Compilation Support for Superscalar Processors

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

by Mikko H. Lipasti

Department of Electrical and Computer Engineering Carnegie Mellon University Pittsburgh, PA 15213

December 1992
Abstract

This thesis describes work done in two areas of compilation support for superscalar processors; register allocation and instruction scheduling. Chapter 1 describes an approach to register allocation for superscalar processors that supports dynamic and speculative out-of-order execution of instructions and guarantees precise interrupts without expensive hardware for managing register usage and maintaining an in-order processor state. The approach is called extended register allocation, and is based on a graph-coloring paradigm for storage allocation first introduced by Chaitin in [2].

Chapter 2 presents a novel approach to performing aggressive instruction scheduling in the context of the superscalar IBM RS/6000 processor architecture[4, 5]. The approach seeks to enhance the instruction-level parallelism visible to the processor by speculatively moving instructions across conditional branches at compile-time, and taking appropriate measures to preserve correct program semantics. Results are presented which indicate that speedups of up to 6% are achievable on the existing RS/6000 implementation, while performance gains of up to 54% are possible with simple extensions to the current implementation in conjunction with the aggressive instruction scheduler that has been implemented.

Chapter 3 explores the interaction of the register allocation and instruction scheduling [35, 42], and makes an attempt at developing a better understanding of the underlying interdependencies between the two techniques. A novel framework for integrating the two techniques, based on the ideas presented and the concept of coagulation [41, 40] is also presented.
Contents

CHAPTER 1  
Register Allocation for Superscalar Processors  1  
Concepts and Definitions  2  
Definitions  2  
Difficulties with the Sequential Execution Model  3  
Application to Out-of-order Execution  4  
Chaitin’s Heuristic for Register Allocation  4  
Extended Register Allocation Scheme  5  
Practical Extended Register Allocation  8  
Application to Speculative Execution  9  

CHAPTER 2  
Instruction Scheduling for Superscalar Processors  11  
Introduction  12  
Background and Justification  13  
Machine Model  15  
Scheduler Operation  17  
Scheduling Framework  18  
List Scheduling Algorithm  18  
Preserving Semantic Correctness  20  
Results  23  
Existing Machine Model  24  
Augmented Machine Models  25  
Future Work  26  
Existing Approaches  28  

Compilation Support for Superscalar Processors  iii
CHAPTER 3

Interaction of Register Allocation and Instruction Scheduling  28

Register Allocation  29
The Interference Graph  30
Definition  30
Causes of Variable Interference  31
Integrating Register Allocation and Scheduling  33
Coagulation  33
Modified Coagulation Framework  34
Advantages of Integration  35

List of References 36
List of Figures and Tables

FIGURE 1. Types of Speculative Code Motion 14
FIGURE 2. Configurable Machine Model 15
FIGURE 3. Execution Pipelines with Forwarding Paths 16
FIGURE 4. Scheduling Example 19
FIGURE 5. Sample Dependence Graph 20
FIGURE 6. Example of Register Copy Creation 22
FIGURE 7. Example of Instruction Replication 23
FIGURE 8. Dispatch Stalls by Hazard Type 25
FIGURE 9. Non-essential and Essential Interferences 32

TABLE 1. Speedups for Existing Machine Model 24
TABLE 2. Speedups For Augmented Machine Models 26
This chapter describes an approach to register allocation for superscalar processors that supports dynamic and speculative out-of-order execution of instructions and guarantees precise interrupts. It does so while eliminating the need for dynamic techniques for register renaming, preservation of an inorder processor state, and nullifying the effects of speculatively executed instructions.

Johnson [1] suggests that significant performance improvement can be gained from superscalar processor implementations only if they support out-of-order and speculative issue and execution of instructions. Out-of-order execution hardware extracts additional performance from sequential code by decoupling instruction issuing and instruction execution within a processor; essentially, it enables early execution of independent instructions and deferred execution of data-dependent instructions. Speculative execution attempts to predict the outcome of conditional branches, and issues instructions from the predicted path before the branch condition has been resolved. Unfortunately, both of these techniques complicate the implementation of precise interrupts and result in storage conflicts (anti- and output dependences) arising from register reuse. Currently, these problems are addressed with complex hardware that keeps track of an in-order processor state, copies operands to reservations stations, allocates register storage dynamically to avoid storage conflicts, and nullifies the effects of instructions fetched down a mispredicted path. As an alternative, an intelligent register allocation scheme is suggested that avoids these hardware mechanisms without sacrificing precise interrupts or the high performance obtained through out-of-order and speculative execution.

The remainder of this chapter is structured as follows: Section 1.1 introduces definitions and concepts related to the various aspects of extended register allocation; Section 1.2 describes its application to out-of-order execution and precise interrupt support; Section 1.3 suggests a practical implementation of extended register allocation; while Section 1.4 extends its application to speculative execution.
1.1 Concepts and Definitions

In order to communicate effectively about extended register allocation and its application to out-of-order execution, speculative execution, and support for precise interrupts, the following terms are introduced and defined:

- sequential execution model
- out-of-order execution
- speculative execution
- state preservation
- exception condition
- precise vs. imprecise exceptions

1.1.1 Definitions

Sequential Execution Model. The sequential execution model inherent in most instruction set architectures requires that any implementation of that architecture maintain or provide a means for recovering architectural state as if the instructions in an instruction stream were being executed sequentially without any overlap.

Out-of-order execution. This term describes any implementation that deviates from the sequential execution model by allowing instructions within a sequential stream to execute in a dynamically determined order based on data and structural dependencies.

Speculative execution. This term describes any implementation that deviates from the sequential execution model by allowing instructions that reside beyond a conditional branch to begin execution before the branch is resolved.

State Preservation. This term refers to the process of maintaining or recovering the proper architectural state required by the sequential execution model when a control flow interruption such as an exception occurs.

Exception Condition. Exception conditions are defined as unusual or unexpected circumstances in the operation of a processor that usually demand some kind of remedial processing to either ensure the correctness of computation or to notify the user of an error condition. Exception conditions are
resolved by interrupting the execution of the thread of control that caused the exception at an instruction boundary, calling an exception handler, and possibly resuming execution at the same instruction boundary.

*Precise Exception.* This term is introduced to differentiate those exceptions that conform to the sequential execution model from those that don’t. In other words, precise exceptions are those that support the architectural illusion that instructions are executed sequentially, while imprecise exceptions are those that fail to do so.

### 1.1.2 Difficulties with the Sequential Execution Model

Supporting the sequential execution model in pipelined, multiple-issue implementations that support out-of-order and speculative execution is extremely difficult, as instructions that occur later in the instruction stream are allowed to complete before earlier ones, and may destroy architectural state required by the sequential execution model. This model is useful, and necessary in almost any computing system, because it enables a thread of control to be interrupted and later restarted at any instruction boundary.

The introduction of out-of-order completion of instructions in an instruction stream, which occurs in both single- and multiple-issue machines, causes great difficulties for the state preservation. If instructions are issued to different functional units with divergent result latencies, the potential exists for instructions to complete out of order. Once this occurs, instructions that occur later may destroy architectural state needed by earlier instructions that have yet to complete. If processing is not interrupted by an exception condition, this is not a problem. However, if it is, the question of attempting to reconstruct the state at the instruction boundary of the exception-causing instruction becomes extremely difficult. Hence, hardware mechanisms are introduced to preserve the architectural state or to journal changes to it in a way that enables backing out of those changes.

