation chains that would be added by scheduling the direct descendants of the instruction under consideration.

The results of these experiments, using the ULS as the base technique in the speedup calculation, are shown in Figure 12.2,. It was remarkable that there was little variation in performance between the priority functions. Examination of the code generated showed that while different instructions were selected for scheduling, the differences were inconsequential. In many cases most of the ready instructions had enough common descendants that their look ahead values were the same. In the cases where there was a difference there was little impact on the amount of splitting required.

The strong similarity of the quality of code generated is the result of the inherently limited ability of list scheduling to look ahead. The groups of instructions within a region tend to be too interrelated for any priority function based on the number of chains or future excessive demands to differentiate between different ready instructions. Priority function La4ULS was specifically designed to negate the over aggressiveness of list scheduling to trying keep all functional units busy in each

cycle. Examination of the heuristic's logs shows that few occasions arose to delay instructions that might increase register pressure and that any such benefits realized were typically offset by delaying some other instructions that shouldn't have been delayed. It is believed that this result is due at least in part to the difference between the rescheduled height of the instructions and the location of the instruction in the final schedule. The difference between these two locations is related to the dilation of the critical path length that occurs during resource allocation.

12.4 Schedule Sensitive Register Allocation (SSRA)

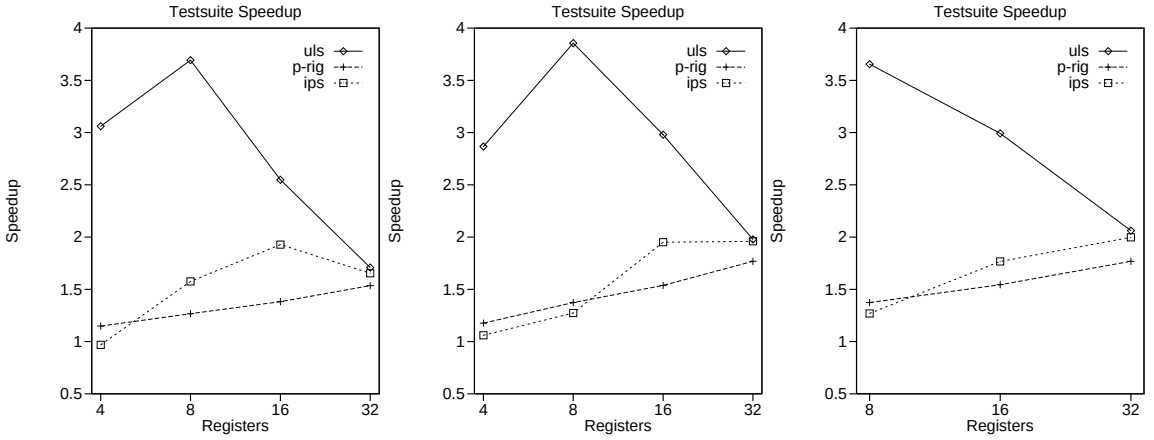
Then next set of experiments performed investigated the performance of scheduler sensitive register allocation (SSRA) techniques. This section describes implementation details and experimental results of both the base and hybrid algorithms.

12.4.1 Base Techniques

This work implemented parallel register interference graph (P-RIG), based on the work of Pinter [Pin93] and Norris and Pollock [NP93]. The P-RIG is constructed in manner similar to the standard register interference graph. Because it contains all interference edges between nodes repressing live ranges that may overlap in the dependence DAG for the region, it is a superset of the RIG computed for a fully sequential program. After the graph has been constructed, it is simplified by removing all nodes that are incident on less than K edges, where K is the number of registers available, since these nodes can always be assigned registers. The remaining nodes represent the excessive demands as computed by this method.

The reduction of the excessive demands is achieved through sequentialization of instructions and spilling of live ranges. Live range spilling rather than live range splitting was implemented in this work since splitting is more complex when instructions are only partially ordered rather than fully ordered, due to the uncertainty of whether two live ranges will overlap in the final schedule. Spilling is performed by storing the value after it is computed and loading the value prior to each use. Sequentialization of a live value D_1 is performed by finding a second live value, D_2, which interferes with D_1 and introducing temporal dependences from all uses of D_2 to D_1. Only such sequentializations that do not introduce dependence cycles are considered.

The priority function used to select a value for spilling or sequentializing compares the costs for the respective reduction methods for each excessive live range. The cost for spilling a value is



(a) 2 issue architecture (b) 4 issue architecture (c) 6 issue architecture

Figure 12.3: Comparison of SSRA and RSS techniques

given by

$$cost_{spill} = \frac{storeCost \times def.execCnt + \sum_{u \in uses} loadCost \times u.execCnt}{numInterferences} \tag{12.3}$$

where $def.execCnt$ is the execution count of the region containing the definition of the value and $u.execCnt$ is the execution count of the region containing a use u of the value. The cost for sequentialization is given by

$$cost_{seq} = \frac{\max_{u \in D_2 uses} (u.EST + D_1.LST) \ominus cpl}{numInterferences} \tag{12.4}$$

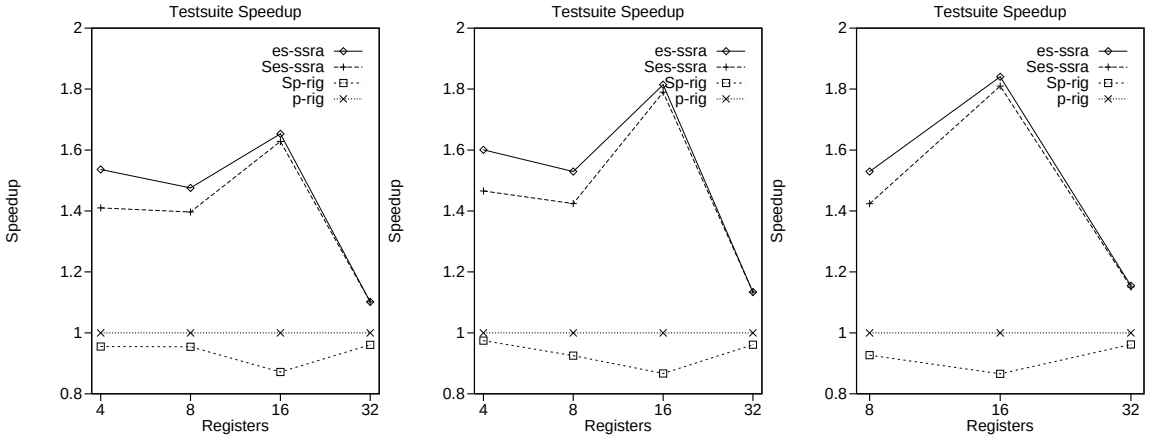
where cpl is the length of the critical path of the region containing D_1, and the symbol $\ominus$ represents the *floored subtraction* function, defined as

$$a \ominus b = \begin{cases} a - b & a > b \\ 0 & \text{otherwise} \end{cases} \tag{12.5}$$

A comparison of schedule sensitive register allocation using the P-RIG to IPS and ULS is shown in Figure 12.3. In most cases P-RIG performed worse than IPS. As mentioned earlier, the P-RIG contains more interferences than the typical RIG computed. These interferences represent a worst case scheduling of the dependence DAG. On the other hand, the RSS algorithms know the precise register pressure at any point in the schedule and only perform a locally minimum number of reductions. The result is better performance.

12.4.2 Hybrid SSRA techniques

The SSRA technique can be modified in a number of ways. The computation of the cost of spilling a value as computed in Equation 12.3 does not consider the fact that definition and uses of the values



(a) 2 issue architecture (b) 4 issue architecture (c) 6 issue architecture

Figure 12.4: Comparison of hybrid SSRA techniques

may have slack time. If one of these instructions has some slack time then there is a hole into which the corresponding memory access can be inserted, reducing its cost. This observation suggests the following modified formulation of the spill cost, referred to as the slack spill cost:

$$Scost_{spill} = \frac{(storeCost \ominus def.slack) \times def.execCnt + \sum_{u \in uses} loadCost \times (u.execCnt \ominus u.slack)}{numInterferences} \tag{12.6}$$

The second observation is that the simplified P-RIG and register excessive sets compute the same type of information using different methods. The P-RIG approach computes excessive demands by identifying interferences directly from the program's dependences, while URSA first computes $CanReuse_{reg}$ from the dependences and then computes excessive demands from the resulting allocation chains. The relative precision of the two computation methods can be compared by substituting the register excessive set for the simplified P-RIG set in the SSRA algorithm.

In all, four SSRA techniques, resulting from all combinations of spill cost functions and excessive demands computations, were implemented and compared.

- p-rig - uses the P-RIG and the spill cost function in Equation 12.3.
- Sp-rig - uses the P-RIG and slack spill cost function in Equation 12.6.
- es-ssra - uses Ursa's excessive sets and the spill cost function in Equation 12.3.
- Ses-ssra - uses Ursa's excessive sets and the slack spill cost function in Equation 12.6.

The results of these compilations are shown in Figure 12.4. There are two remarkable results from these compilations. The first is that the modified priority function degraded performance rather

than improving it. Examination of several cases revealed that more spill code was generated because either some values where spilled prior to attempts to sequentialize live ranges, or values that had less of an impact on reducing the size of the excessive requirements sets were spilled first. The consideration of slack time tended to negate the effects of the priority function's divisor to account for the number of other live values interfered with.

The second result was that the use of URSA's excessive sets significantly reduce the amount of spill code generated. Examination of the generated code revealed a common occurrence mentioned in Brigg's dissertation [Bri92]. Although all nodes in the reduced interference graph interfere with at least K other values, those K other values may not need all K colors. The simplest example is an interference graph of four nodes connected in the shape of a diamond, with $K = 2$. URSA's chain computations naturally realize when such a situation occurs and count fewer interferences. The result of fewer interferences is either a smaller excessive interference set is generated than by interference graph reduction, or no excessive interference set at all is generated while interference graph reduction does generate one. The better performance of the URSA based coloring hybrids is directly due to this effect.

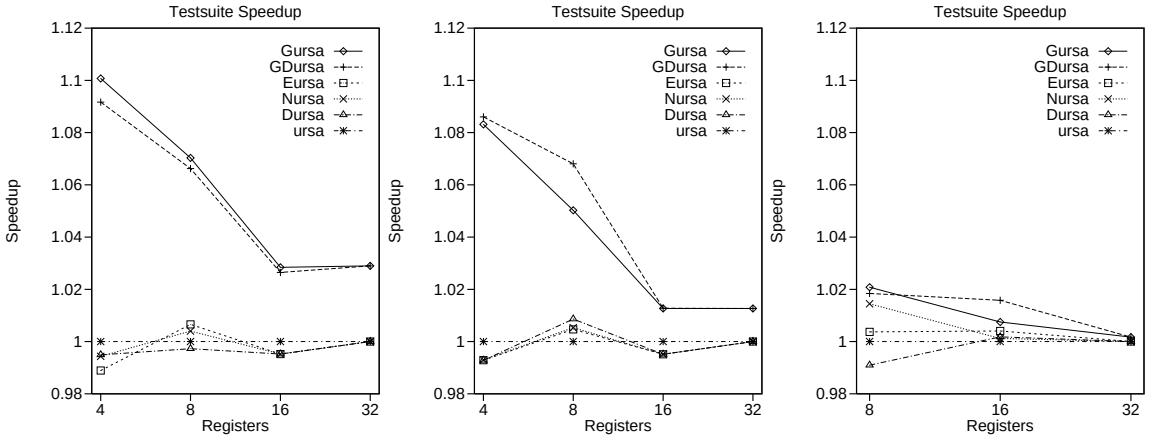
Examination of cases where the interference graph reduction reported an excessive set while URSA's chain computation revealed a common situation. There was typically a group of instructions which legitimately required $K - 1$, registers and they interfered with a chain of several instructions which could all share a single register. Figure 1.1(a) in chapter 1 is an abstract example of such situation. The `D` subgraph contains six instructions which require three registers while the `A` subgraph contains eight instructions which only require one register. In practice, the `A` subgraph may be a series of calculations involving constants and indirect array indexing.

12.5 Unified Resource Allocation

A prototype of URSA was implemented using the resource spackling transformations described in Chapter 6 and cost functions based on the techniques discussed in Chapter 7. The inter-region motions suggested in Chapter 9 were not implemented.

In addition to comparing URSA to the other techniques mentioned above, experimentation was performed to tune the priority function used. Different weights and orders for calculating the component priorities were considered in order to identify key properties of a good heuristic. The priority functions considered are list below.

- Eursa - favor instructions that interfere with larger numbers of excessive instructions.



