Efficient Superscalar Performance Through Boosting

Michael D. Smith, Mark Horowitz, Monica S. Lam
Computer Systems Laboratory
Stanford University

Abstract

The foremost goal of superscalar processor design is to increase performance through the exploitation of instruction-level parallelism (ILP). Previous studies have shown that speculative execution is required for high instruction per cycle (IPC) rates in non-numerical applications. The general trend has been toward supporting speculative execution in complicated, dynamically-scheduled processors. Performance, though, is more than just a high IPC rate; it also depends upon instruction count and cycle time. Boosting is an architectural technique that supports general speculative execution in simpler, statically-scheduled processors. Boosting labels speculative instructions with their control dependence information. This labelling eliminates control dependence constraints on instruction scheduling while still providing full dependence information to the hardware. We have incorporated boosting into a trace-based, global scheduling algorithm that exploits ILP without adversely affecting the instruction count of a program. We use this algorithm and estimates of the boosting hardware involved to evaluate how much speculative execution support is really necessary to achieve good performance. We find that a statically-scheduled superscalar processor using a minimal implementation of boosting can easily reach the performance of a much more complex dynamically-scheduled superscalar processor.

1 Introduction

The RISC revolution has driven processor design to the point where scalar machines are able to maintain an execution rate of nearly one instruction per cycle on non-numerical code. For computer architects, the next step is obvious: drive the IPC above one by executing multiple instructions per cycle. Multiple instruction execution machines exploit instruction-level parallelism through superscalar and/or superpipelined techniques. And as Jouppi and Wall [16] point out, these techniques are equivalent in their ability to exploit ILP. In non-numerical applications, the amount of ILP is limited. *Limit studies*, studies which try to bound the amount of exploitable ILP in applications, show that superscalar processors must look beyond branch boundaries to exploit the available ILP in non-numerical applications [22][29]. These studies show that good performance requires both a good instruction schedule and *speculative execution*, the execution of instructions before it is known for certain whether those instructions will be executed. What is not known is how to best schedule instructions for a superscalar machine that supports aggressive speculative execution and how

much speculative execution support is necessary to achieve good performance.

A growing perception is that dynamically-scheduled superscalar processors are the only effective way to couple instruction scheduling and speculative execution [19]. This perception seems to be supported in the commercial world as superscalar implementations move from dynamic dependence checking (e.g. the Sun SuperSPARC [5]) toward more complex dynamic scheduling techniques with support for speculative execution (e.g. the Motorola 88110 [5]). Yet, this hardware-intensive approach has a fundamental problem: these machines analyze only a small window of instructions at a time and use simplistic heuristics for choosing among the available instructions. Thus they are not guaranteed to generate a good instruction schedule.

On the other hand, many compiler algorithms exist for the effective scheduling of instructions across branch boundaries (e.g. [8][10][20]). These techniques, which are referred to as global scheduling techniques, have advantages over run-time instruction scheduling because they are able to analyze a much larger portion of the program at any time, and they can use sophisticated heuristics to choose among the available instructions. These advantages allow the compiler to optimize the schedule for the critical paths of the program. While these compiler-based approaches have the benefit of much simpler issue hardware, they have been limited in their ability to use speculative execution. To augment these global scheduling algorithms, a number of researchers have proposed architectural techniques (e.g. guarding [14] and non-excepting instructions [6][7]) which extend, but still limit, the compiler's ability to schedule instructions for speculative execution.

Recently we proposed a general architectural mechanism called *boosting* that provides the compiler with an unconstrained model of speculative execution [23]. That paper discusses the ideas that lead to the concept of boosting, and it contains a preliminary experiment to justify further research. Since then, we have constructed a complete compiler system and a working hardware model to better understand the capabilities and costs of boosting. Section 2 reviews speculative execution and the concept of boosting. Section 3 describes the global scheduling algorithm used in our compiler system. Boosting is supported by mechanisms in the hardware, and these mechanisms can be complex in the general case. By varying the hardware constraints on the instruction scheduler, we can answer the question of how much speculative execution support is really necessary for good performance. Section 4 discusses the hardware and scheduling implications that result from different boosting constraints. Performance comparisons show that good performance is achievable with only a minimal amount of hardware support for speculative execution and no hardware support for instruction scheduling.

2 Speculative Execution

Speculative execution is necessary because a conditional branch imposes a control dependence upon the instructions that occur after the branch. For example, the instructions in the THEN and ELSE portions of an IF statement are control dependent upon the IF condition because we cannot determine whether the THEN or the ELSE instructions should be executed until the condition in the IF is executed. Speculative execution allows for the THEN instructions or the ELSE instructions (or both) to be moved above and executed before the execution of the condition.

The movement of an instruction above its control dependent branch results in one of four types of speculative execution. The next subsection enumerates these different cases, and it briefly explains what mechanisms are necessary to support the movement of any instruction above its control dependent branch. Using this description, the second subsection reviews two of the more popular architectural approaches to overcome control dependence constraints. The last subsection explains how boosting is different from these previous approaches, and how boosting is able to support any speculative movement.

2.1 Achieving Safe Speculative Execution

Speculative movement, the straightforward movement of an instruction from below to above its control dependent branch, does not always maintain program semantics. Even if we assume that the movement preserves operand availability of the moving instruction, the speculative execution of this operation can still violate program semantics in two ways. The combination of these two possible violations results in the four types of speculative execution which are illustrated in Figure 1.
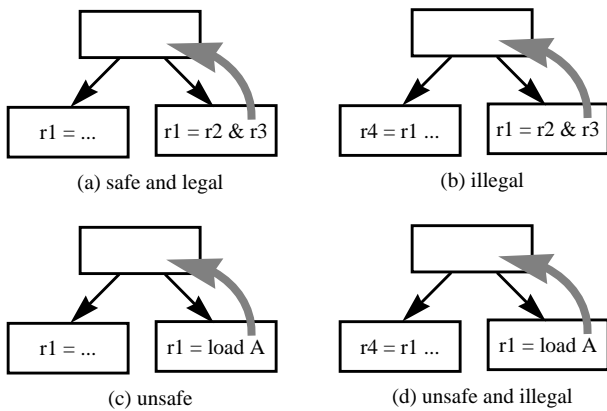

(a) safe and legal   (b) illegal

(c) unsafe   (d) unsafe and illegal

Figure 1: Types of speculative execution.

For a speculative operation, a branch can be either *correctly predicted* or *incorrectly predicted*. A branch is said to be correctly predicted for a speculative operation if the block from which the speculative operation was moved is executed after the branch is executed; otherwise, the branch is said to be incorrectly predicted. A speculative operation is said to be *illegal* if that operation overwrites a location whose previous value is needed by some instruction when the branch is incorrectly predicted; this speculative movement can be thought of as breaking an existing true data dependence constraint along the incorrectly-predicted path of the branch. Figure 1b is an example of an illegal speculative operation. A speculative operation is said to be *unsafe* if that operation can cause an exception to occur; this exception should only occur if the branch is correctly predicted. Figure 1c is an example of an unsafe

speculative operation since the load operation can cause an addressing exception. A speculative operation can obviously be both unsafe and illegal as in Figure 1d.

To preserve program semantics, a speculative movement should only result in speculative execution that is *safe and legal*. This requirement constrains the code motions available to static global scheduling techniques. A compiler may overcome some speculative movements that are illegal by renaming the destination register of a speculative operation so that it does not conflict with the set of registers that are needed (i.e. the set of registers that are *live*) on the incorrectly-predicted path of the branch. This renaming may require extra instructions later to select between multiple reaching values. Register renaming does not overcome speculative movements that are illegal due to a dependence through memory. Furthermore, a compiler can never transform an unsafe speculative movement into safe speculative execution, and thus a compiler alone cannot support the general movement of instructions above their control dependent branch.