The same difficulty exists with speculative issuing of instructions; not only must their effects on the architectural state be nullified if execution is interrupted by an exception, but this must also be done if a branch is mispredicted.
1.2 Application to Out-of-order Execution

The fundamental difficulty of avoiding storage conflicts and implementing precise exceptions in pipelined and multiple-issue processors involves ensuring that a sequential execution model processor state exists or can be successfully reconstructed at any point in time. This problem is especially difficult in multiple-issue machines that allow out-of-order completion of instructions, and hence alter processor state in a manner that conflicts with the sequential execution model.

The problem is compounded by modern compilers that employ aggressive register allocation methods which seek to maximize the reuse of processor registers as storage locations. Since these register allocators operate according to the sequential execution model, they assume that any instruction that follows the last use of a register in a sequential instruction stream is free to overwrite the register with a new value. When out-of-order execution of instructions is allowed, however, the processor is no longer constrained by the sequential execution model, and the problem of preserving or reconstructing sequential processor state becomes extremely difficult.

Currently, this problem is addressed with complex hardware that keeps track of an in-order processor state by keeping multiple copies of register contents and allocating register storage dynamically to avoid conflicts. As an alternative, an intelligent register allocation scheme is suggested that avoids these hardware mechanisms without sacrificing support for precise exceptions. This approach involves modifying the register allocation phase of the compilation process to account for out-of-order execution of instructions.

1.2.1 Chaitin’s Heuristic for Register Allocation

In many modern compilers, register allocation is performed via Chaitin’s heuristic [2] on a program variable interference graph. In Chaitin’s interference graph, each node represents a program variable within the scope that register allocation is being performed. Edges between nodes in the graph simply mean that those two variables have overlapping liveness periods; that is, both variables exist simultaneously at some point during the execution of the program. The graph is then colored using Chaitin’s
heuristic such that each node in the graph receives a color distinct from all of its neighbors. Naturally, the heuristic attempts to minimize the number of colors used, but no guarantee of minimality is provided. The coloring obtained is then used to map each variable to a processor register.

The shortcoming of Chaitin’s scheme stems from the way the interference graph is constructed. To ensure correctness, the interference graph must represent overlapping liveness periods in the time domain. As long as the sequential execution model is adhered to, there exists a direct mapping from the instruction sequence domain to the time domain; that is, we know that an instruction occurring earlier in the instruction stream will also execute earlier in the time domain. This mapping no longer exists, however, once out-of-order and speculative execution are allowed. For this reason, the following extension to Chaitin’s scheme is suggested.

1.2.2 Extended Register Allocation Scheme

In the extended register allocation scheme, the standard interference graph is modified to account for potentially extended liveness periods for each variable. The liveness period of a variable is extended beyond its last use in the instruction stream to account for potential out-of-order execution of subsequent instruction within a specific processor implementation. Register allocation performed on this graph eliminates the occurrence of write-after-read (WAR) and write-after-write (WAW) dependences and guarantees precise interrupts without additional hardware for preserving an in-order processor state. Of course, hardware mechanisms will still be needed to maintain an in-order processor state for instructions with implicit side-effects, such as overwriting special purpose registers that cannot be renamed using this technique.

Define S as the strength of ordering of instruction execution within a processor (S=0 for a standard scalar processor that adheres to the sequential execution model). S is a static value defined for a given processor implementation and reflects the degree to which instructions in an execution stream can be reordered by the issuing hardware within the processor. The following affect it:

- size of the lookahead instruction window
- number of reservation stations (in a Tomasulo-type setup [3])
- result and issue latencies of the available functional units
- branch prediction strategy, or lack of such
A worst-case bound for $S$ can be defined as follows:

$$S = \left( \sum_{f \in F} (RS(f) \times \max \{RL(f)\}) - 1 - \min (RL) \right) \times \left( \frac{n}{\min \{AIL(n)\}} \right) :: n = \min \{IWS,1F1\}$$

where $F$ is the set of all functional units, $RS(f)$ the number of reservations stations for a functional unit $f$, $RL(f)$ the set of result latencies of functional unit $f$, $RL$ the set of result latencies, $AIL(n)$ the set of average issue latencies for any size n permutation of functional units, and $IWS$ is the size of the instruction window. The term on the left reflects the number of issuing cycles available before the last use instruction must execute, while the term on the right reflects the maximal issue rate for the processor.

The worst case is derived from a situation in which the instruction issue rate before the last use $u$ of a variable $x$ is minimal and the instruction issue rate after it is maximal. In other words, $u$ is at the end of a RAW dependence chain that contains an instruction of maximum result latency for each reservation station. Once all the instructions preceding $u$ have executed and $u$ can finally begin execution, $S$ subsequent and independent instructions with minimum result latency may already have executed (the issue rate for these instructions is limited by the instruction window size or the number of functional units, whichever is lesser). In this case, the worst-case value of $S$ will guarantee that none of the subsequent instructions will overwrite $x$ before its last use at $u$.

Clearly, the above global worst-case value is not feasible for augmenting the liveness periods of variables in the interference graph. For example, consider a processor with 4 functional units:

- one load/store unit with $\max \{RL\} = \min \{RL\} = 2$
- one FP divide unit with $\max \{RL\} = 10$ and $\min \{RL\} = 1$
- one FP multiply/add unit and one integer unit, both with $\max \{RL\} = 2$ and $\min \{RL\} = 1$
- an instruction window size of 2
- 2 reservation stations for each functional unit

The value of $S$ for such a processor would be 60. The number of interference edges introduced by such a large $S$ would increase register pressure beyond a reasonable amount, and the resultant register allocation would be very inefficient.
Fortunately, tighter bounds for $S$ can be established. In particular, $S$ can be defined for each data type used by the processor, as long as the register sets and operations for each data type are independent (e.g. separate FP and integer register sets). This will reduce the magnitude of $S$ by a considerable amount.

Once $S$ has been determined for a given processor, the interference graph for the program must be modified to account for the extended liveness periods of each variable. An edge must be added from each variable $x$ to the variables defined by the $S$ instructions following the last use $u$ of $x$, subject to the following considerations:

- If the variable defined is of a different data type and is stored in an independent register file, no edge needs to be added.
- If the processor stalls to resolve a branch instruction, the stall cycles multiplied by the maximal issue rate can be counted towards $S$ on both branch paths.
- If the processor predicts either branch path (taken or not taken), the branch delay cycles can be counted only towards the non-predicted path.
- If branch prediction is based on dynamic values (e.g. a branch target buffer), the branch delay cycles cannot be counted towards either path.

The usefulness of the proposed approach must be measured against the increased register pressure that it creates. Since many additional interferences are added to the interference graph, an increase in the number of colors needed to color the graph is to be expected. Consequently, either the compiler must spill more variables from registers to memory to decrease register pressure, or the processor must provide more architected registers. Considering the hardware eliminated by removing the need for complex mechanisms for keeping track of in-order processor state, some hardware resources should become available for this purpose.

