Compiler Technology for Future Microprocessors

Wen-mei W. Hwu Richard E. Hank David M. Gallagher* Scott A. Mahlke†
Daniel M. Lavery Grant E. Haab John C. Gyllenhaal David I. August

Center for Reliable and High-Performance Computing
University of Illinois
Urbana, IL 61801

Correspondent: Wen-mei W. Hwu
Tel: (217)-244-8270
Email: w-hwu@uiuc.edu

Abstract

Advances in hardware technology have made it possible for microprocessors to execute a large number of instructions concurrently (i.e., in parallel). These microprocessors take advantage of the opportunity to execute instructions in parallel to increase the execution speed of a program. As in other forms of parallel processing, the performance of these microprocessors can vary greatly depending on the quality of the software. In particular, the quality of compilers can make an order of magnitude difference in performance. This paper presents a new generation of compiler technology that has emerged to deliver the large amount of instruction-level-parallelism that is already required by some current state-of-the-art microprocessors and will be required by more future microprocessors. We introduce critical components of the technology which deal with difficult problems that are encountered when compiling programs for a high degree of instruction-level-parallelism. We present examples to illustrate the functional requirements of these components. To provide more insight into the challenges involved, we present in-depth case studies on predicated compilation and maintenance of dependence information, two of the components that are largely missing from most current commercial compilers.

1 Introduction

The microprocessor industry continues to successfully meet the demand for increased performance in the market place. In 1994, high-end microprocessors [1] [2] executed integer code at about 100

*With the Department of Electrical and Computer Engineering, Air Force Institute of Technology, WPAFB, OH 45433

†With Hewlett-Packard Laboratories, 1501 Page Mill Rd., MS 3L-5, Palo Alto, CA 94304

times the performance of a high-end microprocessor introduced in 1984 [3].[1] Such rapid growth in microprocessor performance has stimulated the development and sales of powerful system and application programs that demand even more performance. This positive cycle has helped create a very high rate of growth for the microprocessor industry. In order to sustain such growth, the microprocessor industry will continue to seek innovations required for the sustained performance improvement of their products.

An area of innovation that is important to the next major boost of microprocessor performance is instruction-level parallel processing. This is reflected by the fact that high-performance microprocessors are increasingly designed to exploit instruction-level parallelism (ILP). While mainstream microprocessors in 1990 executed one instruction per clock cycle [4] [5], those in 1995 execute up to four instructions per cycle [6] [7]. By the year 2000, hardware technology is expected to produce microprocessors that can execute up to sixteen instructions per clock cycle. Such rapid, dramatic increases in the hardware parallelism have placed tremendous pressure on the compiler technology.

Traditionally, optimizing compilers improve program execution speed by eliminating unnecessary instruction processing [8]. By keeping memory data and intermediate computation results in high speed processor registers, optimizing compilers reduce the program execution cycles spent waiting for the memory system and performing redundant computation. While this model of optimization will remain important to the performance of future microprocessors, the rapidly increasing hardware parallelism demands parallelization techniques that are currently missing from most commercial compilers.

The quality of compiler parallelization techniques can potentially make an order of magnitude difference in program execution performance for processors that exploit ILP. For a processor that executes up to eight instructions per cycle, a well-parallelized program can run at several times the

[1] The measure of integer code performance is the SPECint number published by SPEC (Standard Performance Evaluation Corporation), an industry consortium for standardizing performance comparison practices in the computer industry.

speed of a poorly parallelized version of the same program. With so much at stake, one can expect industry to incorporate parallelization techniques in their compilers in the near future.

The purpose of this paper is to give an advanced introduction to the new generation of compiler parallelization technology. The reader is assumed to have a general interest in microprocessors such as programming microprocessors or designing systems using microprocessors. This paper begins with an overview of the techniques required to parallelize general purpose programs. These techniques address the problems caused by frequent control decisions, dependences between register accesses, and dependences between memory references. The overview is designed to give the reader a functional understanding of these techniques rather than detailed theorems and algorithms.