There are numerous hardware techniques that allow dynamic schedulers to safely move any instruction above its control dependent branch [15][21]. The basis of all these techniques is the inclusion of extra buffering in the hardware which holds the effects of the speculative operations[1]. The *sequential state* of the machine is defined as that machine state that is not a result of any speculative operation, and conversely, the *speculative state* of the machine is defined as the machine state that is a result of speculative operations. The extra buffering separates the sequential and speculative states, and it postpones the speculative side effects (e.g. speculative exceptions) until the branch(es) is resolved. If all branches that a speculative instruction depends on are correctly predicted, hardware mechanisms must correctly update the sequential state of the machine with the speculative effects of that instruction. We refer to the updating of the sequential state as a *commit* of the speculative effects. If any dependent branch for a speculative instruction is incorrectly predicted, the machine simply discards the speculative state and side effects for that operation. We refer to the action of throwing away the speculative effects as a *squash* or *nullify* operation.

2.2 Existing Architectural Approaches

To extract the most ILP from a program, we need to incorporate the sophisticated scheduling ability of the pure compiler approaches with the unconstrained speculative execution of the pure hardware approaches. To accomplish this goal, we require an architectural technique that removes the control dependence constraints on static instruction scheduling. Guarded instruction architectures [14] and non-excepting instruction architectures [6][7] are the most widely accepted of these architectural techniques.

Guarded instruction architectures predicate a control dependent operation with its dependent branch condition. If the predicate evaluates to true, the effect of the operation is committed; if the predicate evaluates to false, the effect is squashed. The guarding predicates can be quite complex and can encode the control dependence information for multiple branches. Guarding also allows for the possible elimination of branch instructions. Another advantage of this technique is that the required speculative state is very small, and is held in the pipeline bypass registers that are already in the machine. The problem with guarding, however, is that the schedul-

---

1. In some of the proposals, the hardware buffering actually backs up the state that was displaced by the speculative operations. These schemes must ensure that the correct state is rebuilt on an incorrect prediction. These schemes do not postpone exception processing, and thus are not discussed further.

ing of guarded operations is constrained by the availability of the dependent branch condition.

Non-excepting instruction architectures rely on hardware mechanisms to handle unsafe speculative movements and on software renaming to handle illegal speculative movements. Non-excepting instruction architectures label unsafe speculative movements as non-excepting instructions. The semantics of a non-excepting instruction is that this instruction never signals an exception. If it causes an exception, it simply generates a polluted result. Eventually, some later (regular) instruction may try to use this polluted value, and it is at this time that the exception is signalled. In this way, non-excepting instruction architectures can detect exceptions on speculative operations. These polluted values are often implemented by building a tagged-data architecture. By carrying the address of the "excepting" non-excepting operation in the data field of the polluted operand, these architectures can indicate which instruction originally caused the exception.

A complexity inherent to any architecture that postpones the signalling of an exception involves the restart of the program after the handling of the postponed exception. The exception handler must be able to determine which instructions between the excepting point and the signalling point need to be re-executed (i.e. which instructions are dependent upon the original excepting operation[2]). Also, the compiler or hardware must guarantee that all of the operands for these re-executed instructions are still available. In summary, non-excepting architectures rely on software renaming to overcome some illegal speculative movements and on hardware mechanisms to overcome unsafe speculative movements. For unsafe speculative movements, the hardware completely removes the control dependence constraints on static scheduling, but the movement of the operations is still limited by the capability of the exception handling mechanism.

2.3 Boosting

Boosting is an architectural mechanism that provides the compiler with an unconstrained model of speculative execution. Boosting an operation removes all the control dependence constraints that inhibit the movement of that operation. In contrast to a non-excepting architecture, a boosting architecture includes hardware buffering to convert both unsafe and illegal speculative movements into safe and legal speculative execution. Furthermore, boosting cleanly handles all aspects of exception handling by providing a precise exception mechanism for all operations.

Only the hardware can efficiently buffer all effects of a speculative operation (including those side effects that would cause unsafe speculative execution), and only the hardwares can efficiently monitor the dynamic state of the conditional branch. Boosting requires that the hardware provide these features so the compiler can rely on the hardware to ensure that the semantics of the program is not violated during speculative execution. In order for the boosting hardware to correctly maintain the program semantics, the compiler must communicate its assumptions to the hardware. Thus, whenever the compiler moves an instruction above a control dependent branch, the compiler may label this instruction as a *boosted instruction*. This labelling encodes the control dependence information needed by the hardware so that the hardware can determine when the effects of the boosted instruction are no longer speculative. The labelling indicates which branch or branches the

boosted instruction is control dependent upon, and the labelling indicates the predicted direction of each of these branches.

A speculative instruction that is moved above n control dependent branches is referred to as an instruction that is boosted n levels. The instruction i_2 in Figure 2 is an example of an instruction that is boosted two levels; this is indicated by adding a ".BRR" suffix to the instruction destination. A labelling of ".BRR" indicates that the instruction is dependent upon the next two branches going RIGHT. In this example, the number of Rs or Ls that follow the B indicate the level of boosting while each R (RIGHT) or L (LEFT) indicates the direction of the dependent branch. In general, an independent branch can be included in the sequence by inserting an X (DON'T CARE). A boosting suffix on a destination register implies that a future value has been generated for that register. A boosting suffix on the destination of a memory store operation implies that a future value has been generated for that memory location. In general, a boosting suffix names a readable and writable location for future values, and thus the sources of a boosted instruction may also have boosting level suffixes as in the base register of instruction i_2.

Even though the effects of a boosted instruction are accessible by other instructions boosted along a path, speculative effects do not update the sequential state until after the execution of the last branch upon which the instruction depends. In other words, the result of instruction i_1 in Figure 2 is accessible to instruction i_2, but the result returned by the load instruction is not committed to the sequential state (i.e. the value in r4.BRR is not accessible by the name r4) unless both branches in Figure 2 are correctly predicted. If either branch is incorrectly predicted, the effects of the load operation are prevented from affecting the sequential state. With these semantics, the effects (including the side effects) of the boosted operations only affect the sequential state if the flow of control would have executed those instructions anyway. In terms of the types of speculative execution discussed in Figure 1, boosting effectively renames registers (r1.BR is different from r1) so that speculative movements that would have been illegal are now legal, and boosting postpones all side effects so that speculative movements that would have been unsafe are now safe. The details of how exceptions are handled are given later in this section.
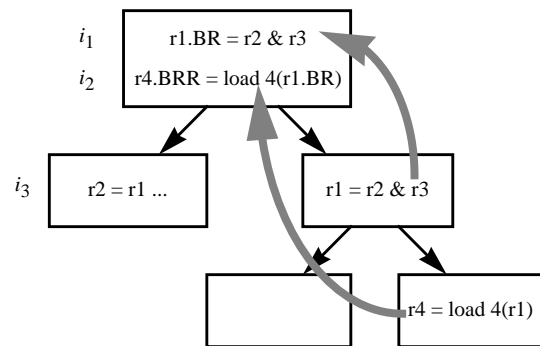

Figure 2: Boosting example.