Although increasing register file size may be a viable approach to reducing register pressure, it can be argued that as processors attempt to exploit instruction-level parallelism by adding more and more functional units and permitting a greater degree of out-of-order execution, the $S$ parameter will increase to the point where applying extended register allocation will become intractable. Perhaps a better alternative would be to limit $S$ somehow by constraining the degree to which instructions can execute out-of-order. The following section explores this idea further.
1.3 Practical Extended Register Allocation

Three of the most aggressive multiple-issue microprocessors introduced to date—the IBM RS/6000 [4, 5], the Intel i80960CA [6], and the DEC Alpha AXP [7]—have all abandoned the idea of implementing precise floating-point exceptions (precise in the traditional sense). Two of them, the IBM and Intel processors, implement sequential-execution modes that severely degrade floating point performance but enable precise exceptions to be implemented. All the processors, however, implement synchronization instructions that can be used to drain the floating point pipeline and guarantee that all potentially exception-causing instructions have completed successfully. The intended use of these instructions is to provide languages such as Ada a mechanism to prevent separate blocks of code that have divergent exception handling routines from overlapping during execution. We propose a method for combining these synchronization primitives and the extended register allocation described earlier in a practical implementation that would guarantee precise semantic interrupts without either the severe performance degradation caused by the sequential execution mode of these processors or the dramatic increase in register pressure mentioned earlier. A section in [7] alludes to an approach similar to what we are proposing; we are not aware of any implementation of such an approach, however.

The method we are proposing consists of limiting the magnitude of the $S$ parameter described in Section 1.2.2 by judiciously placing synchronization points in the instruction stream to impose restrictions on the degree to which instructions may execute out of order. For example, by placing synchronization instructions at fixed intervals in the instruction stream, we need only to extend the liveness periods of variables to the next synchronization instruction. Since the synchronization instruction guarantees that all instructions preceding it complete execution before any of the instructions following it can alter architectural state, we have restored the notion of a one-to-one mapping from the instruction stream domain to the time domain, albeit at a coarser level of precision. Hence, we only need to consider the effects of instructions overlapping within the bounds of synchronization, rather than within the bounds of the instruction issuing mechanisms of the processor implementation. Since the magnitude of $S$ is now directly controllable by the compiler, it can be adjusted to account for register pressure considerations. In other words, code segments with a great deal of register pressure (many simultaneously live variables) would have a shorter interval between synchronization points, while code with low register...
pressure would have longer intervals between synchronization instructions, or possibly even no synchronization instructions at all.

The other main advantage of this approach stems from the fact that the magnitude of the S parameter is no longer dependent on details of the processor implementation. Since the distribution of synchronization points is under direct control of the compiler, the compiler can make judicious decisions about trading off the high performance due to out-of-order execution brought on by less frequent synchronization points vs. the register pressure and possible register spill code that may result.

### 1.4 Application to Speculative Execution

Another interesting application of extended register allocation is speculative execution. Control dependences are a major performance bottleneck for superscalar processors [1, 8], since they can stall instruction issuing and cause processor resources to idle much of the time. Speculative execution beyond conditional branch instructions has been suggested as a way to keep these resources busy productively. Executing instructions speculatively, however, requires that the processor must be able to undo the results of a mispredicted branch, to prevent violating the sequential execution model. An extension of the register allocation scheme described above is suggested that eliminates the need for complex hardware to undo the speculatively executed instructions.

Consider modifying the interference graph in Chaitin’s heuristic such that variables that are live on either control flow path following a conditional branch instruction interfere with the liveness set at the branch instruction as well as with each other. This will prevent instructions that are executed speculatively on either path from destroying state information needed on the other in case a branch is mispredicted. As a result, none of the defining instructions need to be undone or predicated until the branch is resolved. The only instructions that still may destroy processor state are stores to memory, but since these are rarely in a critical path, delaying their execution until branch resolution will have little impact on performance.

In order to modify the interference graph, we need to determine the maximum number $M$ of instructions following a branch instruction that can be issued before the branch is resolved. This number is simply the maximum issue rate of the processor multiplied by the number of branch delay cycles. The
value of M is analogous to S defined above for out-of-order execution. However, since branch delays are typically on the order of 1-3 cycles, M will be much smaller than S. If both techniques are combined, however, M will be similar to S, due to the indeterminism introduced by out-of-order execution. In other words, a branch instruction may be the last in a dependence chain with multiple long-latency instructions. Hence, out-of-order execution would allow multiple subsequent instructions to issue and execute before the branch is resolved, and destination registers for these instructions must be allocated so that no processor state is destroyed.

Once M has been determined, the interference graph needs to be modified subject to the branch prediction strategy of the processor in question. If the processor predicts branches statically (i.e. a certain type of branch is always predicted as either taken or not taken), edges need to be added from each variable defined in the M instructions following the branch on the predicted path to the liveness set at the branch instruction. Doing so will guarantee that executing these instructions while the branch outcome is being determined will not destroy processor state needed if the branch was mispredicted. If the processor has dynamic branch prediction, edges must be added from the liveness set at the branch instruction to the variables defined on both branch paths. Additionally, if the processor has interleaved speculative execution, where instructions from both paths are executed until the branch is resolved, edges need to be added from the variables defined on one path to the variables defined on the other path. If this is the case, unless the processor has additional instruction issuing bandwidth for speculative execution, only M/2 instructions need to be considered on each path.
Instruction scheduling is a compilation technique that seeks to maximize functional unit utilization within a processor by scheduling long latency instructions early in the instruction stream and filling data-dependence-induced stall slots with independent instructions. This is done in order to enhance the instruction-level parallelism that is visible to the processor and increase the instructions-per-cycle or IPC count [9], and consequently improve program performance.

Most modern compilers implement at least some form of instruction scheduling, ranging from simple load and branch delay slot filling [10] all the way to extremely aggressive approaches like trace scheduling [11, 12] and percolation scheduling [13] which employ speculative code motion. This chapter presents a novel approach to performing aggressive instruction scheduling in the context of the superscalar IBM RS/6000 processor architecture [4]. Results are presented which indicate that speedups of up to 6% are achievable on the existing RS/6000 implementation. These agree with results presented by others [14]. Increasing the available machine parallelism by adding additional functional units while matching this increase by enhancing program parallelism through aggressively speculative code motion is shown to yield substantially higher performance gains. An exploration of this machine design space is presented, along with results that indicate that speedups on the order of 30%-40% are achievable with simple architectural extensions in conjunction with aggressive instruction scheduling.

The remainder of this chapter is structured as follows: Section 2.1 introduces important concepts and outlines previous work; Section 2.2 presents additional background information and a justification for our work; Section 2.3 presents the machine model used; Section 2.4 describes the operation of our scheduler; Section 2.5 presents preliminary results; and Section 2.6 discusses ongoing and future work.
2.1 Introduction

Instruction scheduling techniques can be subdivided into three main categories; those that are restricted to code motion within basic blocks, those that allow code motion between basic blocks only in the context of techniques such as loop unrolling [15, 12] or software pipelining [16, 17], and those that allow speculative code motion between basic blocks. Speculative code motion consists of moving instructions between basic blocks that are not control flow equivalent [18], i.e. basic blocks where the execution of one basic block does not necessarily imply the execution of the other. The control flow equivalence of basic blocks A and B means that any control flow path used to reach B must pass through A (A dominates B in the control flow graph), while any control flow path that passes through A must also reach B (B postdominates A). Avoiding speculative code motion simplifies the compiler’s task significantly, since performing it requires certain corrective actions to ensure that the semantics of the program are not violated.