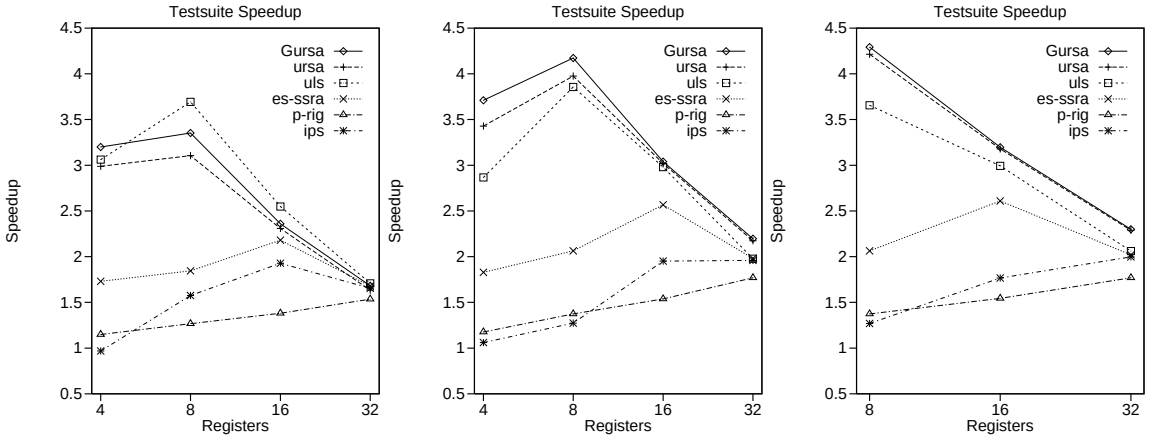
(a) 2 issue architecture (b) 4 issue architecture (c) 6 issue architecture

Figure 12.5: Comparison of variant URSA techniques

- Cursa - ignore all priority components except the cost of delaying instructions or spill code.
- Gursa - be greedy instead of conservative in the order of spackling the instructions selected.
- Dursa - favor reductions that sequentialize more instructions with excessive demands. This function was inspired by Norris and Pollock's sequentialization reductions.
- GDursa - combine the Gursa and Dursa heuristics.
- Nursa - reverse the order the lexigraphic comparison of the priority components. This heuristic was inspired by the success of the Gursa heuristic.

The results of these experiments are shown in Figure 12.5. The most significant improvement resulted from the use of the greedy heuristic GUrsa. This result was unexpected due to the fact that Hsu recommends scheduling instructions with the least amount of slack first [Hsu87]. This experiment suggests that scheduling the instructions with the most slack first achieves better performance because these instructions are the ones most likely to be moved beyond the range of the excessive set. Thus fewer reduction transformations are typically required. Most other variations had little effect on the performance of Ursa. However, the reversed ordering of the priority components consistently did slightly worse. This result indicates that the base ordering of the priority components is correct.

The best heuristics from the each set of experiments are compared against the base heuristics in Figure 12.6. As seen from the graph both URSA and ULS outperformed all prior techniques for all architectures. Between URSA and ULS there was no clear winner. ULS performed better on the two issue architectures while URSA performed better on all of the wider architectures. Examination of the



(a) 2 issue architecture (b) 4 issue architecture (c) 6 issue architecture

Figure 12.6: Comparison of base and best techniques

	ULS		URSA		hand	
Benchmark	CPL	Insts.	CPL	Insts.	CPL	Insts.
loop2	63	70	95	79	22	40
loop10	117	131	181	150	42	71

Table 12.2: Critical path lengths and number of instructions for 2-4 architecture

individual benchmarks revealed that no single heuristic consistently generated the best or worse code. Two cases where URSA performed worst than ULS were selected for an in–depth examination. These cases were the single region loop bodies of `loop2` and `loop10`. These regions were scheduled and allocated by hand to compare the heuristics to the optimal and identify weaknesses in the heuristics. The results are shown in Table 12.2.

In both cases, the number of available registers was sufficient and no extra memory accesses were required. Thus, the difference between the number of instructions in the hand coded cases and the heuristics indicates the total number of load and store instructions inserted by the heuristics.

Examination of the allocation decisions made by ULS showed that it has very limited knowledge of the unscheduled instructions. List scheduling in general has no knowledge of the width of the resource requirements of the unscheduled instructions. Thus heuristics based on list scheduling typically use only an instruction's height in the dependence DAG to decide how long it can be delayed. This limitation is evidenced in both cases by the register thrashing resulting from scheduling live ranges based only on the height of the defining instruction.

A similar situation occurs for URSA in `loop2`. Examination of the region showed that there was insufficient slack time in the DAG to insert spill code, let alone attempt to delay live ranges. As a result URSA attempted to schedule all instructions as close as possible to their heights

in the DAG. The results obtained by hand coding were achieved by observing that some groups of instructions would have to be delayed. Thus groups of instructions that were closely related were delayed enmasse. In this manner values were not computed until they were needed in the final schedule and generation of spill code was avoided. A similar situation occurred in URSA's allocation of `loop10`. However, the problem was compounded by the existence of a number of store instructions at the end of small groups of instructions. Although most of the instructions in the small groups have little or no slack time, the stores have relatively large amounts of slack time. URSA preferred to delay these stores to near the end of the region due to the increased availability of functional units to execute them. Delaying the stores caused the lifetimes of the values they spilled to be lengthened and thus cause interference with a large number of live ranges on critical paths. Once again, thrashing of registers occurred. Hand coding again exploited the relations of instructions by delaying and scheduling instructions in small groups.

12.6 Measurement and Compile Time Statistics

Besides quality of code generated, there are two issues concerning the practical use of the URSA technique developed in this work:

1. How accurate are the measurement heuristics for spanning resources in practice?
2. What is the compilation time of URSA based techniques compared to existing techniques?

The appropriate sections of the prototype compiler were instrumented to collect data to answer these two questions.

12.6.1 Measurement Heuristics

The architecture targeted by the host compiler did not contain any instructions that used more than two input values. Thus, any NP-complete components that were encountered could be solved using the specialized matching algorithm referenced in Appendix A. However, this matching algorithm was not implemented in the prototype due to the extra data structures needed. Instead a greedy algorithm was used to compute the minimum cover sets. As a result, the measurement algorithms could still produce imprecise measurements. To determine how often such imprecisions occurred the measurement algorithm was instrumented with code to record all NP-complete candidate components that were given to the greedy heuristic for analysis. Both the components and the solutions found by the greedy heuristic were recorded in a log file. These log files were then analyzed offline.

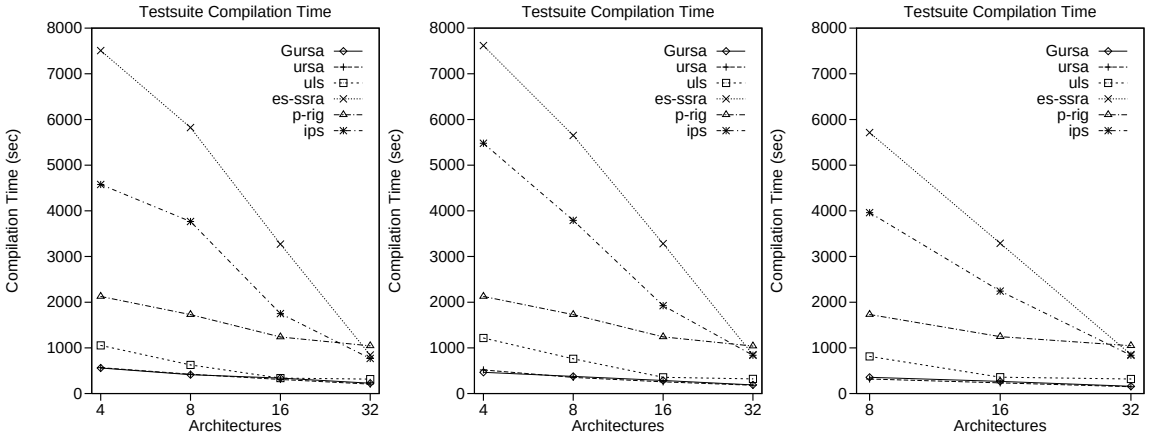
Initially, the offline analysis consisted of performing an exhaustive search for the minimal covering set and comparing its size to the size of the covering set found by the greedy heuristic during compilation. While the exhaustive search was too costly to implement in the compiler, most of the components were small enough that offline analysis was not unreasonable. However one component was found which was intractable for performing an exhaustive search. Upon examination of this component it was noticed that a large number of the parent instructions had single children using them. Such children must be in the covering set as they are the only ones that can kill these single use parents. In the particular component being examined, the minimum covering set consisted of all of the children with single use parents.

As a result, the offline analysis algorithm was redesigned to perform a prepass that added all children of single use parents to the cover set and then to perform an exhaustive search on the remaining children. As demonstrated in the component examined, the prepass may find a minimal covering set, avoiding the need to run the exhaustive search at all. Such cases are called *trivial* and were specially noted by the offline analysis program.

In an effort to gauge the number of different components encountered in benchmark programs the offline analysis program performed a partial isomorphism of the components. This isomorphism consisted of normalizing the node numbers in the range 1 to N, where N is the number of nodes in the component. The partiality of the isomorphism is a result of the fact that the relative ordering of the nodes was still preserved.

From all of the measurements performed during the more than 4,000 compilations performed, a total of 44,432 NP-Complete component candidates were recorded. In all 44,432 cases the greedy heuristic implemented in the compiler found the minimal solution, resulting in perfect precision for the calculation of all resource requirements in the benchmark programs considered. The 44,432 problems reduced to only 69 unique components with respect to the partial isomorphism. Of these 69 components, 67 were trivial in that the prepass algorithm found the minimal covering set. The 2 nontrivial components only occurred a total of 8 times out of the 44,432 components encountered. Performing a complete isomorphism by hand on the 69 semi-unique components reduced them to 42 completely unique components. There was only one completely unique nontrivial component which contained a total of five instructions.

Examination of the two nontrivial components showed that any algorithm that selects children for the minimal cover set if and only if they kill at least one of the remaining live parents will find a minimal cover. This result is due to the symmetrical nature of the nontrivial components encountered.



(a) 2 issue architecture (b) 4 issue architecture (c) 6 issue architecture

Figure 12.7: Comparison of compilation times

This experiment suggests a better method for implementing spanning resource Reuse DAG computation than was done in the prototype compiler. All NP-Complete candidate components should be analyzed by the prepass algorithm mentioned above. Only if a true NP-complete component remains after this analysis should a more expensive analysis algorithm be invoked. Three possible algorithms are suggested for consideration, based on the desired trade off between analysis time and precision of the requirements. This experiment suggests that an exhaustive search is probably not unreasonable if complete precision is desired; the exponential computation time is most likely easily amortized over the total compilation time. As a middle ground, either a matching based algorithm or the greedy heuristic implemented in the prototype compiler can be used. This heuristic can also be used if a threshold for the size of components handed to an exhaustive search is implemented. For pure speed a random ordering of the children nodes for searching for those that kill at least one parent can be used. The algorithmic complexities of these algorithms are $\Omega(2^C)$, $O(C^{2/5})$, $O(C^2)$, and $O(C)$ set operations respectively, where C is the number of child nodes in the component.

The component identification algorithm has a time complexity of $O(N^2)$ set operations, where N is the total number of nodes in the program DAG. Thus, asymptotically, the difference between the random and greedy heuristics is inconsequential in the overall computation. The combination of the rarity of encountering a true NP-Complete problem and its likely size after reduction by the prepass makes the selection of an algorithm mostly a theoretical issue.

12.6.2 Compilation Time

The second issue concerning practical use is the compile times of the respective algorithms. The times for the base and best hybrid techniques are shown in Figure 12.7. The times shown are only for the resource allocation phase of the compiler. There are several interesting items to note about the relative times.

Despite the fact that the resource requirements are recomputed from scratch each time, the URSA based algorithms had the lowest compilation times. Since only the priority function differed between the two URSA techniques, it is not surprising that they had very similar times.

The ULS algorithm was the next best performer and was quite close to the URSA algorithms expect when few registers were available. The reason for ULS performing worse than URSA is unclear. One possible explanation is that ULS operates on a DAG and introduces temporal dependences as each cycle is scheduled, requiring an update of transitive dependence information. The number of such updates is greater than the number times that the resource demands are computed in the URSA algorithms.

The IPS algorithm is slower than the ULS algorithm by nearly a constant factor. This extra time is due to the invocation of the register coloring algorithm after the scheduling phase.

The poorest performing algorithms are the coloring based algorithms. The primary reason for this fact is that the sequentialization reduction is not optimized. As mentioned earlier, this implementation of the technique attempts to make a minimal sequentialization as possible to avoid over-sequentialization. As a result, several sequentialization steps may be taken to achieve the desired effect. Norris and Pollock have developed heuristics to reduce the number of sequentialization steps needed, but have not given the details of their techniques [NP93].

The difference in time between the base P-RIG algorithm and the hybrid ES-SSRA algorithm is the time required to compute URSA's resource demands information. This difference is less when more registers are available as fewer reduction steps are performed by the algorithms.