To provide the reader with further insight into the challenges involved, we present two case studies. The first case study examines compiling for predicated execution, a technique to remove the performance limitations imposed by frequent control decisions on high performance processors. We describe the basic transformations needed to make use of this hardware feature as well as the more advanced optimizations required to handle difficult situations.

The second case study examines the maintenance and use of memory dependence information throughout the compilation process. This capability is critical for parallelizing programs that use sophisticated data structures. These two case studies were selected for several reasons. First, they both touch upon critical aspects of parallelization. Inadequate capability in either area can result in excessive constraints on parallelization. Second, most current commercial compilers have inadequate capabilities in both areas. Thus, they are prime examples of the difference between new and traditional compiler technologies.

2 ILP Compilation

To successfully parallelize a program, the compiler must perform three tasks. First, it must accurately analyze the program to determine the dependences between instructions. Second, it must perform *ILP optimizations* which remove dependences between instructions, allowing them to be executed in parallel. Finally, it must reorder the instructions, a process known as *code scheduling*, so that the microprocessor can execute them in parallel. When performing these tasks, the compiler must examine a region of the program which is large enough to potentially contain many independent instructions. These tasks are collectively referred to as *ILP compilation*.

2.1 Basic Concepts

Figure 1(a) shows an example loop written in C. This loop was chosen because it is small and simple enough to be used as an example, yet requires many sophisticated compilation techniques to expose ILP. The body of the loop consists of a simple *if-then-else* construct. In each iteration, it evaluates the condition $A[i] \neq 0$ to determine if the *then* part or the *else* part should be executed.

The low-level (assembly-level) code [2] for the loop, after traditional optimization [8], is presented in Figure 1(b). The descriptions of the register contents are represented using C-language syntax. For example, $\&B[1]$ is the address of the second element of array B. The horizontal lines partition the code into *basic blocks*. These are regions of straight-line code separated by decision points and merge points. Basic block L0 performs the initialization before the loop. L1 tests the *if* condition specified in the source code. L2 corresponds to the *then* part of the *if-then-else* construct and L3 to the *else* part. L4 tests for the loop condition and invokes the next iteration if the condition is satisfied.

Figure 1(c) shows an abstract representation of the control structure of the loop body called a *control flow graph*. Each node in the graph represents a basic block in the low-level code. There is an arc between two blocks if a control transfer between them is possible, either by taking a branch or by executing the next sequential instruction after the branch. The two possible paths through the loop body are clearly shown in this representation of the loop. The *then* path is enclosed by

[2] In the assembly language syntax used in this paper, the destination operand is listed first, followed by the source operands. For branches, the operands being compared are listed first, followed by the branch target.

dotted lines in Figure 1(c).

In Figure 1(b), register r2 is written by instruction 1 and read by instruction 2. Thus, instruction 2 is dependent on instruction 1 and cannot be executed until after instruction 1. This type of dependence, from a write of a register to a read of the same register, is called a *register flow dependence*. There is a different type of dependence between instruction 6 and instruction 9. Here, the value in r3 used by instruction 6 is overwritten by instruction 9. This is called a *register anti-dependence*. The dependences described above are between instances of the instructions in the same iteration of the loop, and are referred to as *loop-independent* dependences. In contrast, instruction 1 uses the value written into r3 by instruction 9 during the previous iteration of the loop. This register flow dependence is referred to as a *loop-carried* dependence because the dependence is carried by the loop from one iteration to another. Another loop-carried dependence exists between an instance of instruction 5 in the current iteration and the instance of the same instruction in the next iteration. The latter instance overwrites the value written into r6 by the earlier instance. This loop-carried dependence is called a *register output dependence*.