Others [8, 20, 21] have argued that speculative code motion is necessary for extracting instruction-level parallelism from branch-intensive, non-scientific programs. While well-understood compilation techniques like loop unrolling and software pipelining are very effective at extracting parallelism from loop-intensive programs, they fall short with programs that are dominated by control flow. Numerous approaches have been proposed for implementing speculative code motion, both in hardware and in software [1, 22, 23, 24].

One of the most promising approaches combines hardware and software techniques for supporting speculative code motion by relegating code motion decisions to the compiler, and the elimination of unwanted side effects to the hardware. This approach is called boosting, and was introduced by Smith et al. in [25]. The authors present a convincing argument against dynamic instruction scheduling methods, pointing out their complexity, limited instruction selection scope, and inability to take advantage of control dependence information. In contrast, they point out a static instruction scheduler’s ability to exploit control dependence information and to select instructions from a much larger scope.

Boosting is an instruction scheduling paradigm that enhances available instruction-level parallelism by speculatively moving instructions up past conditional branches in the compiler and providing hard-
ware support to delay committing the results and side effects of the boosted instructions until the con-
ditional branches have been resolved. This is accomplished via tagging boosted instructions with their
control dependence information, and storing their results temporarily in shadow register files until the
program control flow resolves the branch in question. At this point, the results in the shadow register
file structure are either committed or discarded, depending on the outcome of the branch.

This chapter describes an enhanced compilation technique similar to boosting that reduces the need for
complex hardware support [25]. The technique described utilizes register dataflow information at basic
block boundaries to determine what, if anything, needs to be done in order to preserve correct program
semantics while allowing operations to be moved across basic block boundaries.

We have implemented a post-pass speculative instruction scheduler and trace-driven pipeline simulator
for the IBM RS/6000 architecture [4, 5] in order to evaluate the effectiveness of this approach. The
instruction scheduler takes as input a binary executable image, disassembles it, and performs control
and dataflow analysis [26] on it. This information, along with branch statistics generated by a profiled
version of the image, are used by the scheduler to create a new code schedule which exploits idle
machine resources by speculatively moving operations across basic block boundaries. The new code
schedule is then fed to a pipeline simulator for performance analysis. Both the scheduler and the simu-
lator can be configured to support various extensions to the RS/6000, and work is in progress for eval-
uating the performance attainable for a variety of such configurations.

2.2 Background and Justification

Smith et al. [27] identify two characteristics of speculative code motion—safety and legality—that
describe how correct program semantics may be violated by unwanted side effects. The cross-product
of the two produces four different types of code motion; safe and legal, safe but illegal, legal but
unsafe, and unsafe and illegal. These are summarized in Figure 1. The safety characteristic of code
motion describes its capacity for causing an exception at an incorrect point in the control flow, whereas
the legality of code motion refers to whether or not the code motion destroys program state that may be
needed if the branch was mispredicted.
To address these two potential violations of program semantics, Smith et al. propose a combination of compiler and hardware techniques. To maintain correctness in the case of unsafe code motion, they propose labeling boosted instructions as non-excepting, and delaying exception handling until the control dependences have been resolved. Similarly, to maintain correctness for illegal code motion, they propose that instructions that are labeled as boosted temporarily store their results into shadow register files, again until their control dependences have been resolved.

While we agree that hardware support is necessary for maintaining correctness in the case of unsafe code motion, we think that a case can be made against Smith et al.’s fairly elaborate scheme for handling illegal code motion. Rather than extending the hardware to contain multiple instances of the register file (one for each conditional branch that an instruction is allowed to move across) to prevent destructive side effects, or limiting performance by the less aggressive alternatives presented in [27], we hold that non-destructive storage allocation can be performed efficiently by the compiler. By building on the ideas presented in Chapter 1 in the context of dynamic instruction scheduling, we conclude that modifying the register allocation for instructions that have no implicit side effects is sufficient for guaranteeing semantic correctness for illegal code motion. Of course, without hardware support, we must forfeit unsafe code motion as well as motion of instructions with destructive implicit side effects. While the results of our approach may not be as impressive as those presented by Smith et al., our
approach is applied to a real machine and uses existing hardware. Also, our scheduler is compatible with existing code, since our approach does not require modifying the instruction formats to encode control dependence information within them.

2.3 Machine Model

The machine model used by the scheduler approximates the structure of the implementation of the IBM RS/6000 architecture presented in [4] and [5]. This implementation incorporates an instruction cache unit (ICU), which handles instruction fetching and dispatching, as well as execution of branch and condition code instructions; a fixed point unit (FXU), which executes fixed point and load/store instructions; a floating point unit (FPU), which executes floating point instructions; and disjoint condition code (CC), general purpose (GP), and floating point (FP) register files. The scheduler’s machine model can be configured to extend the existing implementation by including one or more FXUs, FPUs, and zero or more load/store units (MEMs), which are dedicated to executing load/store instructions. Figure 2 summarizes the configurable machine model.

The execution pipelines of each of the functional unit types are summarized in Figure 3. The ICU has two stages, Fetch and Disp/BRE. During the first stage, instructions are fetched from the instruction

FIGURE 2. Configurable Machine Model
cache. During the second, they are either dispatched to an appropriate functional unit, or executed in the ICU (if they are Branch or CC instructions). The throughput (or width, in terms of how many instructions are processed simultaneously) of these stages is determined by the throughput of the functional unit configuration specified by the user. For example, the existing implementation presented in [4, 5] is capable of executing one FXU, one FPU, one Branch, and one CC instruction concurrently. As a result, the Fetch and Disp stages are four instructions wide. If the user extends the machine model by specifying additional functional units, the scheduler assumes that the throughput of these stages will be scaled accordingly. Doing so will complicate the dependence-checking that must be performed in the Disp stage somewhat, but since all instructions will be dispatched in-order, the hardware will be significantly simpler than that required by a processor that supports out-of-order dispatch [1].

**FIGURE 3. Execution Pipelines with Forwarding Paths**

The two forwarding paths shown leading to the Disp stage are used to convey the results of condition-code generating instructions from the FPU and FXU pipelines to the dispatching logic. In the current implementation, there is an additional off-chip delay in these paths that increases the compare-branch delays to 4 and 7 cycles for the FXU and FPU, respectively.
The FXU and MEM functional units have the four-stage pipeline shown in Figure 3. There are forwarding paths from the Execute and Memory stages back to the Execute stage. Consequently, most instructions have a single-cycle result latency; the ones that don’t are either loads, which have a two-cycle result latency, or non-pipelined instructions like multiply and divide.

The FPU functional units have the six-stage pipeline shown in Figure 3. The Remap stage is used for dynamic register renaming for floating point load instructions [5]. The forwarding path from the Execute2 stage to Execute1 stage allows all pipelined instructions to have a 2-cycle result latency (the only non-pipelined instruction is FP divide). Additionally, the forwarding path from the Memory stage allows a FP load (executed by FXU or MEM) and a dependent FP use to be issued simultaneously.