Finally, it should be noted that although URSA's resource requirements computations are prime candidates for incremental updating, such techniques were not implemented in the prototype. Instead, full recomputation is performed.

12.7 Comments

Several observerations can be made from the results of the various sets of experiments performed in this work. First, the most significant impact on the quality of code generated is due to live range

splitting. The two new algorithms developed in this work, ULS and URSA, both naturally perform live range splitting as a part of their reduction techniques. Currently, the only apparent method of incorporating live range splitting into techniques based on register coloring is to use the spackling transformations. However, after replacing the excessive demands detection and reduction methods of SSRA techniques, the only difference between them and URSA is the priority function used for selecting which value to split.

The second observation is that while the two new algorithms presented in this work make significant improvements over prior techniques, there is still a large amount of room for further improvement. The results of the hand coded examples suggest that further improvements in heuristics must come from considering reductions in a larger scope.

A third observation made is that list scheduling remains constrained by its limited scope of knowledge of the code being allocated, as evidenced by the difficulty in affecting the quality of code generated by varying the priority schemes. On the other hand, while the improvements where not significant in the final comparison, affecting the quality of code generated by URSA is easily accomplished.

The combination of the second and third observations with the unified representation used by URSA is encouraging. There appears to be much potential for improvement by future algorithms based on GURRR and/or URSA.

Finally, the precision of computing register excessive sets, with respect to both the NP-Completeness of the problem and to the current performance of interference graph based techniques is practical. These experiments show that computation of register excessive sets are quite beneficial. As a result, other needs for estimates of register pressure, such as parallelizing transformations and software pipelining, should consider using register excessive sets as a part of their analysis.

Chapter 13

Concluding Remarks

13.1 Summary

Architectures for Instruction Level Parallelism(ILP) present several new challenges to traditional compiler implementations. The goal of this work was to address important code generation problems for ILP involving register allocation and instruction scheduling. In particular, there are two significant problems: 1) ILP highlights negative interactions between register allocation and instruction scheduling, 2) existing techniques for register allocation either are not designed to handle ILP or do not fully exploit ILP when performing spilling.

This work addresses both problems simultaneously by designing new techniques for both register allocation and instruction scheduling. The design of the techniques is motivated by two observations. First, register allocation and instruction scheduling both carry out the allocation of necessary resources and then assign specific instances of the resources to the instructions that need them. A part of the allocation process is determining *when* to allocate a resource to an instruction. Second, allocation is only a problem in sections of the program where the demand for resource exceeds the number of resources available. The introduction of temporal dependences by a compiler affects allocation by imposing constraints on scheduling decisions. These observations form the basis for proposing the *Measure and Reduce* paradigm.

The enabling technology developed in this work is the measurement of worst case resource demands for both types of resource and their incorporation into a single representation usable by all allocation and assignment tasks. URSA's measurement information was designed to be easily incorporated into existing intermediate representation. The measurement information is of use to a wide variety of compiler tasks concerned with architectural features including register allocation, instruction scheduler, resource constrained global code motion, and transformations and optimizations. The result is a framework flexible enough to address most of the recent features of ILP architectures.

The main contributions of this work are as follows:

- Development of techniques for measuring the maximum number of resources needed to exploit all inherent ILP in a segment of a program. The techniques are general enough to incorporate both functional unit and register types of resources. The availability of such information enables the resource allocation phases to concentrate precisely on the problem areas.

- Development of a DAG based intermediate representation which incorporates resource requirements information. Basing the representation of the requirements information on a structure as flexible as a DAG demonstrates that it can be used by commonly used intermediate representations. Using existing representations as a base provides a convenient medium to communicate the needed information to the resource allocation phases.

- Development of a resource allocation framework that specifies how allocation of resources to instructions can be performed using the resource requirements information computed. The framework provides a method to directly assess and compare the impact of all allocation options under consideration.

- The development of a powerful framework for describing and comparing approaches to resource allocation, whether they are integrated or not. This framework consists of the Measure and Reduce Paradigm and the theory of resource spackling, which identifies the necessary conditions for achieving reductions in resource requirements.

- Development of high level heuristics that drive a unified allocation of all resources. These heuristics demonstrate how the allocation framework can be applied to resource allocation and assignment for several common tasks, including global register allocation, local scheduling and resource constrained global code motion designed for multiple issue architectures. These heuristics incorporate live range splitting in an ILP environment, an improvement over previous techniques that perform either no or limited live range splitting.

- Experimental results that demonstrate the benefits of the techniques developed. Important results from the experiments conclude the following. Live range splitting is critical to achieving good performance. Even when compared to other fully capable live range splitting techniques, the resource allocation based techniques generally perform better due to the unification of resource allocation. Although in theory maximum register requirements is an NP-complete problem, in practice they can be computed efficiently with a high degree of precise (completely

precise in the experiments). Finally, the experiments show that the resource requirements information improves traditional coloring based approaches to register allocation.

13.2 Future Work

There are a number of problems open for future research.

- The heuristics for selecting both instructions and reduction transformations in the URSA experiments were simple in concept and served their purpose as a proof of concept. There are no doubt better, although possibly more complex, heuristics for these problems.

- The unification of register allocation and instruction scheduling is only a first step in reorganizing the back end of the compiler. Higher degrees of integration are possible by combining transformations and optimizations with resource management. A central concept of this work is to perform transformations *on demand*, that is, to expose additional instruction level parallelism only in places where the level of inherent ILP is insufficient. Furthermore when additional ILP is desired, only as much as is needed to fill the idle resources should be exposed. In this manner extra work is not generated for register allocation and instruction scheduling.

  Several areas must be addressed to support this integration, including prediction of the changes that would be affected by the transformations, and methods to use URSA's information to "throttle" the transformations. In addition, the use of URSA to guide optimizations performed early in the back end, such as loop unrolling, will require URSA to operate on a high level intermediate representation. URSA as a framework is sufficiently flexible to accommodate such uses. However, for the measurements to be beneficial, the approximations must reasonably account for the eventual results of lowering the representation.

- Fast accurate estimations of resource requirements may have other uses, such as guiding global code motion (GCM) and heuristic selection. Resource profiles may be used to determine which blocks should be used as targets for GCM and which ones should be evaporated to eliminate branches. It has been observed before that particular transformations and resource heuristics perform better on some programs than others [Whi91]. Indeed, their effectiveness may vary with in a single program. The availability of resource requirements profiles provides new information that may be guide the selection of which heuristic that is most likely to obtain the best result for the section of the program in question.

- The simple use of excessive sets as clique candidates in interference graphs in register allocation shows the benefit of incorporating URSA's information in traditional coloring based register allocation techniques. Further work needs to be done in communicating the allocation chain information to the register assignment phase. Further, there may be ways to use URSA's information in the selection of a live value for spilling and the placement of spill code. In addition to sequentializing or spilling the selected value, the information in GURRR may be able to be used to find ways to split the live range even though the instructions have not been fully ordered yet.

  Despite the lack of benefit in the register sensitive scheduler experiments, there may still be ways to incorporate resource requirements information into instruction scheduling as well. Further examination of sample cases may suggest other methods for both determining the proper look ahead scope and how to incorporate the information into the selection priority function. An alternative approach to the look ahead problem would be to preprocess the DAG. That is, to compute the resource requirements information for the DAG and annotate the instructions with scheduling hints prior to actual scheduling. These hints could then be incorporated into the priority function.

Appendices

Appendix A

NP-Completeness and a Heuristic for Computing $Kill()$

This appendix addresses the problem of computing the function $Kill()$. $Kill(a)$ is a function that returns the node that kills the value computed by node a, and is used to build the $Reuse_{Reg}$ DAG. Section A.1 shows that computing $Kill()$ to maximize register requirements is an NP-Complete problem. Section A.2 presents a practical heuristic for computing $Kill()$.

A.1 NP-Completeness

The $Reuse_{Reg}$ DAG is used to determine the maximum number of registers that the program can use under any schedule. Therefore, for a node a, $Kill(a)$ should be computed so that a is alive with as many other values at the same time as possible. Given a collection of nodes, the maximum number of instructions requiring registers is the maximum number of independent values plus the maximum number of their children that can be executed without killing any of them. Thus, $Kill(a)$ must be in a minimum set of children that kills all of the of values from the maximum live set containing a. As an example, consider the DAG in Figure A.1. If node `E` is the last to execute then the values from all other nodes are alive at once, requiring six registers. If node `E` is the first child to execute then each other child kills a parent and can reuse its register. In this case only four registers are required.

Formally, the problem of computing $Kill()$ is treated as a decision problem of finding K or fewer children that kill a set of parents' values:

Minimum Killing Set

INSTANCE: A DAG (N, E), positive $K <= |N|$.

QUESTION: Does there exist a minimum killing set, *i.e.*, $N' \subset N$ such that $|N'| <= K$, and $\underset{n \in N}{\forall} \exists c \ni (n, c) \in E$ and $c \in N'$?

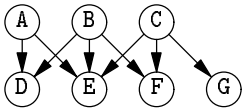


Figure A.1: A complex case for defining *Kill()*

The complex subcase of computing $Kill()$ can be described as a subDAG that is bipartite and has at least one parent node with an out-degree of at least two and at least one child node with an in-degree of at least two. Such cases can be shown to be NP-Complete by reduction to the Minimum Cover problem. The statement of the Minimum Cover problem, taken from Garey and Johnson [GJ79], is as follows:

> Minimum Cover
>
> INSTANCE: Collection C of subsets of a finite set S, positive $J <= |C|$.
>
> QUESTION: Does C contain a cover for S of size J or less, *i.e.,* a subset $C' \subset C$ with $|C'| <= J$ such that every element of S belongs to at least one member of C'?
>
> REFERENCE: [Kar72] Transformation from Exact Cover by 3-Sets.

THEOREM A.1 *Computing $Kill()$ for all nodes in the DAG to maximize the register requirements, i.e., Minimum Killing Set, is NP-Complete.*

PROOF: by reduction to the Minimum Cover problem.

Minimum Killing Set is in NP since a nondeterministic algorithm can guess a solution and check in polynomial time that there are K or less nodes in N' and that all parent nodes are killed.

Let (N, E) be a DAG with $N = S \cup C$ and $E = \{(s,c) | s \in S, c \in C, \text{ and } s \in c\}$. Then (N, E) is a bipartite graph, constructed in $O(|S| + |C| + \prod_{c \in C} |c|)$ time. Let $K = J$. If there is a solution to the Minimum Cover problem, then $N' = C'$ is a solution for the Minimum Killing Set problem. Conversely, if there is a solution to the Minimum Killing Set problem, then $C' = N'$ is a solution to the Minimum Cover problem. Thus, Minimum Killing Set is NP-Complete since it is in NP, Minimum Cover can be transformed to it in polynomial time, and there is a solution to Minimum Killing Set if and only if there is a solution to Minimum Cover. ■

In practice, a simple greedy method based on selecting the child that kills the most parents can be used and is quite effective [CLR90, pp. 974–978]. Furthermore, if no child has in-degree of greater than 2, a specialized matching algorithm can be used []. In this matching algorithm the parents and the child form the two partitions of the bipartite graph. Any child node incident on a matching edge is in the minimum cover set. It can be shown that if there a parent node that is not

incident on a matching edge, it still will be covered by a child that is incident on a matching edge. If this were not the case, then the matching would not be maximum because the edge connecting the parent to the child could be added to the set of matching edges. Thus, all parents will be covered.

A.2 Computing $Kill()$

For many NP-Complete problems only certain cases or portions of the problem cause the NP-Completeness. This section identifies the characteristics of the portions of a program where computing $Kill()$ is NP-Complete. An algorithm is presented that both identifies the portions of the program where computing $Kill()$ is NP-Complete, and computes $Kill()$ precisely for all of the portions of a program that are not NP-Complete problems.

Partitioning of the portions of the program into NP-Complete and polynomial parts for computing $Kill()$ is based on the out-degree of the nodes to be killed and the in-degree of the potential killing nodes. A *Multiple Out* (MO) node is a node with an out-degree greater than one. A *Single Out* (SO) node is a node with an out-degree of one. *Multiple In* (MI) and *Single In* (SI) nodes are similarly defined based on the in-degree of a node. A *set* of nodes is called Multiple Out if all nodes in the set are Multiple Out. A set of nodes is called Single Out if all nodes in the set are Single Out. A set of nodes is called *Both Out* if it contains both Multiple Out and Single Out nodes. Multiple In, Single In, and Both In are similarly defined based on the in-degree of the nodes in the set.

The heuristic first partitions the DAG into bipartite subDAGs. The killing nodes are then found for each bipartite subDAG. The covering is found by performing a variation of the algorithm for finding connected components. In this variation a node is connected to other nodes in the component either by only its incoming edges or only its outgoing edges. Thus all nodes except the root and leaf of the DAG will exist in two components, one where the node is a parent and one where the node is a child.