The most general form of boosting requires hardware exponential in the boosting level, since speculative state is needed for each possible branch prediction path. To limit the hardware to a more reasonable level, we only boost instructions that are speculative on the most-frequently taken direction of a branch (this restriction is used throughout the rest of the paper). Since boosting now applies to the most-frequently taken direction of each branch, the branch instructions can encode the prediction information (i.e. taken or not taken), and each boosted instruction can simply indicate that it is dependent upon the next n conditional branches (e.g. the labelling of the destination register of instruction i_2 in Figure 2 is sim-

_____

2. This problem becomes even more difficult when the path between the excepting point and the signalling point is dynamically determined.

plified from ".BRR" to ".B2"). To simplify matters further, once an instruction is boosted to indicate dependence upon a conditional branch, that boosted instruction is assumed to be control dependent upon all subsequent branches it is moved above. By encoding the boosting level as a count of the number of these control dependent branches, the hardware can reconstruct the control dependence information. That is, a boosted instruction of level n is control dependent upon the execution of the next n conditional branches, and this boosted instruction is committed only if all of the next n conditional branches are correctly predicted. This constraint makes the boosting information easier to encode, and the hardware simpler to build. The actual hardware mechanisms that support these trace-based boosting semantics are discussed in Section 4, and the viability of this simplification is discussed in Section 5.

Before leaving the topic of boosting, we briefly describe the handling of exceptions on boosted operations. The key to our exception handling is the realization that, though the hardware can efficiently postpone exception processing, only the compiler can efficiently determine which instructions to re-execute. In boosting, all speculative exceptions are postponed until the commit point so that no extraneous exceptions are signalled. This postponing is handled through a one-bit shift buffer where each location in the shift buffer corresponds to a level of boosting. If a boosted instruction of level n excepts, the hardware sets the appropriate bit in this buffer and discards the exception signal. On an incorrectly-predicted branch, the buffer is cleared, and all the speculative exceptions are ignored. On a correct prediction, the buffer is shifted and the out-shifted bit is checked. If the bit indicates that no boosted exceptions occurred, the speculative state is committed. If, on the other hand, a boosted exception occurred, the hardware discards the speculative state and invokes a boosted exception handler. This exception handler was constructed by the compiler and lives in user space. The handler uses the address of the committing branch to index into a jump table so that it can vector to a copy of the boosted code (the *recovery code*) that is dependent upon the committing branch. This jump table and recovery code were also generated by the compiler. In this way, the machine regenerates the "speculative" state, except this time, the exception will occur on a sequential instruction (remember that the branch was correctly predicted). This exception is a normal sequential exception, and it can be handled precisely. The recovery code ends in a unconditional jump back to the predicted target of the committing branch. The cost of this solution is an increase in the size of the object file (less than a two-times growth), and an approximate 10-cycle overhead in exception processing time caused by the boosted exception handler. Both of these overheads are quite acceptable since most exception processing routines occur infrequently, and when they do occur, they run for a long time. A more in-depth discussion of boosted exception handling can be found in Smith [25].

Boosting provides the compiler with a clean and unconstrained model of speculative execution. Boosted instructions are free to move far above their dependent branches, and boosting makes exception handling simple and precise. The tradeoff is between different types of hardware complexity (e.g. our renameable buffer space versus the tagged data architecture of the non-excepting instruction architectures).

3 Global Instruction Scheduling

Global scheduling algorithms, such as Trace Scheduling [10] and Percolation Scheduling [20], define a framework within which a compiler can perform code motions across basic block boundaries. This section begins with a discussion of the existing global scheduling algorithms, and explains the tradeoffs involved in constructing a scheduler. The basic issues that need to be resolved are determining what instructions are available to schedule, and how much freedom the algorithm has to generate the schedule. After reviewing previous work we describe our global scheduling algorithm, which is optimized for the scheduling non-numerical codes on modest superscalar machines.

3.1 Background

Global instruction scheduling grew out of the work done on local microcode compaction techniques of the 1970s and early 1980s (see [27] for a comprehensive reference list). The early attempts at global scheduling understood the basic rules for code motion between basic blocks, and they optimized a program by repeatedly moving instructions between dynamically adjacent basic blocks to improve the local schedules. The culmination of these iterative, "neighborhood" scheduling algorithms is Percolation Scheduling [20] which describes a complete set of semantics-preserving transformations for moving any operation between adjacent blocks.

The next step taken by global scheduling researchers was to extend the "neighborhood" to include *conditional pairs* [27] (also called *equivalent basic blocks* [2]). Two basic blocks are equivalent if and only if the execution of one block implies the execution of the other block; equivalence is simply a combination of the *move-op* and *unification* transformations of Percolation Scheduling for control-independent basic blocks. We refer to these types of algorithms as "neighbor and peer" scheduling algorithms. Tokoro [27] discusses some early versions of iterative "neighbor and peer" scheduling. Region Scheduling [13] is another iterative "neighbor and peer" scheduling algorithm that uses a program dependence graph [9] to determine equivalent basic blocks. All of these algorithms repeatedly apply transformations using a "local" policy. This type of incremental scheme does not always lead to a good global schedule.

More recent work has focused on implementing a global instruction scheduler that uses a *global* policy during instruction scheduling. A global schedule must first find a set of *ready* instructions. These are instructions where all their data-dependent predecessor instructions have been scheduled and their latencies fulfilled. To generate this pool of instructions, the scheduler finds the instructions that are *available* for scheduling at a point in the control flow graph (CFG) by determining if there is some *set* of global transformations which result in a ready instance of this instruction. At every point in the generation of an instruction schedule, the schedulers first find the available instructions, then use heuristics to choose which of the available instructions will produce the best schedule, and finally invoke the global transformations to safely move the requested instructions to the current scheduling point. The key difference between schedulers is how the available instruction set is generated, and the types of global transformation used.

Bernstein and Rodeh [2] describe a scheduling algorithm that looks in "neighbor and peer" basic blocks for available instructions. "Neighbor and peer" basic blocks are only a small set of the blocks from which instructions are available, and thus, this decision greatly limits the size of the available set. The calculation of available instructions is further constrained by the use of weak transformation rules. For example, Figure 3 contains part of a CFG in which blocks A and D are equivalent. Let us assume that the machine architecture in this figure uses a delayed branching scheme. Under their definition of availability (called *M-ready* [3]), instruction i_4 is not available for scheduling in the delay slot of the branch at the end of basic block A until instruction i_3 in block B is scheduled. Yet, it is possible to schedule instruction i_4 in block A

by placing a copy of the instruction at the end of the unscheduled block B. If the path ACD is executed more frequently than the path ABD, we definitely want to perform this code motion. Even though instruction i_4 is executed twice along the path ABD, we have not lengthened the execution time of that path (the delay slot cycle exists whether the scheduler fills it or not), and we have shortened the execution time of ACD (the more frequent path).
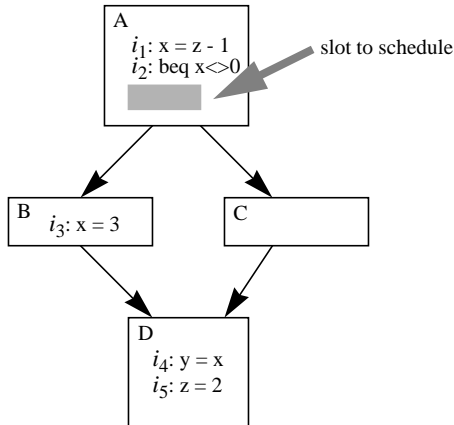

Figure 3: Availability example.

Trace Scheduling [10] was the first attempt at an instruction scheduler with a more global calculation of availability. Trace Scheduling uses probabilities to select a *trace* of basic blocks. From this trace, a directed acyclic graph (DAG) is built which contains all of the necessary constraints on code motion within the trace. This DAG includes extra constraint edges to indicate the limitations of the global transformations. The algorithm then schedules this DAG as if the trace was one large basic block. The calculation of availability is simply a calculation of readiness within the DAG of the trace. Because of the trace, Trace Scheduling looks well beyond "neighbor and peer" blocks for available instructions. Still, the calculation of availability in Trace Scheduling is limited by "tunnel vision". Instructions are available only from basic blocks on the most probable trace and not from all possible successor paths. Furthermore, a trace ends when it reaches a back edge or an already scheduled block; in other words, Trace Scheduling does not implement the movement of instructions between traces.