2.4 Scheduler Operation

As mentioned earlier, the instruction scheduler takes as input a binary executable image, generated by the IBM XL compiler back end [28], disassembles it, and performs control and dataflow analysis on it. This information, along with branch and execution statistics generated by a profiled version of the image, are fed to the scheduling framework presented below. The disassembly, control flow analysis, and profiling are all performed by an enhanced version of the goblin system [29].

The scheduler is designed to take advantage of profiling information, not only for predicting the outcomes of conditional branches and boosting from likely successor basic blocks, but also for favoring those portions of the code executed most often by scheduling them first. This approach allows us to generate the best possible execution schedule for those portions that contribute the most to the dynamic cycle count of the program. We expect the importance of profile-guided scheduling to increase as we augment our scheduler to support live-range splitting [36] and spill-code insertion, since the scheduler will be able to include execution-frequency statistics into its spill code cost/priority functions [36].
2.4.1 Scheduling Framework

For reasons of computational efficiency, the scheduling framework divides the image into fragments. A fragment is defined as a sequence of basic blocks that has one or more external entry points (i.e. subroutine call destinations) followed by one or more exit points (subroutine return statements). Each fragment is processed separately through the following sequence of steps:

1. Compute the control flow and control dependence information for the fragment.
2. Compute register dataflow and def-use chains for the fragment.
3. Order the basic blocks in the fragment by their execution frequency.
4. Select the basic block with the highest execution frequency and add it to the scheduling scope as the host basic block.
5. Find the most likely successor basic block, and add it to the scheduling scope as a guest.
6. Apply Step 5 recursively, until the number of conditional branches crossed matches that specified by the user.
7. Construct a dependence graph (defined in Section 2.4.2) for the scheduling scope.
8. Schedule instructions in the scope, subject to the dependence graph, until all instructions in the host basic block have been scheduled. Allow upward motion of instructions from successor basic blocks to the host by performing the actions specified in Section 2.4.3 for preserving semantic correctness.
9. Repeat from Step 4 until all basic blocks in the fragment have been scheduled.

For our purposes, a scheduling scope is defined as a collection of basic blocks with one host basic block, and one or more guest basic blocks. The number of guest basic blocks is indirectly determined by the number of conditional branches the user specifies should be crossed. Instructions are scheduled primarily from the host, but whenever there is an idle functional unit and no available instruction in the host’s data ready set, the algorithm looks for schedulable instructions from guest basic blocks.

2.4.2 List Scheduling Algorithm

The list scheduling algorithm employed by our scheduler is fairly straightforward. As the host and guest basic blocks are added to the scheduling scope, the instructions in them are scanned for dependences, which are then added to the dependence graph. Our dependence graph contains three types of edges; RAW edges, which represent read-after-write, or true dependences between operations; WAW edges, which represent write-after-write, or output dependences; and MISC edges, which are introduced to the graph to impose miscellaneous ordering constraints. These are used to constrain the scheduler to keep all memory accesses strongly ordered, to prevent upward code motion of exception-
causing instructions, and to prevent downward code motion. Figure 5 shows an example *dependence graph*, one that corresponds to the scheduling example in Figure 4. Note that the RAW edges are weighted with the result latencies of the instructions in question. The weights on these edges are used to compute weights for each node in the graph, based on the longest path from the instruction to any other instruction in the same basic block.

The scheduler computes two *ready sets*, one for the host and one for the guests. The *ready set* is the set of instructions whose reverse RAW and MISC dependences have all been satisfied. Each *ready set* is then ordered by the following cost function:

\[
Cost = \text{weight} \times 10 + |\text{kill}| - |\text{gen}| - 2 \times |\text{ren}| - \text{CP} \times |\text{copy}|
\]

where *weight* is the original weight assigned earlier, *kill* is the set of last uses, *gen* is the set of register values generated, *ren* is the set of destination registers that must be renamed, *CP* is the user-specified cost of introducing a copy instruction, and *copy* is the set of copy instructions that must be generated. The *kill* and *gen* terms are a heuristic for decreasing register pressure, while the *ren* and *copy* terms are used to discourage excessive resource consumption.

Once the *ready sets* are available, the scheduler attempts to find instructions from the host *ready set* for each idle functional unit; if it fails to do so, it looks for instructions in the guest *ready set*. Once an instruction has been scheduled for each available functional unit (or, in the absence of available

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Scheduled Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>load r4= ...</td>
<td>load r4= ...</td>
</tr>
<tr>
<td>load r5= ...</td>
<td>load r5= ...</td>
</tr>
<tr>
<td>cmpi c0,r4,10</td>
<td>cmpi c0,r4,10</td>
</tr>
<tr>
<td>add r4=r4+r5</td>
<td>add r4=r4+r5</td>
</tr>
<tr>
<td>bc c0, A1</td>
<td>sub r3=r7-r4</td>
</tr>
<tr>
<td>st ... =r4</td>
<td>and r6=r3&amp;r5</td>
</tr>
<tr>
<td>A1: sub r3=r7-r4</td>
<td>bc c0, A1</td>
</tr>
<tr>
<td>and r4=r3&amp;r5</td>
<td>st ... =r4</td>
</tr>
<tr>
<td>st ... =r4</td>
<td>A1: st ... =r6</td>
</tr>
</tbody>
</table>

**FIGURE 4. Scheduling Example**
instructions, the functional unit has stalled), the scheduler increments its cycle count and proceeds to reconstruct the ready sets for scheduling the next cycle.

The final schedule in Figure 4 illustrates how the scheduler boosted two instructions from the A1 guest basic block to cover the latency between the cmpi and bc instructions; the second of these also had its destination renamed to r6 in order to prevent it from overwriting r4, which must be preserved in case the branch falls through.

2.4.3 Preserving Semantic Correctness

In order to preserve semantic correctness while moving instructions up from guest basic blocks to the host, we perform some combination of three simple transformations; register renaming, register copy creation, and instruction replication. These transformations are performed subject to dataflow, control flow, and control dependence information, and are employed only when required to preserve semantic correctness.
Register Renaming

Register renaming is the key to avoiding the multiple register files proposed by Smith et al. [27, 25]. To preserve program semantics while performing illegal code motion of the type illustrated in Figure 1(b), the result of the instruction in question must be stored in a location that is not live on the alternate control flow path. Smith et al. accomplish this by creating a duplicate shadow register file structure from which the value is copied once the branch condition is resolved. Instead, we rename the destination register as well as all of its dependent uses to an unused register.

The scheduling algorithm maintains a register scoreboard for determining which registers are in use, and uses a first-fit policy for selecting a replacement register for instructions that are boosted. The register scoreboard, which is the static scheduler’s counterpart to the scoreboards used in dynamic scheduling mechanisms [1], is also used to determine when renaming is necessary. It is initialized to mark those registers that are live on alternate exits from the scheduling scope as busy, so that any instructions that are moved beyond those exit points and overwrite the live storage locations are renamed. Also, as instructions are scheduled, the scoreboard is updated to reflect the number of outstanding uses of the current variable residing in each register, so that once all uses have been scheduled, the register becomes available for reuse.