Certain edges in the DAG must be ignored for the bipartite components algorithm to work. Consider the example in Figure A.2. Node `B` is a parent to `C`, which is one of `B`'s parent's other children. The edge (`A`, `B`) must be removed so that `B` does not appear as a child in more than one bipartite subDAG. This is allowable since `B` can never kill `A`'s value.

Computing $Kill()$ can be broken into cases, based on the combinations of the types of the parent and child sets. Figure A.3 shows all nine combinations of types of the parent and child sets. The first letter of the combination name indicates the type of the parent set, the second letter

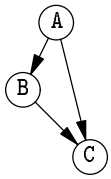
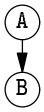
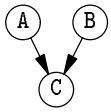


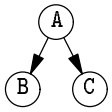
Figure A.2: A special case for partitioning into bipartite subDAGs



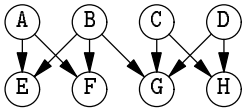
(a) SS - Single Out, Single In



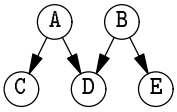
(b) SM - Single Out, Multiple In



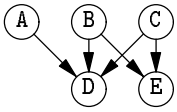
(c) MS - Multiple Out, Single In



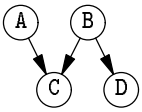
(d) MM - Multiple Out, Multiple In



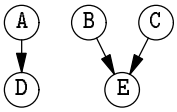
(e) MB - Multiple Out, Both In



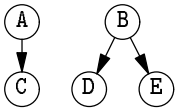
(f) BM - Both Out, Multiple In



(g) BB - Both Out, Both In



(h) SB - Single Out, Both In



(i) BS - Both Out, Single In

Figure A.3: All combinations of Out and In nodes

indicates the type of the child set. Note that cases SB (Figure A.3(h)) and BS (Figure A.3(i)) are each actually two disjoint subDAGS, represented by previous cases.

There are three rules used to compute $Kill()$, depending on the combination of node types in the bipartite subDAG.

1. SS, SM

   If all parents are Single Out, they each have only one child that can kill them. This child is defined as the $Kill(p)$ for each parent p.

2. MS

   In this case there can only be one parent. The parent's value will be alive with all but one of its children. The child that kills the parent p, $Kill(p)$, can be randomly selected.

3. MM, MB, BM, BB

   These are the complex cases that may be NP-Complete. The order of execution of the children effects the number of registers required. The greedy heuristic mentioned in Section A.1 is used to compute the minimum killing set. For each parent p, defining $Kill(p)$ by select any child of p from the killing set.

The computation of $Kill()$ is summarized in Figure A.4.

```
function computeKill( DAG (N, E) ) returns function Kill()
{
      /* mark edges to ignore during bipartite coverage */
   foreach n ∈ N
      foreach p ∈ Parents(n)
         if ( Children(n) ⋂ Children(p) <> φ )
            mark edge (p,n) as ignore;

      /* find the bipartite components */
   components = findBipartiteConnectedComponents( (N, E) );

      /* compute Kill() */
   foreach component B ∈ components
         /* rule 1 */
      if ( ∀ p.outDegree = 1 )
           parents p inset c
         foreach parent p ∈ B
            define Kill(p) = Child(p);
      else
         /* rule 2 */
      if ( ∀ c.inDegree = 1 )
           children c inset B
         foreach parent p ∈ B
         {  select c from Children(p);
            define Kill(p) = c;
         }
      else
         /* rule 3 */
         while ( ∃ p ∈ B ∋ p.alive = True )
         {  select c from the children that kill the most parents;
            foreach p ∈ Parents(c)
               if ( p.alive )
               {  p.alive = False;
                  define Kill(p) = c;
               }
         }
}
```

Figure A.4: Function *computeKill()*

Appendix B

Speedup Tables

benchmark	ips	uls	La1uls	La2uls	La4uls	La3uls	La5uls
bubble	1.28	3.92	3.92	3.92	3.10	3.10	3.10
hanoi	0.00	2.70	2.70	2.70	2.45	2.70	2.70
heapsort	1.01	1.71	1.95	1.95	0.00	1.94	1.94
intmm	1.20	3.53	3.51	3.53	3.07	3.53	3.53
nsieve	0.91	2.62	2.62	2.62	2.62	2.62	2.62
perm	1.17	4.38	4.38	4.38	3.91	4.38	4.38
puzzle	0.00	0.98	0.97	0.98	0.98	0.98	0.98
queens	0.62	1.37	1.18	1.47	1.33	1.37	1.29
quick	1.02	2.35	2.17	2.35	2.37	2.35	2.35
loop1	0.78	2.74	2.74	2.74	3.63	2.74	2.74
loop2	0.86	2.95	2.95	2.95	2.72	2.92	2.99
loop3	0.76	4.11	4.11	4.11	3.36	4.11	4.11
loop4	0.85	3.30	3.29	3.30	2.41	3.43	3.43
loop5	0.95	3.34	3.34	3.34	2.60	3.34	0.00
loop6	0.98	2.47	2.47	2.47	2.33	2.47	0.00
loop7	0.79	2.70	2.70	2.70	2.22	2.77	2.28
loop9	0.95	3.15	3.15	3.15	0.00	0.00	2.77
loop10	1.33	4.79	4.79	4.79	4.72	4.72	4.38
loop11	0.83	3.06	3.06	3.06	3.50	3.06	3.06
loop12	0.85	4.70	4.70	4.70	3.91	4.70	4.70
loop13	1.10	3.32	3.22	3.22	0.00	0.00	3.02
loop14	1.14	3.16	3.16	3.16	3.66	3.66	0.00

Table B.1: Individual speedups for the rss heuristics on architecture 2-4

benchmark	ips	uls	La1uls	La2uls	La4uls	La3uls	La5uls
bubble	1.45	1.79	1.79	1.79	1.79	1.79	1.79
hanoi	0.00	1.40	1.40	1.40	1.27	1.40	1.40
heapsort	1.16	1.92	2.04	2.04	2.13	2.06	2.06
intmm	1.41	8.54	8.54	8.54	8.54	8.54	8.54
nsieve	1.20	1.60	1.60	1.60	1.60	1.60	1.60
perm	1.92	3.00	3.00	3.00	2.77	3.00	3.00
puzzle	0.98	1.53	1.53	1.53	1.53	1.53	1.53
queens	1.50	2.71	2.12	2.91	1.81	2.12	2.01
quick	1.97	5.08	5.08	5.08	5.08	5.08	5.08
loop1	1.19	3.73	3.73	3.73	3.05	3.73	3.73
loop2	1.17	5.87	5.87	5.87	4.08	5.87	5.87
loop3	1.40	1.56	1.56	1.56	1.27	1.56	1.56
loop4	1.17	5.42	5.42	5.42	4.06	5.42	5.42
loop5	6.45	6.45	6.45	6.45	4.27	6.45	3.62
loop6	1.22	6.34	6.34	6.34	4.23	6.34	3.10
loop7	1.16	4.28	4.28	4.28	3.43	4.28	4.28
loop9	1.06	2.55	2.56	2.56	2.78	2.78	2.15
loop10	1.21	3.71	3.71	3.71	3.71	3.71	2.48
loop11	1.55	1.55	1.55	1.55	1.21	1.55	1.55
loop12	1.60	1.60	1.60	1.60	1.33	1.60	1.60
loop13	1.06	2.75	2.75	2.75	2.75	2.75	2.68
loop14	1.24	7.88	7.88	7.88	7.88	7.88	7.88

Table B.2: Individual speedups for the rss heuristics on architecture 2-8

benchmark	ips	uls	La1uls	La2uls	La4uls	La3uls	La5uls
bubble	1.78	1.78	1.78	1.78	1.78	1.78	1.78
hanoi	1.40	1.40	1.40	1.40	1.27	1.40	1.40
heapsort	1.38	2.20	2.19	2.19	2.45	2.19	2.19
intmm	1.69	1.69	1.69	1.69	1.69	1.69	1.69
nsieve	1.58	3.12	3.12	3.12	3.12	3.12	3.12
perm	1.78	1.78	1.78	1.78	1.65	1.78	1.78
puzzle	0.00	1.14	1.14	1.14	1.14	1.14	1.14
queens	1.70	1.70	1.33	1.82	1.14	1.33	1.26
quick	1.30	1.68	1.68	1.68	1.68	1.68	1.68
loop1	1.64	1.64	1.64	1.64	1.35	1.64	1.64
loop2	0.85	4.14	4.14	4.14	2.88	4.14	4.14
loop3	1.56	1.56	1.56	1.56	1.27	1.56	1.56
loop4	1.67	1.67	1.67	1.67	1.25	1.67	1.67
loop5	1.78	1.78	1.78	1.78	1.18	1.78	1.61
loop6	5.99	5.99	5.99	5.99	4.00	5.99	5.58
loop7	1.65	1.65	1.65	1.65	1.32	1.65	1.65
loop9	1.31	4.25	4.25	4.25	4.25	4.25	3.00
loop10	1.46	4.54	4.54	4.54	4.54	4.54	4.54
loop11	1.55	1.55	1.55	1.55	1.21	1.55	1.55
loop12	1.60	1.60	1.60	1.60	1.33	1.60	1.60
loop13	1.33	3.70	3.70	3.70	3.70	3.70	3.70
loop14	5.50	5.50	5.50	5.50	5.50	5.50	5.19

Table B.3: Individual speedups for the rss heuristics on architecture 2-16

benchmark	ips	uls	La1uls	La2uls	La4uls	La3uls	La5uls
bubble	1.56	1.56	1.56	1.56	1.56	1.56	1.56
hanoi	1.40	1.40	1.40	1.40	1.27	1.40	1.40
heapsort	1.11	1.93	1.93	1.93	1.92	1.93	1.93
intmm	1.69	1.69	1.69	1.69	1.69	1.69	1.69
nsieve	1.39	1.65	1.65	1.65	1.65	1.65	1.65
perm	1.78	1.78	1.78	1.78	1.65	1.78	1.78
puzzle	0.00	1.13	1.13	1.13	1.13	1.13	1.13
queens	1.39	1.39	1.09	1.49	0.93	1.09	1.03
quick	1.33	1.46	1.46	1.46	1.46	1.46	1.46
loop1	1.64	1.64	1.64	1.64	1.35	1.64	1.64
loop2	1.71	1.71	1.71	1.71	1.19	1.71	1.71
loop3	1.56	1.56	1.56	1.56	1.27	1.56	1.56
loop4	1.67	1.67	1.67	1.67	1.25	1.67	1.67
loop5	1.78	1.78	1.78	1.78	1.18	1.78	1.61
loop6	1.67	1.67	1.67	1.67	1.12	1.67	1.56
loop7	1.65	1.65	1.65	1.65	1.32	1.65	1.65
loop9	1.46	1.46	1.46	1.46	1.46	1.46	1.46
loop10	1.72	1.72	1.72	1.72	1.72	1.72	1.72
loop11	1.55	1.55	1.55	1.55	1.21	1.55	1.55
loop12	1.60	1.60	1.60	1.60	1.33	1.60	1.60
loop13	3.28	3.78	3.78	3.78	3.78	3.78	3.78
loop14	1.80	1.80	1.80	1.80	1.80	1.80	1.70

Table B.4: Individual speedups for the rss heuristics on architecture 2-32

benchmark	ips	uls	La1uls	La2uls	La4uls	La3uls	La5uls
bubble	1.21	2.75	2.75	2.75	2.66	2.65	2.65
hanoi	0.00	3.18	3.18	3.18	2.84	3.18	3.18
heapsort	0.99	1.94	2.08	2.08	0.00	1.88	1.88
intmm	1.18	3.36	3.35	3.36	3.73	3.07	3.07
nsieve	1.98	2.62	2.62	2.62	2.62	2.62	2.62
perm	1.22	4.57	4.57	4.57	3.91	4.57	4.57
puzzle	1.11	0.98	0.98	0.98	0.98	0.98	0.98
queens	0.65	1.44	1.11	1.44	1.18	1.24	1.08
quick	1.12	1.76	1.76	1.76	1.57	1.56	1.56
loop1	0.80	2.41	2.41	2.41	4.44	2.60	2.82
loop2	0.90	2.72	2.72	2.72	2.95	3.14	2.85
loop3	0.97	4.62	4.62	4.62	3.70	4.62	4.62
loop4	0.83	3.71	3.70	3.71	2.54	3.71	3.71
loop5	1.01	2.29	2.29	2.29	2.91	3.05	0.00
loop6	0.98	2.49	2.49	2.49	2.27	2.49	0.00
loop7	0.66	2.57	2.57	2.57	2.59	2.19	0.00
loop9	0.95	2.66	2.88	2.88	3.37	3.37	2.86
loop10	1.35	4.20	0.00	0.00	0.00	4.24	4.60
loop11	0.94	2.23	2.23	2.23	3.77	2.04	2.23
loop12	1.24	5.22	5.22	5.22	4.27	5.22	5.22
loop13	1.06	3.07	0.00	0.00	0.00	0.00	3.10
loop14	1.09	2.27	2.11	2.11	0.00	0.00	0.00