Though the code transformations in Trace Scheduling are powerful enough to completely determine all the available instructions within a trace, they ignore information that is important in creating space and time efficient code after a global code motion. For instance, to maintain semantic correctness, the movement of instruction i_4 in Figure 3 from block D to block A along the trace ACD requires a copy of that instruction to be placed at the end of block B, but the same movement of instruction i_5 does not require a copy. The transformation rules in Trace Scheduling create copies for both movements. By focusing only on the trace, this algorithm does not take into account the cost of the transformations on the off trace paths. This cost can be significant for applications where there is more than one important trace. The insertion of *compensation code* to maintain semantic correctness in the face of global code motions is referred to as *bookkeeping*. Gross and Ward [12] describe some modifications to Trace Scheduling to improve the transformations and optimize the compensation code, but the general algorithm still does not consider the cost of compensation code on the off-traces during the scheduling of the main trace.

The IMPACT compiler [6] also uses the concept of traces to obtain a scheduling algorithm with a more global calculation of availability. In the IMPACT work, a trace of basic blocks is converted into a *superblock* by code duplication. A superblock is a block of code with a single entry at the top of the block and one or more exits throughout the block. The single entry point ensures that upward code motions in the superblock never require compensation. Though this approach eliminates the determination of whether duplication is required during the scheduling of a superblock, the schedules that are not part of the most-probable superblock are inefficient because all possible code duplications are made before any scheduling takes place (i.e. an instruction is duplicated independent of whether or not it is later moved). Like the original Trace Scheduling algorithm, IMPACT does not worry about the cost of compensation on the less-likely traces.

Ebcioğlu and Nicolau [8] discuss an approach to instruction scheduling called Percolation Scheduling with resources (PSr) that is more global than Trace Scheduling in its calculation of available instructions. In PSr, the available instructions (called the *unifiable-ops*) for a basic block are calculated from all successor blocks on all paths from the current basic block (a path stops when it reaches a back edge in the CFG). That is, PSr computes availability as a dataflow calculation on a CFG with its back edges removed. PSr uses the Percolation Scheduling rules to determine availability and to transform the code after the scheduling of an operation. Like Trace Scheduling, PSr was developed for VLIW machines with a large number of resources, and it seems that the routines to transform the code after scheduling were developed only with concern for correctness and not with concern for efficiency.

3.2 Our Algorithm

Most of the existing algorithms were developed for VLIW machines where resources are abundant, and thus efficient compensation code is a secondary concern. Or, they were developed for the scheduling of numerical applications where traces are extremely predictable, and thus again, efficient compensation code is a secondary concern. In contrast, the goal of our global scheduling algorithm is to produce good schedules for non-numerical applications running on superscalar machines with limited resources. This domain requires an algorithm which is based on powerful *and* efficient transformations, and it requires an algorithm whose heuristics limit the penalties imposed on the less-likely traces.

Section 3.2.1 reviews the top-level scheduling process for the first implementation of our algorithm. This implementation uses traces as our global view of the program structure. We found that most conditional branches in non-numerical code are predictable (see Table 1 on page 9 and Chang et al. [6]), and therefore, traces are a good, first approximation of the entire availability set. Unlike the original Trace Scheduling algorithm though, we maintain the concept of basic blocks within the trace so that can tightly control the scheduling process and limit the penalties imposed on the less-likely traces.

Our global scheduling algorithm relies on transformations which implement only upward code motion—the motion of instructions against the direction of the edges in the CFG. Section 3.2.2 overviews the implementation of our upward code motion algorithm. This algorithm inserts compensation code to maintain semantic correctness during upward code motion. Since boosting augments the capabilities of this algorithm, our bookkeeping process not only includes the duplication of instructions, but also the boosting (i.e. speculative marking) of instructions. Finally, this subsection discusses the use of liveness and equivalence information to improve the efficiency of the compensation code produced by upward code motion algorithm.

3.2.1 A Trace-Scheduling Framework

Figure 4 contains an outline of our global scheduling algorithm. At the outermost level, the algorithm schedules one procedure at a time. Within each procedure, scheduling proceeds from innermost to outermost regions, where a region is either a loop or the procedure body. The algorithm does not perform code motions across a region boundary; traces are constrained to remain within a region. Traces are selected and scheduled within a region until no unscheduled traces exist. At this point, the region is collapsed, and its data-flow information is summarized so that code motions can occur around this inner region.

```
foreach PROCEDURE {
    generate CFG and compute global data-flow info;
    foreach REGION (innermost loop out to procedure level) {
        while (exists unscheduled TRACE) {
            select next best TRACE;
            foreach BB in TRACE {
                list schedule BB;
                fill in the holes through upward code motion;
            }
        }
        collapse REGION;
    }
}
```

Figure 4: Overview of our global scheduling algorithm.

Trace selection proceeds through a region in a topological order. The next unscheduled basic block is choosen as the start of the new trace. A trace is grown from this basic block using branch probabilities until one of four conditions is met: the next block is not in the current region (e.g. a call); the next basic block is dynamically determined (e.g. an indirect jump); the next basic block is already in the trace (e.g. a loop edge); or the next basic block is already scheduled. For the last two conditions, the trace is extended one more basic block to mitigate the usual lack of scheduling lookahead associated with the end of a trace. During the construction of the trace, two data structures are built. One is a simple data dependence graph of all the instructions in the trace, and the other captures the control dependence and off-trace data dependence information needed by the upward code motion routine.

Within each trace, a list scheduling algorithm is used to top-down, cycle schedule each basic block. By viewing a trace as a collection of basic blocks, we can tightly control the scheduling process. This tight control exists for two reasons. First, we are scheduling for superscalar machine models with small amounts of parallel resources; we do not have a lot of spare resources to handle excessive compensation code. Second, the traces in non-numerical applications are not as obvious as they are in numerical code, and we do not want to penalize the off-trace paths with excessive compensation code. Consequently, the list scheduling algorithm gives priority to those instructions that originally lived in the current basic block[3], and the instructions are not allowed to move down out of the current basic block. The list scheduler tries to fill in the empty slots in the current basic block schedule with instructions from basic blocks later in the trace. In this way, a basic block schedule is never lengthened by a global code motion; global code motions only occur to fill empty instruction slots. Our scheduler may not produce as good a schedule as Fisher's Trace Scheduling algorithm would produce for a very probable trace, but our scheduler never lengthens an off-trace schedule.

The calculation of available instructions is extremely simple in our current algorithm. To the list scheduler, the data dependence graph for the trace looks just like a DAG for a basic block, and thus a ready instruction in the data dependence graph is an available instruction. Because boosting supports general speculative execution, our available set is larger than the available set for Trace Scheduling. No edges are added to our data dependence graph to enforce control dependence constraints. Our algorithm only adds edges to the data dependence graph to maintain the original order of the branches. This ordering is imposed to minimize the potential for code explosion during scheduling.

Currently, our algorithm separates instruction scheduling and register allocation. Though a number of techniques exist for handling the interaction between register allocation and instruction scheduling [4][11], we only perform instruction scheduling after register allocation. This means that scheduling is constrained by anti- and output dependences created by the register allocator, and we try to minimize these dependences by using a round-robin register allocator. Our scheduler is capable of scheduling instructions for an infinite register model. In this way, we can bound the performance of an integrated register allocator and instruction scheduler.

3.2.2 Upward Code Motion

Bookkeeping is done during our upward code motion routine. All instructions (except branches)[4] are allowed to move upward across multiple basic block boundaries. Our algorithm for upward code motion follows three basic rules:

(1) a rule for intra-block motion,

(2) a rule for motion out of the top of a block, and

(3) a rule for motion into the bottom of a block.