Register Copy Creation

Register renaming is not always sufficient for maintaining correct program semantics. In cases where the boosted instruction is one of multiple reaching definitions of a variable, a mechanism is needed for choosing the appropriate definition based on the outcome of control flow dependences. An example of such a case is in Figure 6; if the definition of r1 in basic block (ii) is boosted into basic block (i), it must be renamed to prevent it from overwriting the previous value of r1, which must be preserved in case the left branch is taken. However, since multiple definitions of r1 reach the use in basic block (iii), we cannot rename that use. Hence, we must have a mechanism for choosing which reaching definition we want to use. The obvious solution is to insert a register copy in basic block (ii). This way, whichever branch is taken, we ensure that the correct value resides in r1 when it is used in basic block (iii).

While this may seem cumbersome at first, upon closer inspection we realize that by inserting the register copy in basic block (ii), we are only doing in software what the original boosting algorithm does in
hardware [25]. In a sense, we are making the classic RISC trade-off of code size vs. hardware complexity. In Smith et al.’s approach, the register copy that we specify explicitly is implicitly encoded within the conditional branch instruction; we pay the price of adding an additional instruction to the stream, while he pays the price of additional hardware for brute-forcing the copy from a duplicate register file as the branch is executed.

We expect that some form of hardware support for executing these register copies may be desirable in a very aggressive future implementation of the architecture. One means for support would be to add instruction opcodes that encode multiple register copy operations, as well as functional units to support these instructions. The primary resource required by these functional units would be additional register file ports.

**Instruction Replication**

To support the general case of upward speculative code motion, we must also consider the case in which instructions are moved across an entry point, or control flow join. In this case, a mechanism is needed for ensuring that the boosted instructions are executed irrespective of which control flow path is taken to reach the *guest* basic block in which they originally resided. The simple solution is to replicate the boosted instruction at the end of each basic block that precedes the *guest* in the control flow...
graph. In cases where the guest is not an immediate successor of the host, however, the solution is not quite so simple. For example, in the control flow graph shown in Figure 7, an instruction boosted from basic block (iv) to basic block (i) must be replicated in basic block (v), but not in (ii) or (iii).

---

**FIGURE 7. Example of Instruction Replication**

We compute a *replicate* set for each guest basic block in the scope by searching all paths from the guest to the host, and marking each node that is on a path. We then check the predecessor set of each marked basic block for unmarked basic blocks; these are added to the *replicate* set of the guest. Then, as instructions are boosted into the host, they are also appended to each basic block in the *replicate* set.

### 2.5 Results

The results presented here are from a small set of integer benchmarks; our purpose was to explore the operation of the scheduler in the context of small programs with easily identifiable characteristics in order to facilitate a better understanding of why the scheduler performed the way it did. Also, we concentrated our initial efforts on ways to improve the integer performance of the RS/6000 architecture, since the floating point performance is already quite impressive [29, 30]. Work is currently in progress for collecting data from a larger set of standard benchmarks with longer run-times, including several from the SPEC benchmark suite.

The data we have collected indicate that, depending upon the characteristics of the program, speedups from 0% to 6.4% are achievable on the existing machine model, while speedups of up to 54% are possible with an aggressive machine model that contains 3 fixed point units (FXUs).
2.5.1 Existing Machine Model

The speedups for the existing machine model are summarized in Table 1. We are encouraged by the significant performance gains achieved on the grope and quicksort benchmarks. We attribute these gains to the scheduler’s ability to reduce the stalls caused by data dependences (by moving instructions up so they execute earlier) as well as control dependences (by covering the compare-branch latency of the machine). These reductions are clearly visible in the graph in Figure 8, which shows the proportion of stalls caused by each type of hazard (structural, data, and control), both before and after scheduling.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Cycles Unscheduled</th>
<th>Cycles Scheduled</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>eightq</td>
<td>22,088</td>
<td>21,874</td>
<td>0.98%</td>
</tr>
<tr>
<td>grope</td>
<td>880,094</td>
<td>826,870</td>
<td>6.44%</td>
</tr>
<tr>
<td>quicksort</td>
<td>4,252</td>
<td>4,006</td>
<td>6.14%</td>
</tr>
<tr>
<td>inmult</td>
<td>507,172</td>
<td>507,067</td>
<td>0.02%</td>
</tr>
<tr>
<td>shellsort</td>
<td>5,136</td>
<td>5,136</td>
<td>0.00%</td>
</tr>
<tr>
<td>Harmonic Mean</td>
<td></td>
<td></td>
<td>2.63%</td>
</tr>
<tr>
<td>HM of best 3</td>
<td></td>
<td></td>
<td>4.46%</td>
</tr>
</tbody>
</table>

Conversely, we attribute the poor performance of the scheduler on the inmult benchmark to the fact that this program is dominated by data-dependence induced stalls, and while we manage to reduce these slightly, the improvement is offset by structural hazards introduced by the instruction replication and copy creation. Poor performance on the shellsort benchmark, on the other hand, can be attributed to its extremely small average dynamic basic block size of 2.58 instructions. When the basic blocks are this small, it is very difficult for the scheduler to find independent instructions to move up.

The other conclusion we can draw from Figure 8 is that the scheduler is doing fairly well at abiding by its stated objective of only exploiting idle resources for speculatively scheduled instructions. On all of the benchmarks, the number of structural hazards induced by the scheduler does increase, but only by an average of 1.5%. This indicates that the machine model used by the scheduler is accurate enough to avoid inflating the demand for resources beyond those available.
2.5.2 Augmented Machine Models

We also collected performance data for three machine models, called M1, M2, and M3, that are increasingly aggressive in terms of available machine parallelism. The first, M1, augments the existing machine with a MEM functional unit that is dedicated to executing all load/store instructions, and increases the dispatch bandwidth of the ICU to five instructions per cycle. The second machine model, M2, replaces the MEM unit of M1 with a general-purpose FXU. The main cost of doing so involves adding an additional port to the data cache for memory accesses. The most aggressive machine model, M3, incorporates an additional FXU, with a total of 3 ports to the data cache, and a dispatch bandwidth of six instructions per cycle.

The speedups for the augmented machine models are shown in Table 2. Once again, depending on the characteristics of the benchmark, the scheduler produces speedups anywhere from 0% to 10% over the original code. It is encouraging to see that speedups as dramatic as 54% over the existing machine are possible with relatively simple architectural extensions and no hardware support for out-of-order exe-
Future Work

There are four main areas in which we see opportunities for future work. The first is in hardware extensions to our current machine model for supporting unsafe code motion. The second and third are in augmenting our scheduler to exploit unsafe code motion, as well as to allow weak ordering of memory references by performing memory reference disambiguation [31, 32, 33, 34]. Last of all, we feel there is a great deal of work to be done in the area of integrating a better mechanism for register reallocation and spill code insertion into our scheduling framework.