Table B.5: Individual speedups for the rss heuristics on architecture 4-4

benchmark	ips	uls	La1uls	La2uls	La4uls	La3uls	La5uls
bubble	0.43	2.02	2.02	2.02	2.02	2.02	2.02
hanoi	0.00	1.65	1.65	1.65	1.47	1.65	1.65
heapsort	1.36	2.16	2.38	2.38	2.30	2.38	2.38
intmm	1.75	8.57	8.57	8.57	8.57	8.57	8.57
nsieve	1.66	1.60	1.60	1.60	1.60	1.60	1.60
perm	1.97	3.12	3.12	3.12	2.67	3.12	3.12
puzzle	1.08	1.54	1.54	1.54	1.54	1.54	1.54
queens	1.66	2.78	2.17	2.99	1.85	2.43	2.29
quick	1.31	5.87	5.87	5.87	5.87	5.87	5.87
loop1	1.41	5.09	5.09	5.09	3.73	5.09	5.09
loop2	1.17	8.30	8.30	8.30	4.46	8.30	8.30
loop3	0.70	1.75	1.75	1.75	1.40	1.75	1.75
loop4	1.33	5.79	5.79	5.79	4.50	5.79	5.79
loop5	1.33	4.19	4.19	4.19	4.78	4.19	4.12
loop6	1.22	6.86	6.86	6.86	4.45	6.86	3.21
loop7	1.28	6.41	6.41	6.41	4.15	6.41	6.41
loop9	1.02	2.19	2.15	2.15	2.49	2.49	2.74
loop10	1.22	4.25	4.25	4.25	4.02	4.02	3.79
loop11	0.74	1.70	1.70	1.70	1.31	1.70	1.70
loop12	1.78	1.78	1.78	1.78	1.45	1.78	1.78
loop13	1.04	3.15	3.03	3.03	3.32	3.32	3.01
loop14	1.29	4.07	4.07	4.07	3.90	3.90	4.01

Table B.6: Individual speedups for the rss heuristics on architecture 4-8

benchmark	ips	uls	La1uls	La2uls	La4uls	La3uls	La5uls
bubble	2.01	2.01	2.01	2.01	2.01	2.01	2.01
hanoi	1.65	1.65	1.65	1.65	1.47	1.65	1.65
heapsort	1.50	2.30	2.29	2.29	2.57	2.29	2.29
intmm	1.69	1.69	1.69	1.69	1.69	1.69	1.69
nsieve	1.91	3.12	3.12	3.12	3.12	3.12	3.12
perm	1.85	1.85	1.85	1.85	1.59	1.85	1.85
puzzle	0.00	1.14	1.14	1.14	1.14	1.14	1.14
queens	1.74	1.74	1.36	1.88	1.16	1.53	1.44
quick	1.31	2.02	2.02	2.02	2.02	2.02	2.02
loop1	2.24	2.24	2.24	2.24	1.64	2.24	2.24
loop2	0.92	6.06	6.06	6.06	3.15	6.06	6.06
loop3	1.75	1.75	1.75	1.75	1.40	1.75	1.75
loop4	1.78	1.78	1.78	1.78	1.39	1.78	1.78
loop5	2.54	2.54	2.54	2.54	1.32	2.54	2.36
loop6	6.48	6.48	6.48	6.48	4.21	6.48	6.00
loop7	2.47	2.47	2.47	2.47	1.60	2.47	2.47
loop9	1.31	4.26	4.26	4.26	4.26	4.26	3.08
loop10	1.49	6.40	6.40	6.40	6.40	6.40	6.40
loop11	1.70	1.70	1.70	1.70	1.31	1.70	1.70
loop12	1.78	1.78	1.78	1.78	1.45	1.78	1.78
loop13	1.44	4.39	4.39	4.39	4.42	4.42	4.55
loop14	1.42	6.20	6.20	6.20	6.20	6.20	6.20

Table B.7: Individual speedups for the rss heuristics on architecture 4-16

benchmark	ips	uls	La1uls	La2uls	La4uls	La3uls	La5uls
bubble	1.76	1.76	1.76	1.76	1.76	1.76	1.76
hanoi	1.65	1.65	1.65	1.65	1.47	1.65	1.65
heapsort	1.25	2.04	2.04	2.04	2.02	2.04	2.04
intmm	1.69	1.69	1.69	1.69	1.69	1.69	1.69
nsieve	1.39	1.65	1.65	1.65	1.65	1.65	1.65
perm	1.85	1.85	1.85	1.85	1.59	1.85	1.85
puzzle	0.00	1.13	1.13	1.13	1.13	1.13	1.13
queens	1.43	1.43	1.11	1.54	0.95	1.25	1.18
quick	1.56	1.75	1.75	1.75	1.75	1.75	1.75
loop1	2.24	2.24	2.24	2.24	1.64	2.24	2.24
loop2	2.50	2.50	2.50	2.50	1.30	2.50	2.50
loop3	1.75	1.75	1.75	1.75	1.40	1.75	1.75
loop4	1.78	1.78	1.78	1.78	1.39	1.78	1.78
loop5	2.54	2.54	2.54	2.54	1.32	2.54	2.36
loop6	1.81	1.81	1.81	1.81	1.18	1.81	1.67
loop7	2.47	2.47	2.47	2.47	1.60	2.47	2.47
loop9	1.68	1.68	1.68	1.68	1.68	1.68	1.68
loop10	2.43	2.43	2.43	2.43	2.43	2.43	2.43
loop11	1.70	1.70	1.70	1.70	1.31	1.70	1.70
loop12	1.78	1.78	1.78	1.78	1.45	1.78	1.78
loop13	3.87	3.87	3.87	3.87	3.87	3.87	3.87
loop14	2.03	2.03	2.03	2.03	2.03	2.03	2.03

Table B.8: Individual speedups for the rss heuristics on architecture 4-32

benchmark	ips	uls	La1uls	La2uls	La4uls	La3uls	La5uls
bubble	0.43	2.02	2.02	2.02	2.02	2.02	2.02
hanoi	0.00	1.65	1.65	1.65	1.47	1.65	1.65
heapsort	1.32	2.16	2.38	2.38	2.29	2.38	2.38
intmm	1.80	8.57	8.57	8.57	8.57	8.57	8.57
nsieve	1.66	1.60	1.60	1.60	1.60	1.60	1.60
perm	1.97	3.12	3.12	3.12	2.67	3.12	3.12
puzzle	1.04	1.54	1.54	1.54	1.54	1.54	1.54
queens	1.67	2.78	2.17	2.99	1.85	2.43	2.29
quick	1.31	5.87	5.87	5.87	5.87	5.87	5.87
loop1	1.39	5.09	5.09	5.09	3.73	5.09	5.09
loop2	1.26	6.69	6.69	6.69	4.46	6.69	6.69
loop3	0.70	1.75	1.75	1.75	1.40	1.75	1.75
loop4	1.33	5.79	5.79	5.79	4.50	5.79	5.79
loop5	1.23	4.27	4.27	4.27	4.78	4.42	3.73
loop6	1.22	6.86	6.86	6.86	4.45	6.86	3.34
loop7	1.19	4.00	4.00	4.00	4.15	3.95	4.05
loop9	1.24	2.16	2.14	2.14	2.46	2.46	2.94
loop10	1.24	4.02	4.19	4.19	4.55	4.55	4.25
loop11	0.74	1.70	1.70	1.70	1.31	1.70	1.70
loop12	1.78	1.78	1.78	1.78	1.45	1.78	1.78
loop13	0.88	2.87	2.73	2.73	2.96	2.96	3.07
loop14	1.29	4.14	4.14	4.14	4.01	4.01	3.95

Table B.9: Individual speedups for the rss heuristics on architecture 6-8

benchmark	ips	uls	La1uls	La2uls	La4uls	La3uls	La5uls
bubble	2.01	2.01	2.01	2.01	2.01	2.01	2.01
hanoi	1.65	1.65	1.65	1.65	1.47	1.65	1.65
heapsort	1.64	2.27	2.25	2.25	2.57	2.25	2.25
intmm	1.69	1.69	1.69	1.69	1.69	1.69	1.69
nsieve	1.91	3.12	3.12	3.12	3.12	3.12	3.12
perm	1.85	1.85	1.85	1.85	1.59	1.85	1.85
puzzle	0.00	1.14	1.14	1.14	1.14	1.14	1.14
queens	1.74	1.74	1.36	1.88	1.16	1.53	1.44
quick	1.36	2.02	2.02	2.02	2.02	2.02	2.02
loop1	2.24	2.24	2.24	2.24	1.64	2.24	2.24
loop2	0.94	6.79	6.79	6.79	3.15	6.79	6.79
loop3	1.75	1.75	1.75	1.75	1.40	1.75	1.75
loop4	1.78	1.78	1.78	1.78	1.39	1.78	1.78
loop5	0.36	2.54	2.54	2.54	1.32	2.54	2.54
loop6	6.48	6.48	6.48	6.48	4.21	6.48	6.00
loop7	0.51	2.52	2.52	2.52	1.60	2.52	2.52
loop9	1.43	3.08	3.16	3.16	3.08	3.08	2.91
loop10	1.55	6.83	6.83	6.83	6.83	6.83	6.83
loop11	1.70	1.70	1.70	1.70	1.31	1.70	1.70
loop12	1.78	1.78	1.78	1.78	1.45	1.78	1.78
loop13	1.42	4.66	4.66	4.66	4.66	4.66	4.69
loop14	1.31	6.20	6.20	6.20	6.20	6.20	6.20

Table B.10: Individual speedups for the rss heuristics on architecture 6-16

benchmark	ips	uls	La1uls	La2uls	La4uls	La3uls	La5uls
bubble	1.76	1.76	1.76	1.76	1.76	1.76	1.76
hanoi	1.65	1.65	1.65	1.65	1.47	1.65	1.65
heapsort	1.31	2.05	2.03	2.03	2.02	2.03	2.03
intmm	1.69	1.69	1.69	1.69	1.69	1.69	1.69
nsieve	1.39	1.65	1.65	1.65	1.65	1.65	1.65
perm	1.85	1.85	1.85	1.85	1.59	1.85	1.85
puzzle	0.00	1.13	1.13	1.13	1.13	1.13	1.13
queens	1.43	1.43	1.11	1.54	0.95	1.25	1.18
quick	1.56	1.75	1.75	1.75	1.75	1.75	1.75
loop1	2.24	2.24	2.24	2.24	1.64	2.24	2.24
loop2	2.80	2.80	2.80	2.80	1.30	2.80	2.80
loop3	1.75	1.75	1.75	1.75	1.40	1.75	1.75
loop4	1.78	1.78	1.78	1.78	1.39	1.78	1.78
loop5	2.54	2.54	2.54	2.54	1.32	2.54	2.54
loop6	1.81	1.81	1.81	1.81	1.18	1.81	1.67
loop7	2.52	2.52	2.52	2.52	1.60	2.52	2.52
loop9	1.72	1.72	1.72	1.72	1.72	1.72	1.72
loop10	2.59	2.59	2.59	2.59	2.59	2.59	2.59
loop11	1.70	1.70	1.70	1.70	1.31	1.70	1.70
loop12	1.78	1.78	1.78	1.78	1.45	1.78	1.78
loop13	4.05	5.15	5.15	5.15	5.15	5.15	4.03
loop14	2.03	2.03	2.03	2.03	2.03	2.03	2.03

Table B.11: Individual speedups for the rss heuristics on architecture 6-32

benchmark	es-ssra	Ses-ssra	p-rig	Sp-rig
bubble	1.16	1.26	1.16	1.27
hanoi	1.50	1.50	1.50	1.50
heapsort	1.21	1.09	1.14	1.06
intmm	1.30	1.29	1.26	1.17
nsieve	1.42	1.00	1.42	1.01
perm	1.77	1.56	1.20	1.18
puzzle	1.26	1.36	1.17	1.20
queens	1.29	0.81	1.17	0.66
quick	1.16	1.07	1.16	1.06
loop1	1.21	1.21	1.11	0.99
loop2	1.06	1.08	1.03	1.02
loop3	4.11	4.11	1.09	0.95
loop4	1.07	1.14	1.10	1.04
loop5	1.06	1.06	1.11	1.01
loop6	1.03	1.00	0.99	0.98
loop7	1.15	1.15	1.04	1.02
loop9	1.94	1.95	1.06	1.09
loop10	1.98	2.30	1.16	1.29
loop11	3.50	1.20	1.09	1.09
loop12	4.70	4.70	1.04	1.02
loop13	1.89	1.84	1.12	1.25
loop14	1.29	1.24	1.16	1.20