The rule for intra-block motion simply involves the movement of an instruction over earlier instructions in the basic block. This motion is inhibited by earlier instructions which impose data dependences upon the instruction being moved. For instance, we cannot move an instruction above the availability of its operands. If we assume that all preceding, data dependent instructions have already been moved up by the application of the code motion rules, then the current instruction is free to flow up to the top of the basic block.

Once an instruction is at the top of the block, it is free to move to the bottom of the preceding blocks. A copy of the instruction must be placed at the end of each preceding basic block so that the instruction executes on every path that reaches the current basic block. Thus, the motion of an instruction out of the top of a block can require duplication. To ensure that the copied instruction is only executed on those paths that reach the current basic block, this instruction is labelled as boosted in each preceding basic block that has multiple successors. Boosting guarantees that the effects of an instruction are committed only if the predicted edge (the CFG edge that the instruction moved across) is taken. Thus, the motion of an instruction into the bottom of a block can require boosting.

Now that the instruction is at the bottom of a block, we are again at the point where we can apply the first rule. Thus, through successive application of the three basic rules, we can continue to move an instruction up through the CFG. These three rules are sufficient

---

3. There is a whole set of heuristics that further prioritize the ready instructions, but they are uninteresting to this discussion.

4. Branch instructions can be duplicated or boosted, but they are never moved further than into the preceding basic block (due to delay slot scheduling problems).

because they cover all possible entry and exit configurations for a basic block in a CFG.

Boosting makes these rules simple because the algorithm is never limited by a speculative code motion that is unsafe or illegal. It is interesting to note that this is exactly how a dynamic scheduler handles speculative execution—all instructions moved across a branch point are dependent upon that branch. Though these basic rules are a sufficient solution for upward code motion, they are not an efficient solution. These rules boost and duplicate instructions more often than necessary because they do not take advantage of global data-flow information.

Boosting is only necessary for a code motion into a block with multiple successors if the speculative movement is unsafe or illegal. Unsafe speculative movements are easily determinable by checking if the current instruction is capable of signalling an exception. Illegal speculative movements are recognizable if we can determine what values are needed when the non-predicted edge of the block is taken. This information is exactly the information that is provided by live variable analysis [1]. By checking the live-IN sets of the non-predicted successor blocks against the destination register of the current instruction, an algorithm can determine when a speculative movement is illegal. By using the exception and live variable analysis information, an algorithm can boost instructions only when necessary for correctness.

Duplication is used to ensure that an instruction moved above a basic block with multiple predecessors (a *join* block) is executed on every path reaching the join block. Duplication, however, is not required for the upward motion of every instruction out of every join block. Equivalence catches the most important case where duplication may not be required for the upward code motion of an instruction. For example, blocks A and D of Figure 3 are equivalent because the execution of one implies the execution of the other, and an instruction moved from block D to block A is always *useful* (not speculative). We refer to this condition as *control equivalence*.

Control equivalence is not a sufficient condition to remove the need for duplication. The moving instruction must also be free of data dependences with any instruction along any path between the control equivalent blocks. If this second condition holds, we say that the two blocks are *data equivalent* with respect to the moving instruction. If two blocks exist that are both control equivalent and data equivalent with respect to the moving instruction, the code motion algorithm can simply move the instruction between the two blocks without duplication (or boosting). By checking for control/data equivalent pairs of basic blocks during code motion, we can reduce the amount of compensation code produced. Figure 5 outlines our upward code motion algorithm that uses global data-flow information to reduce the amount of boosting and duplication performed.

The upward movement of an instruction and the creation of compensation code can cause changes to the CFG and to the state of the dependence structures. Our trace-based approach allows for on-demand creation of basic blocks to hold duplicated instructions, and it simplifies the dynamic update of the dependence structures due to a duplication. An in-depth discussion of our algorithm can be found in Smith [25].

In summary, our global scheduling algorithm can be thought of as conscientious trace scheduling. It is conscientious because the scheduler is aware of the compensation costs of each code motion, and because the code transformations try to minimize the creation of compensation code. Though a trace-based approach limits the size of our available instruction set, a trace ensures that our set contains those instructions of the larger set which are most benefi-

```
given an instruction I in block A to move;
given a path from block B to block A;
while (A != B) {
    move I to top of A;
    if (control/data equivalent pair to A exists on path) {
        move I to bottom of pair;
        A = equivalent pair on path;
    } else {
        foreach (predecessor C of A) {
            duplicate I at end of C;
            if (duplicate unsafe or illegal) boost duplicate;
        }
        remove I from A;
        A = predecessor of A on path;
    }
}
```

Figure 5: An efficient algorithm for upward code motion.

cial to the creation of a good global schedule. Furthermore, our trace-based approach overcomes many of the problems associated with the original Trace Scheduling algorithm such as lack of overlap between traces and inefficient compensation code. In these ways, we feel that our algorithm, even without boosting, produces efficient code for superscalar processors with limited resources. Boosting simply provides the scheduler with a straightforward mechanism to move any instruction upward over multiple basic blocks. Since boosting and our global scheduling algorithm are orthogonal, we can use our algorithm as an effective base for experimenting with boosting.

4 Evaluating Hardware Support for Boosting

Boosting is a powerful technique for supporting general speculative execution that requires specific hardware support. In this section, we first describe the hardware support that is necessary for global scheduling using boosting. Then we discuss three options for reducing the amount of hardware support for boosting, and we describe how our global scheduling algorithm is modified to generate code for each of these restrictive options. Finally, we choose four machine models which vary only in their hardware support for boosting. By collecting cycle-time independent performance numbers and by understanding the complexity of the hardware support for boosting, we can evaluate exactly how much boosting support is really necessary to achieve good performance. We find that very little hardware is required to support speculative execution efficiently.

4.1 Full Support for Boosting

The effects of boosted instructions are held in hardware *shadow* structures from the time the boosted instruction is executed until the time that the boosted effect is squashed or committed. To determine what shadow structures are necessary, we must determine what effects are possible for all non-branch[5] instructions in a particular architecture. We use the MIPS R2000 architecture [17] as our base architecture. In the MIPS architecture, there are three possible effects from instruction execution: a register is written, a memory location is written, or an exception is signalled. Shadow structures are therefore required for the register file and for the

---

5. Due to limited movement of branches in our scheduler, boosted branch effects are always squashed in the pipeline, and thus no additional shadow structures are needed.

store buffer. Any exception signalled by a boosted instruction is handled as discussed in Section 2.3.

In the simplest case, the shadow structures can be thought of as copies of the sequential structure. For instance, we can construct a shadow register location for each sequential register location. However, a single shadow structure supports only a single level of boosting. To support boosting across multiple branches, each shadow structure must contain a location for each allowable level of boosting. Thus, in the general case, we must construct n shadow register locations for each sequential register location. Figure 6b illustrates a legal instruction schedule that is possible when r3.B1 and r3.B2 are each separate physical locations.[6]
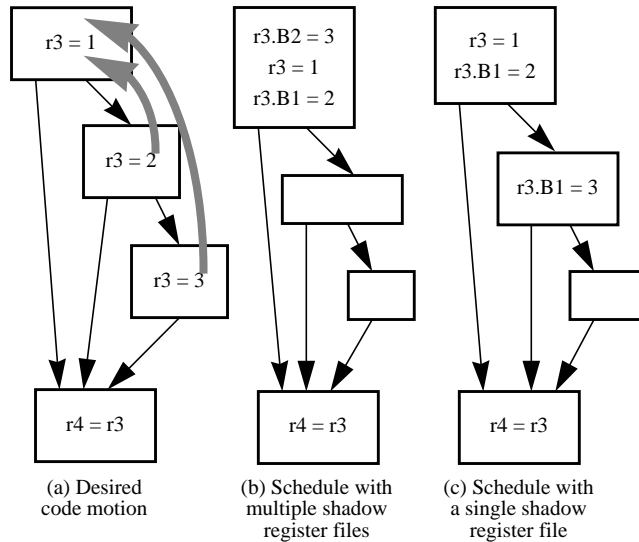

(a) Desired code motion   (b) Schedule with multiple shadow register files   (c) Schedule with a single shadow register file

Figure 6: Schedules possible under different shadow register file configurations.