Numerous results presented by others indicate that significant performance gains can be obtained through unsafe speculative code motion [8]. We hope to explore this area further in the context of our work by adding support for this to our machine model. This would involve two major changes: additional opcodes for identifying non-excepting versions of exception-causing instructions, as well as some mechanism for supporting delayed exception handling for these instructions. A comprehensive discussion of this topic is presented in [35], and we would likely model our extensions after the suggestions made there.

\[ \begin{array}{|c|c|c|c|c|c|c|}
\hline
\text{Benchmark} & \text{M1 Unsched.} & \text{M1 Sched.} & \text{M2 Unsched.} & \text{M2 Sched.} & \text{M3 Unsched.} & \text{M3 Sched.} \\
\hline
\text{eightq} & 9.26\% & 11.21\% & 13.21\% & 13.81\% & 26.03\% & 28.65\% \\
\text{gropo} & 17.44\% & 24.30\% & 19.34\% & 26.45\% & 43.66\% & 54.12\% \\
\text{quicksort} & 21.31\% & 25.91\% & 23.07\% & 28.15\% & 27.53\% & 37.38\% \\
\text{inmult} & 11.41\% & 11.66\% & 11.94\% & 12.23\% & 26.87\% & 27.71\% \\
\text{shellsort} & 1.54\% & 1.54\% & 11.75\% & 11.73\% & 15.08\% & 15.08\% \\
\hline
\text{HM} & 11.77\% & 14.38\% & 15.69\% & 18.04\% & 27.20\% & 31.37\% \\
\text{Gain} & 2.61\% & 2.35\% & 2.35\% & 2.35\% & 2.35\% & 2.35\% \\
\text{Best 3} & 16.58\% & 20.44\% & 18.40\% & 22.46\% & 32.25\% & 39.27\% \\
\text{Gain} & 3.86\% & 4.06\% & 4.06\% & 4.06\% & 4.06\% & 4.06\% \\
\hline
\end{array} \]

2.6 Future Work

Numerous results presented by others indicate that significant performance gains can be obtained through unsafe speculative code motion [8]. We hope to explore this area further in the context of our work by adding support for this to our machine model. This would involve two major changes: additional opcodes for identifying non-excepting versions of exception-causing instructions, as well as some mechanism for supporting delayed exception handling for these instructions. A comprehensive discussion of this topic is presented in [35], and we would likely model our extensions after the suggestions made there.
Much excellent work has been done in the area of memory reference disambiguation [31, 32, 33, 34], and we do not foresee any overwhelming difficulties in integrating this technique into our framework. Likewise, adding the capability for boosting potentially exception-causing instructions should be a fairly simple task. We expect the importance of this extension to grow in proportion to the degree of machine parallelism available.

Integrating register allocation and instruction scheduling is a difficult problem that is further compounded by speculative code motion. Chapter 3 provides a closer look at this topic.
Register allocation and instruction scheduling are two distinct compilation techniques used by almost all modern compilers to improve code execution time. The former accomplishes this goal by seeking to minimize data memory traffic through exploitation of the large register files available on modern processors, while the latter seeks to maximize functional unit utilization by scheduling long latency instructions early and filling data-dependence-induced stall slots with independent instructions. Both techniques are fairly well understood, and have been explored extensively in the literature [2, 10, 11, 12, 35, 37, 36, 39]. What is not well understood is their effect on each other, and the seemingly circular dependence relationship between them. This chapter seeks to explore that effect further, and makes an attempt to eliminate the illusion of a circular dependence through a better understanding of how the two techniques interact.

The remainder of this chapter is structured as follows: Section 3.1 describes existing approaches to both register allocation and instruction scheduling; Section 3.2 defines and extends the notion of an interference graph; and Section 3.3 suggests a coagulation-based [40, 41] approach for integrating register allocation and instruction scheduling.

3.1 Existing Approaches

3.1.1 Instruction Scheduling

Instruction scheduling is usually accomplished via some variant of list scheduling. List scheduling employs a precedence graph, in which nodes represent instructions to be scheduled, and directed edges represent dependences between instructions. The edges are weighted according to the latencies of the dependences, and scheduling proceeds by choosing instructions from the ready set, favoring those on
Existing Approaches

the longest paths in the graph. The ready set consists of instructions whose reverse dependences (incoming edges) have all been satisfied. Please refer to Section 2.4.2 for an example of a list scheduling algorithm.

Instruction scheduling need not precede register allocation, but typically does, since it doesn’t require a pre-existing register assignment for program variables, but can use the single-assignment virtual register assignments specified by most intermediate representations. Nevertheless, it should take into consideration the results of register allocation, since the final instruction schedule has a severe impact on the efficacy of the register allocator [42]. On the other hand, performing register allocation first unduly constrains the scheduling process by introducing false dependences induced by aggressive register reuse (this problem is explored extensively in Chapter in the context of dynamic scheduling, i.e. out-of-order execution, but applies equally well to static scheduling).

Intuitively, it is clear that code motion that increases the distance between definitions of variables and their uses to hide pipeline latencies will increase the demand for register storage; this is borne out by the results of Section 1.2.2, where dynamic scheduling techniques are shown to dramatically increase this demand. Other results indicate that this is also true for static instruction scheduling; if the scheduler has no concept of register availability, instruction schedules that place unrealistic demands on register resources may result. Hence, most schedulers incorporate heuristics for estimating register availability, and seek to create schedules with reasonable resource demands [28, 36, 42]. The difficulty of implementing effective heuristics for estimating register demand is obvious: since the instruction schedule is still in a state of flux, register allocation cannot be performed, and the demand for register storage remains unknown.

3.1.2 Register Allocation

Register allocation, formalized by Chaitin in [2, 37], is dependent on instruction scheduling in that it requires a pre-existing instruction schedule that is used to determine storage conflicts between program variables. A widely used paradigm for register allocation is that of graph coloring (see Section 1.2.1 on page 4), where an interference graph that consists of edges representing storage conflicts and nodes representing program variables is colored such that each node in the graph receives a color distinct from all of its neighbors. The coloring obtained is then used to map each variable to an architected reg-
The Interference Graph

3.2 The Interference Graph

3.2.1 Definition

We introduce the following definition—which largely agrees with the one given by Chaitin in [2]—for the interference graph:

An interference graph I consists of a set of nodes N and a set of edges E. Each single-assignment variable in a program has a corresponding node in the set N. An edge is added to the set E between any two nodes that represent variables that are simultaneously live during execution of the program (i.e., that have a storage conflict in the sense that they cannot be stored in the same location during some point in the execution of the program).

If we expect to use Chaitin’s interference graph to model storage conflicts between program variables, we must have a better understanding of how factors such as the machine model, the sequential execution model, and machine implementation as well as compilation techniques like instruction scheduling impact that model. Chaitin introduces edges to the graph to represent overlapping variable liveness periods. The liveness periods are determined by inspecting the sequential code. In our case, however, the sequential code may not exist yet in its final form. Also, as explored in Chapter 1, even the sequen-
tial code itself may not be sufficient for determining the actual overlapping liveness periods. Hence, we must develop a better understanding of what causes the interferences, which interferences are essential, and what we can do to create and eliminate the ones that aren’t.

3.2.2 Causes of Variable Interference

Variable interferences can be broken down into two main categories: the essential ones which form a lower bound on the number of edges in the interference graph, and the non-essential ones, which are caused by factors such as the machine model, the sequential execution model, and dynamic and static scheduling mechanisms.

Essential Interferences

Essential interferences are intrinsic to the semantics of the program, and are imposed by dataflow constraints. Figure 9 below illustrates dataflow-induced interferences. These essential interferences are defined as follows:

An essential interference exists between a variable v and any variables u_i that are used or defined along any path in the dataflow graph from a definition of v to a use of v.