Table B.12: Individual speedups for the ssra heuristics on architecture 2-4

benchmark	es-ssra	Ses-ssra	p-rig	Sp-rig
bubble	1.79	1.79	1.15	0.35
hanoi	1.40	1.40	1.40	1.40
heapsort	1.30	0.99	1.63	1.07
intmm	2.38	5.59	1.47	1.90
nsieve	1.34	0.81	1.44	0.89
perm	3.00	3.00	1.41	2.60
puzzle	1.53	1.54	1.01	0.95
queens	1.63	1.80	1.24	0.78
quick	1.75	1.53	1.26	1.20
loop1	3.73	3.73	1.21	1.16
loop2	1.90	1.13	1.08	1.13
loop3	1.56	1.56	1.27	1.27
loop4	3.86	1.80	1.19	1.29
loop5	1.23	1.13	1.07	1.09
loop6	1.22	1.05	1.10	1.05
loop7	2.24	1.21	1.25	1.27
loop9	1.36	1.36	1.12	1.32
loop10	1.44	1.47	1.28	1.29
loop11	1.55	1.55	1.21	0.46
loop12	1.60	1.60	1.60	1.60
loop13	1.30	1.36	1.22	1.23
loop14	1.47	1.62	1.26	1.40

Table B.13: Individual speedups for the ssra heuristics on architecture 2-8

benchmark	es-ssra	Ses-ssra	p-rig	Sp-rig
bubble	1.78	1.78	1.46	1.07
hanoi	1.40	1.40	1.40	1.40
heapsort	1.76	1.92	1.26	0.80
intmm	1.69	1.69	1.69	1.69
nsieve	1.80	1.00	1.34	0.83
perm	1.78	1.78	1.78	1.78
puzzle	1.14	1.14	1.12	1.13
queens	1.70	1.70	1.20	1.29
quick	1.57	1.57	1.29	1.03
loop1	1.64	1.64	1.64	1.64
loop2	4.14	4.14	1.09	0.85
loop3	1.56	1.56	1.56	1.56
loop4	1.67	1.67	1.25	1.25
loop5	1.78	1.78	1.27	0.33
loop6	5.85	5.85	1.16	1.02
loop7	1.65	1.65	1.41	0.59
loop9	3.52	3.42	1.21	1.34
loop10	1.42	1.47	1.42	1.43
loop11	1.55	1.55	1.55	1.55
loop12	1.60	1.60	1.60	1.60
loop13	1.48	1.44	1.34	1.31
loop14	5.50	5.50	1.38	1.20

Table B.14: Individual speedups for the ssra heuristics on architecture 2-16

benchmark	es-ssra	Ses-ssra	p-rig	Sp-rig
bubble	1.56	1.56	1.56	1.56
hanoi	1.40	1.40	1.40	1.40
heapsort	1.91	1.91	1.30	0.81
intmm	1.69	1.69	1.69	1.69
nsieve	1.65	1.65	1.37	1.62
perm	1.78	1.78	1.78	1.78
puzzle	1.13	1.13	1.13	1.13
queens	1.39	1.39	1.39	1.39
quick	1.36	1.36	1.33	1.33
loop1	1.64	1.64	1.64	1.64
loop2	1.71	1.71	1.67	1.67
loop3	1.56	1.56	1.56	1.56
loop4	1.67	1.67	1.67	1.67
loop5	1.78	1.78	1.78	1.78
loop6	1.63	1.63	1.63	1.63
loop7	1.65	1.65	1.59	1.59
loop9	1.46	1.46	1.25	1.21
loop10	1.72	1.72	1.66	0.58
loop11	1.55	1.55	1.55	1.55
loop12	1.60	1.60	1.60	1.60
loop13	3.28	3.28	1.45	1.47
loop14	1.80	1.80	1.80	1.80

Table B.15: Individual speedups for the ssra heuristics on architecture 2-32

benchmark	es-ssra	Ses-ssra	p-rig	Sp-rig
bubble	1.16	1.26	1.16	1.64
hanoi	2.00	2.00	2.00	2.00
heapsort	1.22	1.10	1.15	1.08
intmm	1.30	1.29	1.26	1.18
nsieve	1.49	1.00	1.49	1.08
perm	1.83	1.60	1.23	1.22
puzzle	1.29	1.34	1.16	1.21
queens	1.30	0.83	1.17	0.67
quick	1.16	1.08	1.17	1.07
loop1	1.29	1.29	1.11	0.99
loop2	1.08	1.10	1.03	1.02
loop3	4.62	4.62	1.09	0.97
loop4	1.10	1.17	1.10	1.04
loop5	1.09	1.09	1.11	1.01
loop6	1.05	1.01	0.99	0.98
loop7	1.15	1.22	1.04	1.02
loop9	1.94	1.96	1.06	1.09
loop10	1.99	2.31	1.16	1.29
loop11	3.77	1.23	1.09	1.09
loop12	5.22	5.22	1.04	1.04
loop13	1.89	1.84	1.12	1.25
loop14	1.29	1.24	1.16	1.20

Table B.16: Individual speedups for the ssra heuristics on architecture 4-4

benchmark	es-ssra	Ses-ssra	p-rig	Sp-rig
bubble	2.02	2.02	1.15	0.35
hanoi	1.65	1.65	1.65	1.65
heapsort	1.49	1.02	1.72	1.11
intmm	2.79	6.74	1.70	1.54
nsieve	2.10	0.89	1.88	1.11
perm	3.12	3.12	1.65	2.79
puzzle	1.54	1.54	1.02	0.96
queens	1.74	1.80	1.43	0.87
quick	1.77	1.55	1.28	1.20
loop1	5.09	5.09	1.30	1.26
loop2	1.85	1.16	1.11	1.16
loop3	1.75	1.75	1.40	1.40
loop4	4.26	1.88	1.23	1.33
loop5	1.27	1.16	1.10	1.12
loop6	1.23	1.07	1.12	1.08
loop7	2.59	1.29	1.34	1.36
loop9	1.36	1.36	1.12	1.32
loop10	1.45	1.48	1.29	1.30
loop11	1.70	1.70	1.42	0.47
loop12	1.78	1.78	1.78	1.78
loop13	1.34	1.39	1.25	1.26
loop14	1.49	1.65	1.29	1.44

Table B.17: Individual speedups for the ssra heuristics on architecture 4-8

benchmark	es-ssra	Ses-ssra	p-rig	Sp-rig
bubble	2.01	2.01	1.78	1.07
hanoi	1.65	1.65	1.65	1.65
heapsort	1.92	1.98	1.41	0.82
intmm	1.69	1.69	1.69	1.69
nsieve	2.24	1.00	2.10	0.91
perm	1.85	1.85	1.85	1.85
puzzle	1.14	1.14	1.13	1.14
queens	1.74	1.74	1.20	1.30
quick	1.86	1.86	1.07	1.15
loop1	2.24	2.24	2.24	2.24
loop2	6.06	6.06	1.13	0.89
loop3	1.75	1.75	1.75	1.75
loop4	1.78	1.78	1.39	1.39
loop5	2.54	2.54	1.50	0.35
loop6	7.27	7.27	1.17	1.04
loop7	2.47	2.47	1.73	0.64
loop9	3.52	3.42	1.22	1.41
loop10	1.45	1.52	1.44	1.44
loop11	1.70	1.70	1.70	1.70
loop12	1.78	1.78	1.78	1.78
loop13	1.60	1.66	1.42	1.38
loop14	6.20	6.20	1.50	1.31

Table B.18: Individual speedups for the ssra heuristics on architecture 4-16

benchmark	es-ssra	Ses-ssra	p-rig	Sp-rig
bubble	1.76	1.76	1.76	1.76
hanoi	1.65	1.65	1.65	1.65
heapsort	2.01	2.01	1.38	0.84
intmm	1.69	1.69	1.69	1.69
nsieve	1.65	1.65	1.37	1.62
perm	1.85	1.85	1.85	1.85
puzzle	1.13	1.13	1.13	1.13
queens	1.43	1.43	1.43	1.43
quick	1.61	1.61	1.61	1.61
loop1	2.24	2.24	2.24	2.24
loop2	2.50	2.50	2.41	2.41
loop3	1.75	1.75	1.75	1.75
loop4	1.78	1.78	1.78	1.78
loop5	2.54	2.54	2.54	2.54
loop6	2.03	2.03	2.03	2.03
loop7	2.47	2.47	2.00	2.00
loop9	1.68	1.68	1.25	1.21
loop10	2.43	2.43	1.92	0.62
loop11	1.70	1.70	1.70	1.70
loop12	1.78	1.78	1.78	1.78
loop13	3.88	3.88	1.62	1.71
loop14	2.03	2.03	2.03	2.03

Table B.19: Individual speedups for the ssra heuristics on architecture 4-32

benchmark	es-ssra	Ses-ssra	p-rig	Sp-rig
bubble	2.02	2.02	1.15	0.35
hanoi	1.65	1.65	1.65	1.65
heapsort	1.49	1.02	1.72	1.11
intmm	2.79	6.74	1.70	1.54
nsieve	2.10	0.89	1.88	1.11
perm	3.12	3.12	1.65	2.79
puzzle	1.54	1.54	1.02	0.96
queens	1.74	1.80	1.43	0.87
quick	1.77	1.55	1.28	1.25
loop1	5.09	5.09	1.30	1.26
loop2	1.85	1.16	1.11	1.16
loop3	1.75	1.75	1.40	1.40
loop4	4.26	1.88	1.23	1.33
loop5	1.27	1.16	1.10	1.12
loop6	1.23	1.07	1.12	1.08
loop7	2.59	1.29	1.34	1.36
loop9	1.36	1.36	1.12	1.32
loop10	1.45	1.48	1.29	1.30
loop11	1.70	1.70	1.42	0.47
loop12	1.78	1.78	1.78	1.78
loop13	1.34	1.39	1.25	1.26
loop14	1.49	1.65	1.29	1.44

Table B.20: Individual speedups for the ssra heuristics on architecture 6-8

benchmark	es-ssra	Ses-ssra	p-rig	Sp-rig
bubble	2.01	2.01	1.78	1.07
hanoi	1.65	1.65	1.65	1.65
heapsort	1.93	1.98	1.46	0.83
intmm	1.69	1.69	1.69	1.69
nsieve	2.24	1.00	2.10	0.91
perm	1.85	1.85	1.85	1.85
puzzle	1.14	1.14	1.13	1.14
queens	1.74	1.74	1.20	1.30
quick	2.02	1.86	1.07	1.15
loop1	2.24	2.24	2.24	2.24
loop2	6.79	6.79	1.13	0.89
loop3	1.75	1.75	1.75	1.75
loop4	1.78	1.78	1.39	1.39
loop5	2.54	2.54	1.53	0.36
loop6	7.27	7.27	1.18	1.05
loop7	2.52	2.52	1.73	0.64
loop9	3.52	3.42	1.22	1.41
loop10	1.45	1.53	1.44	1.44
loop11	1.70	1.70	1.70	1.70
loop12	1.78	1.78	1.78	1.78
loop13	1.61	1.68	1.42	1.39
loop14	6.20	6.20	1.53	1.32

Table B.21: Individual speedups for the ssra heuristics on architecture 6-16

benchmark	es-ssra	Ses-ssra	p-rig	Sp-rig
bubble	1.76	1.76	1.76	1.76
hanoi	1.65	1.65	1.65	1.65
heapsort	2.02	2.02	1.38	0.84
intmm	1.69	1.69	1.69	1.69
nsieve	1.65	1.65	1.37	1.62
perm	1.85	1.85	1.85	1.85
puzzle	1.13	1.13	1.13	1.13
queens	1.43	1.43	1.43	1.43
quick	1.75	1.61	1.61	1.61
loop1	2.24	2.24	2.24	2.24
loop2	2.80	2.80	2.41	2.41
loop3	1.75	1.75	1.75	1.75
loop4	1.78	1.78	1.78	1.78
loop5	2.54	2.54	2.54	2.54
loop6	2.03	2.03	2.03	2.03
loop7	2.52	2.52	2.00	2.00
loop9	1.72	1.72	1.25	1.21
loop10	2.59	2.59	1.92	0.62
loop11	1.70	1.70	1.70	1.70
loop12	1.78	1.78	1.78	1.78
loop13	4.05	4.05	1.62	1.74
loop14	2.03	2.03	2.03	2.03