Though these general shadow structures require a large amount of hardware, this hardware is straightforward to implement. The key insight in implementing the shadow structures is to realize that the data needs only to logically move on a commit. This logical move is implemented by a technique that is similar to register renaming [18]. For each sequential register in the architecture, we physically build a pool of register and counter pairs. These counters contain the logical name of each physical register. The register in the pool with a count value of 0 holds the sequential state, the one with a count value of 1 holds the boosted-level-one state, etc. Additionally, a valid bit is kept with each register in the pool to indicate whether a valid boosted value exists for this register. During a commit, we want the data in the shadow state to shift down one level of boosting. Rather than moving the data, each counter is decremented since each counter represents the boosting level of its register. The most complex part of this hardware is to make sure that the sequential register is only updated if the boosted-level-one register contains valid data. If it doesn't, then the sequential register counter is not decremented (it stays at 0) and the boosted-level-one counter (which was at 1) is set to the maximum boosting level (essentially it is decremented twice).

---

6. We continue to follow the trace-based boosting notation assumed at the end of Section 2.3.

4.2 Partial Support for Boosting

Providing full support for the movement of any instruction above multiple conditional branches requires a significant amount of hardware as described in the previous subsection. We can reduce the amount of speculation hardware necessary if we constrain the speculative code motions that our scheduler is allowed to perform. In this way, we trade off hardware complexity for performance and compiler complexity. Whether this is a good tradeoff depends upon how much hardware complexity we are able to reduce and still achieve effective instruction scheduling. The rest of this subsection discusses three options for constraining the instruction scheduler and reducing the hardware support necessary. Section 4.3 evaluates four different combinations of these options to determine exactly how much boosting support is necessary for good performance.

In Option 1, we propose that the shadow store buffer be removed. Without a shadow store buffer, the scheduler cannot boost any store instructions, but since the MIPS architecture is a load/store architecture, the scheduler is still free to boost any non-store instructions. This restriction implies that the scheduler cannot boost a calculation that involves a store to memory and then a load of that value from memory. However, given an architecture with enough registers and a compiler with an effective register allocator, this sequence should occur infrequently, and therefore the penalty of this restriction should be minimal.

In Option 2, we propose that the multiple shadow register files be collapsed into a single shadow register file that is capable of handling multiple levels of boosting. Without a distinct storage location for each possible level of boosting, r3.B1 and r3.B2 in Figure 6b refer to the same physical storage location, and the compiler must handle this output-like dependence when it schedules the code. In other words, the scheduler must produce the code schedule in Figure 6c. If this output-like dependence occurs infrequently or the limited overlap of operations as in Figure 6c is sufficient, the cycle-count penalty of this restriction should be minimal.

For supporting multiple levels of boosting, Option 2 significantly reduces the amount of hardware support necessary because it requires only two registers, one counter, and one valid bit for each sequential register name. Figure 7 illustrates how the hardware for this scheme is organized. The counter maintains the current boosting-level of the value in the shadow register. This counter is decremented each time a branch executes that was predicted correctly. If the count field is one and the register holds valid data, then on the next correct branch prediction the flip-flop is toggled to "pong" the registers—the boosted register becomes the sequential register and vice versa. This hardware implements the shadow state and only adds a single gate to the register file access path.

For Option 3, we propose that the entire shadow register file be removed. To still allow for the boosting of operations, we augment the processor's pipeline control so that the scheduler is able to boost into the "shadow" of the conditional branch. This scheme is basically an extension of squashing branches where instructions are nullified in the cycles following the branch if the branch was incorrectly predicted. For our scheme, only the boosted instructions in the cycles following the branch are nullified, not all the instructions in those cycles. Without any shadow storage in the machine, the scheduler is constrained to boosting only a few cycles into a branch-ending basic block. For the MIPS R2000 pipeline, the instructions issued with the branch and in the branch delay slot are simple to squash in the pipeline. This scheme requires the least amount of hardware support, but it also imposes the greatest constraints on instruction scheduling.

For one pair of registers

word line A

word line B

boosting specifier

decoder — register specifier

modulo counter (size of boosting specifier)

ready to commit?

Q Q

T

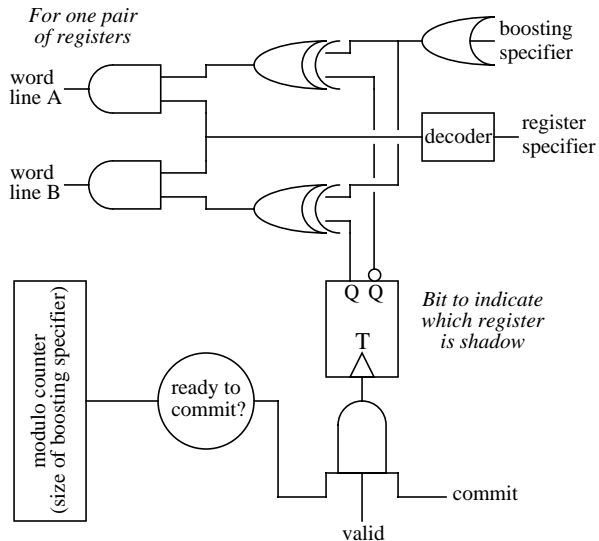*Bit to indicate which register is shadow*

commit

valid

Figure 7: Hardware functionality of the Option 2.

4.3 Performance Evaluation

Table 1 lists the benchmark set we used to evaluate the different options for boosting. There are three SPEC benchmarks and four UNIX utilities. All of these programs are written in C, and all are run to completion[7].

	Total R2000 Cycles	Avg. R2000 IPC	Branch Prediction Accuracy
awk	47.2M	0.89	82.0%
compress	29.3M	0.87	82.7%
eqntott	0.7M	0.95	72.1%
espresso	97.8M	0.89	75.7%
grep	28.6M	0.81	97.9%
nroff	66.4M	0.82	96.7%
xlisp	1.0M	0.89	83.5%

Table 1: Benchmark programs and their simulation information.

Our base scalar machine model is the MIPS R2000. All results are derived from a trace-driven simulator based on pixie [24], though we also have a functional simulator that verifies that the hardware is correct and an instruction-level simulator that verifies that the scheduled code is correct. The trace-based simulator reports the performance of the superscalar processor in terms of speedup over the MIPS R2000 processor, where speedup is a defined as the total number of R2000 cycles divided by the total number of superscalar cycles. The scalar program is scheduled by the commercial MIPS assembler, the superscalar program is scheduled with our global instruction scheduler, and both schedulers start with the same optimized assembly file. The assembly file is generated by our SUIF compiler, and the optimizer in this compiler implements all the standard optimizations [26]. Our scheduler uses a branch profile of the program to generate the static branch prediction information needed during scheduling. This branch profile is generated from a

different input set than is used to determine performance. Table 1 also lists the observed branch prediction accuracy.

For the results presented in this paper, our trace-driven simulator assumes a perfect memory system, i.e. the caches never miss. Thus, we are only modelling the execution time of the CPU. The true speedup of our superscalar processor over a scalar processor is dependent upon the effectiveness of the memory system. The more effective the memory system, the closer these CPU speedups represent the speedups of the entire system.