In Figure 9, variable a, defined by operation (i) in the dataflow graph, has essential interferences with variables b and d, because d is used and b is defined along a path in the dataflow graph from operation (i), where a is defined, to operation (iv), where a is used. It is clear that irrespective of the machine model employed or the order in which the instructions are scheduled, the liveness period of variable a must overlap with the liveness periods of variables b and d. Hence, the two solid interference edges shown in the graph in Figure 9 are termed essential.

Non-essential Interferences

Non-essential interferences differ from essential ones in that they are artifacts of the machine model used, the sequential execution model, and/or the scheduling decisions made at compile-time or runtime.

The machine model impacts the occurrence of variable interferences by specifying the number and type of variables that can be defined or used by an atomic operation in that machine model. For example, an n-wide VLIW [11] machine with functional units that each accept two source operands and one
The Interference Graph

result is capable of defining $n$ variables and using $2n$ variables within each atomic VLIW instruction. Since the execution of each VLIW instruction is deterministic, we know exactly when (in the time domain), a variable is defined, and when it is last used.

In contrast, a sequential instruction stream architecture that supports the *sequential execution model* (refer to Section 1.1 for a definition) may not guarantee exact time domain knowledge of variable definition and use, due to potential out-of-order execution of instructions. Also, the fact that a typical instruction can only contain a single operation from the dataflow graph causes additional interferences to occur. An example of this can be seen in Figure 9, where the interference between variable $b$ and variable $e$ is induced purely by the scheduling decision that places operation (iii) before operation (iv) in the instruction stream. Clearly, these two operations could be scheduled within the same VLIW instruction, and the interference would be avoided. Although a superscalar implementation could also detect that these operations are independent, and could execute them in parallel, the code produced by the compiler must nevertheless adhere to the *sequential execution model*, and must allocate separate registers for the two variables.

In the example in Figure 9, we also see the impact that static scheduling decisions make on the occurrences of interference edges. Had the scheduler decided to schedule operation (iv) before operation
(iii), there would not be an interference between variables b and e. There would, however, be an interference between variables b and c, since the definition of c would now precede the last use of b.

### 3.3 Integrating Register Allocation and Scheduling

Bradlee et al. [42] provide a good introduction to the problem of integrating register allocation and instruction scheduling. They present results comparing three approaches to implementing the two techniques: the first separates the two completely; the second employs heuristics to measure register demand and limits the scheduler correspondingly; and the third claims to integrate the two by employing a prescheduler that actually computes cost estimates for limiting register availability and uses these to guide the register allocator. Their results indicate that the computational cost of the third approach is not worth the slight improvement in code execution time, and they conclude that a heuristic-based approach similar to their second one usually suffices. However, their third approach does not truly integrate the two phases, but really only adds a more sophisticated cost function to the register allocator, allows it to run first, and then schedules subject to the results of the allocator. It does not truly integrate the two tasks and still suffers from a fundamental lack of understanding of their interaction. In addition, they do not address the additional difficulties introduced by scheduling techniques that incorporate speculative code motion (please refer to Section 2.1 for a definition).

In the following section, we introduce a different approach to the problem of integrating register allocation and instruction scheduling, using a profile-guided approach called coagulation.

### 3.3.1 Coagulation

Karr introduces the concept of code generation by coagulation [41]. An implementation of a coagulating code generator is presented in [40]. In this approach, profile data from previous execution is used to guide an integrated code generation and register allocation process to favor those portions executed most often. The program is formed through a process of coagulation, as the portions of generated code grow and are merged with their neighbors. Coagulation uses the following framework:

1. Build a control flow graph for the program.
2. Use an execution profile to label each edge in the flow graph with its expected execution frequency. Mark each edge as uncompiled.
3. Generate code and allocate registers for each basic block (considered in isolation) at minimal cost. Each node now presents boundary conditions concerning the location and type of data it uses and supplies.

4. Select an uncompiled edge with the highest expected frequency. If the boundary conditions on the arc’s entry and exit nodes do not agree, make minimal cost repairs (such as inserting copy instructions or revising storage allocation) to bring them into agreement. Mark the edge compiled and merge the compiled regions connected by the arc.

5. Repeat step 4 until all edges are compiled.

This approach yields significant performance improvements for CISC architectures. With the advent of orthogonal RISC instruction sets, however, the importance of code generation per se (i.e. instruction selection) has been diminished. In contrast, the importance of instruction scheduling has increased, as pipeline latencies must now be covered by reordering instructions in the original stream. Hence, we propose applying a modified coagulation framework for integrating register allocation and instruction scheduling.

3.3.2 Modified Coagulation Framework

We suggest modifying the framework presented above in a way that will exploit our better understanding of variable interference and replace the task of code generation with the task of instruction scheduling. This framework will employ the concept of an *incremental interference graph*; one that is initialized to contain the essential interferences that are determined by the program’s dataflow, and then augmented with non-essential interferences as scheduling and spilling decisions that cause them are made. The framework we suggest is the following:

1. Build a control flow graph for the program.

2. Initialize the *incremental interference graph* to include the essential interferences, derived from global dataflow information.

3. Use an execution profile to label each edge and each node in the flow graph with its expected execution frequency. Mark each edge as uncompiled.

4. Schedule instructions and allocate registers for each basic block at minimal cost, but subject to the constraints of the *incremental interference graph*. Do this in order of execution frequency, and update the incremental interference graph according to the scheduling and register allocation decisions made. Each node now presents boundary conditions concerning the location and type of data it uses and supplies, as well as resource utilization information that reflects the quality of the code schedule generated.

5. Select an uncompiled edge with the highest expected frequency. If the boundary conditions on the arc’s entry and exit nodes do not agree, make minimal cost repairs to bring them into agreement (this may include insert-
Integrating Register Allocation and Scheduling

...ing spill code to split liveness ranges of variables, inserting copy instructions, and possibly revising register allocation). Also, assess the quality of the code schedules in the two nodes connected by the arc, and proceed to move instructions speculatively from one to the other in order to improve the schedules of both. Mark the edge compiled, merge the compiled regions connected by the arc, and update the incremental interference graph appropriately.

6. Repeat step 5 until all edges are compiled.

The framework presented above is ambiguous in many respects; this is intentional, as there are numerous open research areas involved in the details of implementing it.

3.3.3 Advantages of Integration

There are numerous advantages to the integrated framework presented in the previous section. First of all, it integrates the tasks of instruction scheduling and register allocation without an unreasonable amount of overhead. It also avoids ad hoc heuristics for estimating register demand or the impact of scheduling or spilling decisions on register allocation, and instead models variable interference in a unified manner by constantly refining and assigning colors to the incremental interference graph. It avoids many boundary condition conflicts by allocating registers subject to the same incremental interference graph. It is also able to make intelligent choices about minimal cost spill code insertion and liveness range splitting as it compiles the edges in the control flow graph. Finally, it favors the most often executed portions of the program by scheduling and register allocating them first, and relegating spill code and other repairs to less important portions. All of these advantages are expected to result in significant performance improvements over more traditional approaches to instruction scheduling and register allocation.
List of References


List of References


List of References