Table B.22: Individual speedups for the ssra heuristics on architecture 6-32

benchmark	Gursa	GDursa	Eursa	Nursa	Dursa	ursa
bubble	3.52	3.52	3.53	3.53	3.53	3.53
hanoi	2.35	2.35	2.35	2.35	2.35	2.35
heapsort	2.01	2.01	1.90	1.90	1.90	1.90
intmm	3.19	3.19	3.91	3.91	3.91	3.91
nsieve	4.56	4.56	4.56	4.56	4.56	4.56
perm	3.82	3.82	3.69	3.69	3.69	3.69
puzzle	1.90	1.90	1.89	1.90	1.89	1.89
queens	1.70	1.70	1.08	1.08	1.08	1.08
quick	2.39	2.39	2.12	2.11	2.12	2.12
loop1	3.08	3.08	2.82	2.82	2.82	2.82
loop2	3.50	3.50	1.86	1.96	2.11	2.23
loop3	5.28	5.28	5.28	5.28	5.28	5.28
loop4	2.69	2.69	2.28	2.28	2.28	2.28
loop5	2.81	2.91	2.66	2.66	2.66	2.55
loop6	3.02	3.02	2.42	2.49	2.42	2.42
loop7	2.54	2.38	2.22	2.24	2.32	2.45
loop9	3.33	3.15	2.85	2.85	2.85	2.93
loop10	3.56	3.34	3.36	3.41	3.36	3.81
loop11	2.72	2.72	2.72	2.72	2.72	2.72
loop12	5.87	5.87	5.22	5.22	5.22	5.22
loop13	3.65	3.57	3.68	3.76	3.61	3.30
loop14	2.90	2.82	2.74	2.74	2.74	2.74

Table B.23: Individual speedups for the ursa heuristics on architecture 2-4

benchmark	Gursa	GDursa	Eursa	Nursa	Dursa	ursa
bubble	2.00	2.00	2.00	2.00	2.00	2.00
hanoi	1.22	1.22	1.22	1.22	1.22	1.22
heapsort	2.42	2.42	2.52	2.52	2.52	2.52
intmm	9.73	9.73	8.49	8.49	8.49	8.49
nsieve	3.10	3.10	3.10	3.10	3.10	3.10
perm	2.81	2.81	2.81	2.81	2.81	2.81
puzzle	1.81	1.81	1.81	1.81	1.81	1.81
queens	1.77	1.77	1.76	1.76	1.76	1.76
quick	4.18	4.18	4.18	4.18	4.18	4.18
loop1	3.90	3.90	3.57	3.57	3.57	3.57
loop2	4.23	4.63	3.44	3.44	3.44	3.44
loop3	2.00	2.00	2.00	2.00	2.00	2.00
loop4	5.05	5.05	5.06	5.06	5.06	5.06
loop5	3.51	3.51	4.05	4.05	3.41	3.41
loop6	4.62	4.62	3.58	3.58	3.79	3.79
loop7	3.17	3.17	3.33	3.33	3.33	3.33
loop9	3.21	2.74	2.41	2.41	2.26	2.48
loop10	3.02	3.02	2.54	2.65	2.54	2.49
loop11	1.89	1.89	1.70	1.70	1.70	1.70
loop12	2.00	2.00	1.78	1.78	1.78	1.78
loop13	2.78	2.73	2.75	2.48	2.76	2.73
loop14	5.32	5.32	4.68	4.68	4.60	4.60

Table B.24: Individual speedups for the ursa heuristics on architecture 2-8

benchmark	Gursa	GDursa	Eursa	Nursa	Dursa	ursa
bubble	1.99	1.99	1.99	1.99	1.99	1.99
hanoi	1.22	1.22	1.22	1.22	1.22	1.22
heapsort	2.26	2.26	2.34	2.34	2.34	2.34
intmm	1.93	1.93	1.68	1.68	1.68	1.68
nsieve	4.75	4.75	4.76	4.76	4.76	4.76
perm	1.67	1.67	1.67	1.67	1.67	1.67
puzzle	1.33	1.33	1.33	1.33	1.33	1.33
queens	1.35	1.35	1.22	1.22	1.22	1.22
quick	1.18	1.18	1.18	1.18	1.18	1.18
loop1	1.72	1.72	1.57	1.57	1.57	1.57
loop2	3.33	3.33	3.54	3.54	3.54	3.54
loop3	2.00	2.00	2.00	2.00	2.00	2.00
loop4	1.56	1.56	1.56	1.56	1.56	1.56
loop5	1.27	1.27	1.32	1.32	1.32	1.32
loop6	4.80	4.80	4.80	4.80	4.80	4.80
loop7	1.50	1.50	1.45	1.45	1.45	1.45
loop9	3.52	3.59	3.39	3.39	3.39	3.39
loop10	3.17	3.17	3.55	3.55	3.55	3.55
loop11	1.89	1.89	1.70	1.70	1.70	1.70
loop12	2.00	2.00	1.78	1.78	1.78	1.78
loop13	3.18	2.99	2.71	2.71	2.71	3.03
loop14	4.30	4.30	3.71	3.71	3.71	3.71

Table B.25: Individual speedups for the ursa heuristics on architecture 2-16

benchmark	Gursa	GDursa	Eursa	Nursa	Dursa	ursa
bubble	1.74	1.74	1.74	1.74	1.74	1.74
hanoi	1.22	1.22	1.22	1.22	1.22	1.22
heapsort	1.67	1.67	1.73	1.73	1.73	1.73
intmm	1.93	1.93	1.68	1.68	1.68	1.68
nsieve	2.52	2.52	2.52	2.52	2.52	2.52
perm	1.67	1.67	1.67	1.67	1.67	1.67
puzzle	1.32	1.32	1.32	1.32	1.32	1.32
queens	1.10	1.10	1.00	1.00	1.00	1.00
quick	1.02	1.02	1.02	1.02	1.02	1.02
loop1	1.72	1.72	1.57	1.57	1.57	1.57
loop2	1.37	1.37	1.46	1.46	1.46	1.46
loop3	2.00	2.00	2.00	2.00	2.00	2.00
loop4	1.56	1.56	1.56	1.56	1.56	1.56
loop5	1.27	1.27	1.32	1.32	1.32	1.32
loop6	1.34	1.34	1.34	1.34	1.34	1.34
loop7	1.50	1.50	1.45	1.45	1.45	1.45
loop9	1.39	1.39	1.47	1.47	1.47	1.47
loop10	1.67	1.67	1.68	1.68	1.68	1.68
loop11	1.89	1.89	1.70	1.70	1.70	1.70
loop12	2.00	2.00	1.78	1.78	1.78	1.78
loop13	3.71	3.71	3.51	3.51	3.51	3.51
loop14	1.40	1.40	1.21	1.21	1.21	1.21

Table B.26: Individual speedups for the ursa heuristics on architecture 2-32

benchmark	Gursa	GDursa	Eursa	Nursa	Dursa	ursa
bubble	3.90	3.90	3.90	3.90	3.90	3.90
hanoi	3.18	3.18	3.18	3.18	3.18	3.18
heapsort	2.06	2.06	2.35	2.35	2.35	2.35
intmm	3.36	3.36	3.53	3.53	3.53	3.53
nsieve	4.56	4.56	4.56	4.56	4.56	4.56
perm	4.64	4.64	4.45	4.45	4.45	4.45
puzzle	0.00	0.00	1.93	1.93	1.93	1.93
queens	1.92	1.92	1.53	1.53	1.53	1.53
quick	2.41	2.41	2.01	2.01	2.01	2.01
loop1	3.57	3.57	3.03	3.03	3.03	3.03
loop2	4.30	4.30	2.13	2.13	2.29	2.50
loop3	6.16	6.16	6.16	6.16	6.16	6.16
loop4	2.87	2.87	2.96	2.96	2.96	2.96
loop5	2.84	3.29	2.66	2.66	3.09	2.91
loop6	3.10	3.10	2.54	2.89	2.54	2.54
loop7	3.05	3.32	3.02	2.94	3.28	3.02
loop9	3.29	3.29	3.16	3.16	3.08	3.16
loop10	4.37	3.81	4.08	3.43	3.48	4.41
loop11	4.08	4.08	4.08	4.08	4.08	4.08
loop12	6.71	6.71	6.71	6.71	6.71	6.71
loop13	4.18	3.82	4.18	4.25	3.90	3.63
loop14	3.37	3.49	2.91	2.97	2.73	2.91

Table B.27: Individual speedups for the ursa heuristics on architecture 4-4

benchmark	Gursa	GDursa	Eursa	Nursa	Dursa	ursa
bubble	2.67	2.67	2.67	2.67	2.67	2.67
hanoi	1.65	1.65	1.65	1.65	1.65	1.65
heapsort	3.27	3.27	3.60	3.60	3.60	3.60
intmm	9.73	9.73	9.73	9.73	9.73	9.73
nsieve	3.10	3.10	3.10	3.10	3.10	3.10
perm	3.65	3.65	3.65	3.65	3.65	3.65
puzzle	1.81	1.81	1.81	1.81	1.81	1.81
queens	1.78	1.78	1.63	1.63	1.63	1.63
quick	5.79	5.79	4.97	4.97	4.97	4.97
loop1	6.45	6.45	6.45	6.45	6.45	6.45
loop2	5.35	6.34	4.30	4.30	4.30	4.30
loop3	2.33	2.33	2.33	2.33	2.33	2.33
loop4	8.08	8.08	8.08	8.08	8.08	8.08
loop5	4.59	4.59	4.78	4.78	4.78	4.78
loop6	5.18	5.18	4.62	4.70	4.62	4.62
loop7	4.78	4.78	4.85	4.85	4.85	4.85
loop9	3.05	3.33	2.41	2.41	2.79	2.46
loop10	3.85	3.85	3.20	3.13	3.20	3.13
loop11	2.12	2.12	2.12	2.12	2.12	2.12
loop12	2.28	2.28	2.28	2.28	2.28	2.28
loop13	3.15	3.30	3.23	3.30	3.34	3.23
loop14	7.13	7.13	6.60	6.60	6.02	6.02

Table B.28: Individual speedups for the ursa heuristics on architecture 4-8

benchmark	Gursa	GDursa	Eursa	Nursa	Dursa	ursa
bubble	2.66	2.66	2.66	2.66	2.66	2.66
hanoi	1.65	1.65	1.65	1.65	1.65	1.65
heapsort	2.70	2.70	2.81	2.81	2.81	2.81
intmm	1.94	1.94	1.94	1.94	1.94	1.94
nsieve	4.76	4.76	4.76	4.76	4.76	4.76
perm	2.17	2.17	2.17	2.17	2.17	2.17
puzzle	1.35	1.35	1.35	1.35	1.35	1.35
queens	1.86	1.86	1.86	1.86	1.86	1.86
quick	1.98	1.98	1.98	1.98	1.98	1.98
loop1	2.84	2.84	2.84	2.84	2.84	2.84
loop2	4.47	4.47	5.15	5.15	5.15	5.15
loop3	2.33	2.33	2.33	2.33	2.33	2.33
loop4	2.49	2.49	2.49	2.49	2.49	2.49
loop5	2.06	2.06	1.74	1.74	1.74	1.74
loop6	6.31	6.31	5.85	5.85	5.85	5.85
loop7	2.54	2.54	2.19	2.19	2.19	2.19
loop9	3.74	3.98	3.63	3.63	3.63	3.63
loop10	4.14	4.14	4.14	4.14	4.14	4.14
loop11	2.12	2.12	2.12	2.12	2.12	2.12
loop12	2.28	2.28	2.28	2.28	2.28	2.28
loop13	4.17	3.89	3.75	3.75	3.75	4.20
loop14	6.29	6.29	6.29	6.29	6.29	6.29

Table B.29: Individual speedups for the ursa heuristics on architecture 4-16

benchmark	Gursa	GDursa	Eursa	Nursa	Dursa	ursa
bubble	2.32	2.32	2.32	2.32	2.32	2.32
hanoi	1.65	1.65	1.65	1.65	1.65	1.65
heapsort	2.22	2.22	2.21	2.21	2.21	2.21
intmm	1.94	1.94	1.94	1.94	1.94	1.94
nsieve	2.52	2.52	2.52	2.52	2.52	2.52
perm	2.17	2.17	2.17	2.17	2.17	2.17
puzzle	1.33	1.33	1.33	1.33	1.33	1.33
queens	1.53	1.53	1.53	1.53	1.53	1.53
quick	1.71	1.71	1.71	1.71	1.71	1.71
loop1	2.84	2.84	2.84	2.84	2.84	2.84
loop2	1.84	1.84	2.12	2.12	2.12	2.12
loop3	2.33	2.33	2.33	2.33	2.33	2.33
loop4	2.49	2.49	2.49	2.49	2.49	2.49
loop5	2.06	2.06	1.74	1.74	1.74	1.74
loop6	1.76	1.76	1.63	1.63	1.63	1.63
loop7	2.54	2.54	2.19	2.19	2.19	2.19
loop9	1.56	1.56	1.56	1.56	1.56	1.56
loop10	2.09	2.09	2.13	2.13	2.13	2.13
loop11	2.12	2.12	2.12	2.12	2.12	2.12
loop12	2.28	2.28	2.28	2.28	2.28	2.28
loop13	4.94	4.94	4.94	4.94	4.94	4.94
loop14	2.06	2.06	2.06	2.06	2.06	2.06