4.3.1 Superscalar Base Model

Our base superscalar processor is similar to those commercially designed today: the issue size is kept small—2 instructions per cycle; the issue mechanism is restricted—not all pairs of instructions can issue together; and only a single data memory port is provided. We distribute the functional units so that one side of our 2-issue machine contains an integer ALU, a branch unit, a shifter, an integer multiply/divide unit, and a floating-point unit; the other side contains just an integer ALU and a memory port. It is interesting to note that we can perform two integer ALU operations in parallel, but not a branch and a shift operation in parallel. The base superscalar machine also assumes that an instruction that is fetched for one side of the machine can execute on that side. The scheduler is responsible for ensuring that instructions are issued to the correct side of the machine; there is no swap logic. Each functional unit in the superscalar model has the same characteristics as that functional unit has in the MIPS R2000 processor. Like the R2000, load and branch instructions have a single delay slot. The base superscalar processor contains no hardware support for boosting.

Since the cycle time of this simple superscalar processor should be very close to the cycle time of a single-issue implementation of this processor, we can use cycle counts to indicate the relative performance of the base superscalar machine over the base scalar machine. Figure 8 presents speedups for the case where our instruction scheduler only schedules within a basic block, and for the case where the instruction scheduler uses our global instruction scheduling algorithm to move instructions past basic block boundaries. In the global scheduling case, only safe speculative executions are permitted since the base superscalar processor does not contain hardware to support boosting. The lower portion of each global scheduling bar represents the performance when register allocation is done before instruction scheduling. The upper portion of each bar represents the increase in performance when scheduling is done with an infinite register model. (Figure 8 presents the speedups for basic block scheduling only with register allocation before scheduling. Unless specifically stated otherwise, our discussion focuses on the performance numbers with register allocation done before scheduling.) The simplest superscalar processor that we can build uses global scheduling (no boosting), and it gets a geometric mean speedup of 1.24x under global scheduling.

4.3.2 Superscalar Models with Boosting

To this base superscalar, we now add hardware to support speculative execution. The four augmented machine models are called: *Squashing*, *Boost1*, *MinBoost3*, and *Boost7*. To evaluate the usefulness of the additional hardware, we normalize the performance of the augmented superscalar processors to the best performance of our base superscalar processor (i.e. performance under global scheduling with register allocation and without boosting). Table 2 presents the percentage improvement in performance for these augmented superscalar processors.

---

7. The SPEC benchmarks were run with the shorter of the distributed input data sets to limit the run time of the simulations.
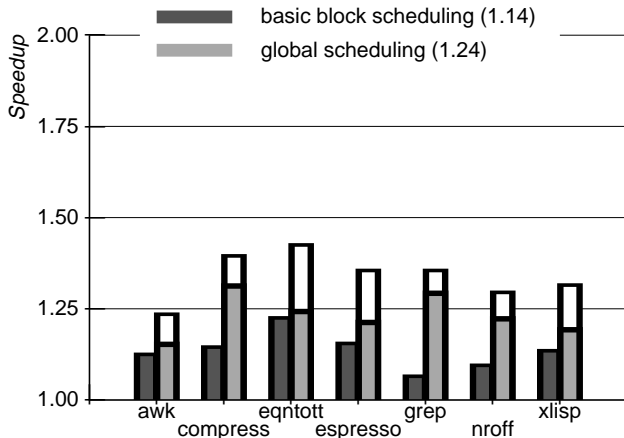
Figure 8: Performance achievable without speculative execution hardware.

	Squashing	Boost1	MinBoost3	Boost7
awk	11.2%	16.4%	17.2%	18.1%
compress	9.1%	10.6%	10.6%	10.6%
eqntott	8.0%	14.4%	16.0%	16.0%
espresso	9.8%	18.0%	21.3%	23.0%
grep	15.4%	27.7%	40.8%	40.8%
nroff	11.4%	24.4%	31.7%	36.6%
xlisp	6.7%	13.3%	12.5%	14.2%
G.M.	9.9%	17.0%	19.3%	20.5%

Table 2: Performance improvements over global scheduling from exploiting various degrees of speculative execution.

The *Squashing* model achieves slightly less that a 10% improvement in performance over our base superscalar processor. The Squashing model contains no shadow structures and boosting is only supported by a squashing pipeline. This scheme adds the smallest amount of hardware possible for a scheme that supports boosting. With this limited boosting ability, our global scheduler with boosting is constrained to boosting only those instructions in the delay branch cycle of a branch-ending basic block, as discussed in Option 3 of Section 4.2.

The *Boost1* model achieves slightly less than double the improvement of the Squashing model. The Boost1 model contains a single shadow register file and a single shadow store buffer without any extra logic to perform multiple levels of boosting. This scheme adds a small amount of hardware to support the boosting of all types of non-branch instructions, but the boosted instructions are constrained to depend only on a single conditional branch. Referring to the hardware in Figure 7, the shadow register file hardware for the Boost1 model does not contain any counters, and the gate which clocks the T flip-flop is simply an AND of *valid* and *commit*. Since the boosting specifier is only one bit, the OR-gate of Figure 7 is also not necessary.

The *MinBoost3* model doubles the improvement of the Squashing model. The MinBoost3 model contains a single shadow register file that supports boosting over three control dependent branches, but it does not contain any shadow store buffer. This scheme adds the smallest amount of hardware possible for a scheme that supports boosting over a large number of branches. For this scheme,

the scheduler is constrained as discussed in Options 1 and 2 of Section 4.2.

The *Boost7* model achieves only a small performance benefit over the Boost1 or MinBoost3 models. The Boost7 model contains all the hardware necessary to support unconstrained boosting over seven control dependent branches. The hardware in this scheme is obviously unreasonable, and the performance numbers show that this amount of extra hardware does little to improve performance.

The best schemes for cost-effective performance are Boost1 and MinBoost3. Both schemes are basically advocating a duplicated register file. As we mentioned in Section 4.2, the addition of a single shadow register file causes the register file access time to be approximately one gate delay longer than the access time of the simple scalar register file. Since the register file is not currently in the critical path of our implementation, we do not expect this complexity to increase the cycle time of our processor. Also, the additional hardware required by our more complex register file is not large. The decoder for a Boost1 machine with 32 sequential registers contains only 33% more transistors than a normal decoder for a register file with 64 registers (50% more transistors are required for a MinBoost3 implementation).

The performance improvements in Table 2 are over the global scheduling model with a 32-register register file. Furthermore, this model is constrained by illegal speculative movements because it does not perform register renaming during scheduling. An interesting question to ask is whether the Boost1 or MinBoost3 schemes actually do better than a global scheduling scheme which incorporates software renaming and a 64-register register file. Though we cannot generate code for this enhanced global scheduling scheme with our current compiler, we have the performance of our global scheduling scheme with an infinite register model (see Figure 8). Global scheduling with infinite registers achieves a 7.8% performance improvement (geometric mean) over our base global scheduling scheme. This is a smaller improvement than that achieved by Boost1 and MinBoost3. Thus hardware support for unsafe speculative code motions improves machine performance beyond the best performance of the pure software schemes.

Figure 9 compares the speedup (relative to the base scalar processor) of the MinBoost3 model to the speedup of a dynamically-scheduled superscalar processor with speculative execution support. The dynamic scheduler is functionally equivalent to our base superscalar machine. The dynamic scheduler fetches and decodes two instructions per cycle. It uses a total of 30 reservation station locations [28] and a 16-entry reorder buffer [21] to implement out-of-order execution with speculation, and it uses a 2048-entry, 4-way set associative branch target buffer to predict branches. It has the same number of functional units as our statically-scheduled machine, but since the dynamically-scheduled machine uses reservation stations, it can issue up to 6 instructions per cycle. In Figure 9, the lower portion of the MinBoost3 bars represent the performance when register allocation is done before instruction scheduling, and the lower portion of the dynamic scheduler bars represent the performance without register renaming logic. The upper portion of the MinBoost3 bars represent the increase in performance when scheduling is done with an infinite register model, and the upper portion of the dynamic scheduler bars represent the performance with register renaming logic. Under the more restricted register model, both machine models achieve about a 1.5x performance improvement over the scalar processor, but the dynamic scheduler does so with a larger amount of hardware.

We believe that we can improve the performance of the dynamic scheduler by using our global scheduling algorithm (without boosting) to preschedule the code. By prescheduling, we can more
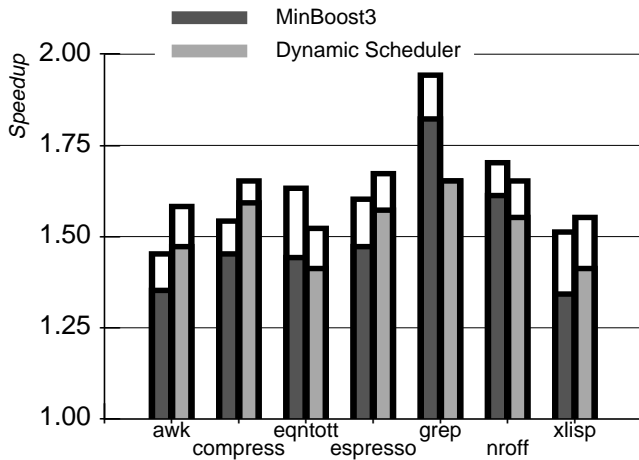
Figure 9: Performance comparison with an dynamic scheduler.

efficiently use the machine resources. We also believe that we can improve the performance of MinBoost3 by carefully adding ILP-increasing optimizations (such as loop unrolling and procedure inlining) to our compiler. We have performed some preliminary experiments with a loop unroller which unrolls all the loops in a program module. Though performance did increase slightly, the improvement was well below what we expected. Upon closer inspection of the code, we discovered that no one problem is limiting the performance. What is required is an effective mix of solutions to problems in many areas (e.g. loop-level optimizations, procedure inlining, and better memory disambiguation). We are continuing our research in this area.

5 Conclusions

This paper presents an integrated solution, consisting of both compiler and architectural components, to the problem of effectively exploiting ILP. Our focus is on supporting *general* speculative execution in an environment with sophisticated instruction scheduling. Speculative execution is needed to find any significant amount of ILP in non-numerical applications. The key idea in our approach is in defining the right architectural interface; an interface that appropriately divides the work among the hardware and software. Our boosting approach relies on simple but useful hardware mechanisms that minimally impact the cycle time of the machine. Our preliminary hardware designs indicate that the cycle time of a superscalar machine with boosting past one conditional branch is nearly identical to the cycle time of a simple (VLIW-like) superscalar machine. By making these hardware mechanisms visible to the software, a sophisticated static instruction scheduler can take full advantage of speculative execution with very little cost.

We have developed a global scheduling algorithm that is useful for machines with and without boosting. Our algorithm is applicable to a wide range of machines, from deeply-pipelined RISC processors to dynamically-scheduled superscalar machines. Our algorithm uses a trace-based approach to efficiently exploit the ILP in non-numerical applications. Like other trace-based approaches, our algorithm concentrates on those code motions that are most beneficial to improving performance. Unlike other trace-based approaches, our algorithm tempers the global movements so as to not adversely affect the performance of the off-trace schedules.

As a result of our compiler implementation and in-depth hardware design, we have been able to evaluate our ideas and study a range of cost/performance tradeoffs. For the limited superscalar machines built today, our statically-scheduled approach achieves cycle count speedups which are comparable to those found in the aggressive dynamically-scheduled approaches.

Acknowledgments

References

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

[2] D. Bernstein and M. Rodeh. Global Instruction Scheduling for Superscalar Machines. In *Proc. ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation*, pp. 241–255, June 1991.

[3] D. Bernstein, D. Cohen, and H. Krawczyk. Code Duplication: An Assist for Global Instruction Scheduling. In *Proceedings of MICRO-24*, pp. 103–113, November 1991.

[4] D.G. Bradlee, S. J. Eggers, and R.R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. In the *Proc. Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 122–131, April, 1991.

[5] B. Case. Superscalar Techniques: SuperSPARC vs. 88110. *Microprocessor Report*, 5(22):1–11, December 1991.

[6] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Warter, and W.W. Hwu. IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors. In the *Proc. 18th Int. Symp. on Computer Architecture*, pp. 266–275, May 1991.

[7] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Papworth, P.K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. In the *Proc. Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 180–192, October, 1987.

[8] K. Ebcioğlu and A. Nicolau. A Global Resource-Constrained Parallelization Technique. In *Proc. 3rd Int. Conf. on Supercomputing*, pp. 154–163, June 1989.

[9] J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319-349, July 1987.

[10] J.A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. on Computers*, C-30(7):478-490, July 1981.

[11] J.R. Goodman and W.C. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proc. 1988 Int. Conf. on Supercomputing*, pp. 442–452, July 1988.

[12] T. Gross and M. Ward. The Suppression of Compensation Code. In *Advances in Languages and Compilers for Parallel Processing*, The MIT Press, Cambridge, MA, pp. 260–273, 1991.

[13] R. Gupta and M.L. Soffa. Region Scheduling: An Approach for Detecting and Redistributing Parallelism. *IEEE Trans. on Software Engineering*, 16(4):421–431, April 1990.

[14] P.Y.T. Hsu and E.S. Davidson. Highly Concurrent Scalar Processing. In *Proc. 13th Int. Symp. on Computer Architecture*, pp. 386–395, June 1986.

[15] W.W. Hwu and Y.N. Patt. Checkpoint Repair for Out-of-order Execution Machines. In *Proc. 14th Int. Symp. on Computer Architecture*, pp. 18–26, June 1987.

[16] N.P. Jouppi and D.W. Wall. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. In *Proc. Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 272–282, April 1989.

[17] G. Kane. *MIPS R2000 RISC Architecture*. Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[18] R.M. Keller. Look-Ahead Processors. *Computing Surveys*, 7(4):177–195, December 1975.

[19] S. Melvin and Y. Patt. Exploiting Fine-Grained Parallelism Through a Combination of Hardware and Software Techniques. In *Proc. 18th Int. Symp. on Computer Architecture*, pp. 287–296, May 1991.

[20] A. Nicolau. Percolation Scheduling: A Parallel Compilation Technique. Computer Sciences Technical Report 85-678, Cornell University, May 1985.

[21] J.E. Smith and A.R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. In *Proc. 12th Int. Symp. on Computer Architecture*, pp. 36–44, June 1985.

[22] M.D. Smith, M. Johnson, and M.A. Horowitz. Limits on Multiple Instruction Issue. In *Proc. Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 290–302, April 1989.

[23] M.D. Smith, M.S. Lam, and M.A. Horowitz. Boosting Beyond Static Scheduling in a Superscalar Processor. In the *Proc. 17th Int. Symp. on Computer Architecture*, pp. 344–354, May 1990.

[24] M.D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, November 1991.

[25] M.D. Smith. Ph.D. thesis, Stanford Univ., in preparation.

[26] S.W.K. Tjiang and J.L. Hennessy. Sharlit—A Tool for Building Optimizers. In *Proc ACM SIGPLAN '92 Conf. on Programming Language Design and Implementation*, pp. 82–93, June 1992.

[27] M. Tokoro, E. Tamura, and T. Takizuka. Optimization of Microprograms. *IEEE Trans. of Computers*, C-30(7):491–504, July 1981.

[28] R.M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal*, 11(1):25–33, January 1967.

[29] D.W. Wall. Limits of Instruction-Level Parallelism. In *Proc. Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 176–188, April 1991.