Table B.30: Individual speedups for the ursa heuristics on architecture 4-32

benchmark	Gursa	GDursa	Eursa	Nursa	Dursa	ursa
bubble	2.67	2.67	2.67	2.67	2.67	2.67
hanoi	1.65	1.65	1.65	1.65	1.65	1.65
heapsort	3.29	3.29	3.61	3.61	3.61	3.61
intmm	9.73	9.73	9.73	9.73	9.73	9.73
nsieve	3.10	3.10	3.10	3.10	3.10	3.10
perm	4.05	4.05	4.05	4.05	4.05	4.05
puzzle	1.82	1.82	1.82	1.82	1.82	1.82
queens	1.78	1.78	1.63	1.63	1.63	1.63
quick	6.05	6.05	5.36	5.36	5.36	5.36
loop1	6.45	6.45	6.45	6.45	6.45	6.45
loop2	6.17	6.34	5.60	5.60	5.60	5.60
loop3	2.33	2.33	2.33	2.33	2.33	2.33
loop4	8.09	8.09	8.10	8.10	8.10	8.10
loop5	4.59	4.59	4.68	4.68	4.68	4.68
loop6	5.40	5.40	6.34	6.34	6.34	6.34
loop7	5.24	5.24	5.69	5.69	5.69	5.69
loop9	3.19	3.16	2.43	2.53	2.29	2.61
loop10	3.63	3.35	3.44	4.06	3.44	3.35
loop11	2.12	2.12	2.12	2.12	2.12	2.12
loop12	2.28	2.28	2.28	2.28	2.28	2.28
loop13	3.07	3.12	3.45	3.49	3.02	3.37
loop14	7.74	7.74	6.77	6.77	6.15	6.15

Table B.31: Individual speedups for the ursa heuristics on architecture 6-8

benchmark	Gursa	GDursa	Eursa	Nursa	Dursa	ursa
bubble	2.66	2.66	2.66	2.66	2.66	2.66
hanoi	1.65	1.65	1.65	1.65	1.65	1.65
heapsort	2.91	2.91	2.91	2.91	2.91	2.91
intmm	1.94	1.94	1.94	1.94	1.94	1.94
nsieve	4.76	4.76	4.76	4.76	4.76	4.76
perm	2.40	2.40	2.40	2.40	2.40	2.40
puzzle	1.35	1.35	1.35	1.35	1.35	1.35
queens	1.88	1.88	1.88	1.88	1.88	1.88
quick	2.02	2.02	2.02	2.02	2.02	2.02
loop1	2.84	2.84	2.84	2.84	2.84	2.84
loop2	5.66	5.66	6.29	6.29	6.29	6.29
loop3	2.33	2.33	2.33	2.33	2.33	2.33
loop4	2.49	2.49	2.49	2.49	2.49	2.49
loop5	2.13	2.13	2.06	2.06	2.06	2.06
loop6	7.05	7.05	7.05	7.05	7.05	7.05
loop7	2.77	2.77	2.89	2.89	2.89	2.89
loop9	3.94	4.52	3.70	3.70	3.70	3.70
loop10	4.55	4.55	4.47	4.47	4.47	4.47
loop11	2.12	2.12	2.12	2.12	2.12	2.12
loop12	2.28	2.28	2.28	2.28	2.28	2.28
loop13	4.32	4.42	4.05	3.82	3.87	3.72
loop14	6.29	6.29	6.09	6.09	6.09	6.09

Table B.32: Individual speedups for the ursa heuristics on architecture 6-16

benchmark	Gursa	GDursa	Eursa	Nursa	Dursa	ursa
bubble	2.32	2.32	2.32	2.32	2.32	2.32
hanoi	1.65	1.65	1.65	1.65	1.65	1.65
heapsort	2.29	2.29	2.29	2.29	2.29	2.29
intmm	1.94	1.94	1.94	1.94	1.94	1.94
nsieve	2.52	2.52	2.52	2.52	2.52	2.52
perm	2.40	2.40	2.40	2.40	2.40	2.40
puzzle	1.33	1.33	1.33	1.33	1.33	1.33
queens	1.54	1.54	1.54	1.54	1.54	1.54
quick	1.75	1.75	1.75	1.75	1.75	1.75
loop1	2.84	2.84	2.84	2.84	2.84	2.84
loop2	2.33	2.33	2.59	2.59	2.59	2.59
loop3	2.33	2.33	2.33	2.33	2.33	2.33
loop4	2.49	2.49	2.49	2.49	2.49	2.49
loop5	2.13	2.13	2.06	2.06	2.06	2.06
loop6	1.97	1.97	1.97	1.97	1.97	1.97
loop7	2.77	2.77	2.89	2.89	2.89	2.89
loop9	1.77	1.77	1.77	1.77	1.77	1.77
loop10	2.46	2.46	2.40	2.40	2.40	2.40
loop11	2.12	2.12	2.12	2.12	2.12	2.12
loop12	2.28	2.28	2.28	2.28	2.28	2.28
loop13	5.29	5.29	4.86	4.86	4.86	4.86
loop14	2.06	2.06	1.99	1.99	1.99	1.99

Table B.33: Individual speedups for the ursa heuristics on architecture 6-32

Bibliography

[AN88] Alexander Aiken and Alexandru Nicolau. A development environment for horizontal microcode. *IEEE Trans. on Software Engineering*, 14(5):584–594, 1988.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, 1986.

[ATGLR93] Ali-Reza Adl-Tabatabai, Thomas Gross, Guei-Yuan Lueh, and James Reinders. Modelling instruction-level parallelism for software pipelining. In *Proc. of IFIP WG 10.3 Working Conference on Architectures and Compliation Techniques for Fine and Medium Grain Parallelism*, pages 321–330, 1993.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. In *Conf. Rec. 15th ACM Symp. on Prin. of Programming Languages*, pages 1–11, 1988.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *Proc. of Sigplan '89 Conf. on Programming Language Design and Implementation*, pages 275–284, 1989.

[BCT92] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proc. of Sigplan '92 Conf. on Programming Language Design and Implementation*, pages 311–321, 1992.

[BEH91] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proc. of 4th International Conf. on ASPLOS*, pages 122–131, 1991.

[BGM+89] David Bernstein, Martin C. Golumbic, Yashay Mansour, Ron Y. Pinter, Dina Q. Goldin, Hugo Krawczyk, and Itai Nahshon. Spill code minimization techniques for optimizing compilers. In *Proc. of Sigplan '89 Conf. on Programming Language Design and Implementation*, pages 258–263, 1989.

[BMO90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proc. of Sigplan '90 Conf. on Programming Language Design and Implementation*, pages 257–271, 1990.

[BR91] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. In *Proc. of Sigplan '91 Conf. on Programming Language Design and Implementation*, pages 241–255, 1991.

[Bri92] Preston Briggs. Register allocation via graph coloring. Technical Report Ph.D. Dissertation, Department of Computer Science, Rice University, April 1992.

[CAC+81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–58, 1981.

[CH90] F. Chow and J. Hennessy. Register allocation by priority-based coloring. *ACM Trans. Prog. Lang. and Systems*, 12(4):501–536, 1990.

[Cha82] G.J. Chaitin. Register allocation & spilling via graph coloring. In *Proc. of ACM Sigplan '82 Symp. on Compiler Construction*, pages 201–207, 1982.

[CK91] David Callahan and Brain Koblenz. Register allocation via heirachical graph coloring. In *Proc. of Sigplan '91 Conf. on Programming Language Design and Implementation*, pages 192–203, 1991.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., 1990.

[DHB89] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt. Overlapped loop support in the cydra 5. In *Proc. of 3rd International Conf. on ASPLOS*, pages 26–39, 1989.

[Dil50] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annuals of Mathematics*, 51:161–166, 1950.

[Dow94] Chyi-Ren Dow. Pivot: A program parallelization and visualization environment. Technical Report Technical Report 94-22, Ph.D. Dissertation, University of Pittsburgh, Computer Science Department, 1994.

[EN89] Kemal Ebcioğlu and Alexandru Nicolau. A *global* resource-constrained parallelization technique. In *Proc. of ACM SIGARCH ICS-89: International Conf. on Supercomputing*, 1989.

[FF65] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, N.J., 1965.

[Fie92] Claude-Nicolas Fiechter. *PDG C Compiler*. University of Pittsburgh, 1992.

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

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. and Systems*, 9(3):319–349, 1987.

[GH88] James R. Goodman and Wie-Chung Hsu. Code scheduling and register allocation in large basic blocks. In *Proc. of ACM Supercomputing Conf.*, pages 442–452, 1988.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completenesss*. W. H. Freeman and Co., New York, New York, 1979.

[GS90] Rajiv Gupta and Mary Lou Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Trans. on Software Engineering*, 16(4):421–431, 1990.

[HD86] Peter Y.T. Hsu and Edward S. Davidson. Highly concurrent scalar processing. In *Proc. of 13th Annual International Symp. on Computer Architecture*, pages 386–395, 1986.

[HMC+93] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The Superblock: An effective technique for VLIW and superscalar compliation. In *The Journal of Supercomputing*, volume A, pages 229–248, 1993.

[HP87] Wen-mei W. Hwu and Yale N. Patt. Checkpoint repair for out-of-order execution machines. In *Proc. of 14th Annual International Symp. on Computer Architecture*, pages 18–26, 1987.

[Hsu87] Wei-Chung Hsu. Register allocation and code scheduling for load/store architectures. Technical Report Computer Sciences TR #722, Ph.D. Dissertation, University of Wisconsin-Madison, 1987.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proc. of Sigplan '93 Conf. on Programming Language Design and Implementation*, pages 78–89, 1993.

[JPP94] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proc. of Sigplan '94 Conf. on Programming Language Design and Implementation*, pages 171–185, 1994.

[Kar72] R. J. Karp. *Reducibility Among Combinatorial Problems*, pages 85–103. Plenum Press, New York, 1972.

[KH93] Priyadarshan Kolte and Mary Jean Harrold. Load/store range analysis for global register allocation. In *Proc. of Sigplan '93 Conf. on Programming Language Design and Implementation*, pages 268–277, June 1993.

[ME92] Soo-Mook Moon and Kemal Ebcioğlu. An efficient resource-constrained global scheduling technique for superscalar and vliw processors. Technical Report Computer Science Research Report RC 17962 (#78691), IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1992.

[MGS92] Brian Malloy, Rajiv Gupta, and Mary Lou Soffa. A shape matching approach for scheduling fine-grained parallelism. In *Proc. of 25th Annual International Symp. on Microarchitecture*, pages 264–267, 1992.

[MLC+92] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proc. of 25th Annual International Symp. on Microarchitecture*, pages 45–54, 1992.

[NG93] Qi Ning and Guang R. Gao. A novel framework of register allocation for software pipelining. In *Conf. Rec. 20th ACM Symp. on Prin. of Programming Languages*, pages 29–42, 1993.

[Nor95] Cindy Norris. *Cooperative Register Allocation and Instruction Scheduling.* PhD thesis, University of Delaware, May 1995.

[NP93] Cindy Norris and Lori Pollock. A scheduler-sensitive global register allocator. In *Proc. of Supercomputing '93*, pages 804–813, 1993.

[NP94] Cindy Norris and Lori L. Pollock. Register allocation over the program dependence graph. In *Proc. of Sigplan '94 Conf. on Programming Language Design and Implementation*, pages 266–277, 1994.

[PBJ+91] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. In *Conf. Rec. 18th ACM Symp. on Prin. of Programming Languages*, pages 67–78, 1991.

[PF92] Todd A. Proebsting and Charles N. Fischer. Probablistic register allocation. In *Proc. of Sigplan '92 Conf. on Programming Language Design and Implementation*, pages 300–310, 1992.

[Pin93] Shlomit S. Pinter. Register allocation with instruction scheduling: A new approach. In *Proc. of Sigplan '93 Conf. on Programming Language Design and Implementation*, pages 248–257, 1993.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conf. Rec. 15th ACM Symp. on Prin. of Programming Languages*, pages 12–27, 1988.

[SHL92] Michael D. Smith, Mark Horowitz, and Monica Lam. Efficient superscalar performance through boosting. In *Proc. of 5th International Conf. on ASPLOS*, pages 248–259, 1992.

[Tar83] Robert E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

[WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *Conf. Rec. 21st ACM Symp. on Prin. of Programming Languages*, pages 297–310, 1994.

[Whi91] Deborah Lynn Whitfield. A unifying framework for optimizing transformations. Technical Report TR 91-24, Ph.D. Dissertation, University of Pittsburgh, 1991.

[WS91] Deborah Whitfield and Mary Lou Soffa. Automatic generation of global optimizers. In *Proc. of Sigplan '91 Conf. on Programming Language Design and Implementation*, pages 120–129, 1991.