Dependences are used by the code scheduler to determine the constraints on the ordering of the instructions and by the optimizer to determine if a code transformation is valid. Figure 2(a) shows the register dependence graph for the loop body of Figure 1(b). Each node in the dependence graph represents an instruction. The arcs represent dependences between the nodes. An arc going from instruction A to instruction B indicates that instruction B is dependent on instruction A. For clarity, the loop-carried output and anti-dependences are not shown.

Similar types of dependences can also occur between instructions which access the same memory locations. In Figure 1(b), instruction 3 reads from the memory location at address $B+r3$ (i.e., $B[i]$) and instruction 6 writes to the same memory location. This is called a loop-independent *memory anti-dependence*. Instruction 4 reads the same memory location (i.e., $B[i+1]$) which instruction 6 overwrites in the next iteration, creating a loop-carried memory anti-dependence. *Memory flow* *dependences*, created by a store followed by a load from the same address, and *memory output dependences*, created by two stores to the same address, are also possible.

For memory accesses, it is also useful to know if two instructions read the same memory location. This is referred to as a *memory input dependence.* For example, in Figure 1(b), instruction 4 reads the same memory location which instruction 3 will read in the next iteration. Optimizations which remove redundant loads can make use of this information.

Figure 2(b) shows the memory dependence graph for the loop body of Figure 1(b). Only the memory reference instructions are shown. It may at first appear that a loop-independent memory output dependence exists between instructions 6 and 8. However, the two instructions are on separate paths through the loop body and cannot both be executed during the same iteration of the loop. Thus, control flow information as well as memory address information must be considered when computing dependences.

The column labeled *IT* (for issue time) in Figure 1(b) illustrates a possible execution scenario in which the *then* path is taken. The column shows the *issue time*, the time at which the instruction is sent to the functional unit.[3] The issue time for the block at label *L3* is not shown because it is not executed in this scenario.

2.2 Enlarging the Scope of ILP Optimization and Scheduling

Early attempts at ILP compilation used the basic block as the scope of the search for independent instructions. This approach is appealing because of its simplicity. During code scheduling, the compiler does not need to handle complications due to the decision and merge points. However, it

[3]The issue times assume a processor that issues instructions in the same order that that is specified in the program and that can begin execution of multiple instructions each cycle. There is no limit placed on the number or combination of instructions that can be issued in a single cycle. An instruction is issued as soon as the producers of its input operands have finished execution. It is also assumed that loads take 2 cycles, and that stores and ALU instructions take 1 cycle. The processor predicts the direction of branches, so instructions following the branch on the predicted path can be executed at the same time as the branch. Thus, the next loop iteration begins in the same cycle that the loop back branch is issued unless delayed by loop-carried dependences.

was discovered, and is the current consensus, that most basic blocks have very limited ILP [9][10]. In Figure 1(b), most of the basic blocks have very few instructions that can be executed in the same cycle. For the *then* path, the average number of instructions executed per cycle is only 1.5. To expose further ILP, the compiler must be able to search beyond the basic block boundaries.

Several approaches have been developed to form regions larger than the basic block. Only two of them are discussed here. They differ in their treatment of the decision point associated with instruction 2 in Figure 1(b). This instruction is highlighted both in Figure 1(b) and later examples to emphasize the difference between the two approaches.

One approach is to form either a trace [11] or a superblock [12], which is a region consisting of a sequence of basic blocks along a frequently executed path. This path is then optimized, sometimes at the expense of the infrequently executed paths. In the example of Figure 1(c), if the *else* path is infrequently executed, the dotted lines enclose a trace formed along the frequently executed *then* path. Figure 3(a) shows the control flow graph after a superblock (highlighted by dotted lines) is formed from the trace along the *then* path. A superblock is formed from a trace by duplicating blocks to remove merge points and their associated complications from the trace. In the example, basic block *L4* is duplicated.

Figure 3(b) shows the low-level code for the superblock after code scheduling. The code is laid out in memory so that the infrequent *else* path (not shown) does not interrupt the sequence of blocks in the superblock. The horizontal lines between basic blocks have been removed to emphasize the fact that the compiler can now optimize and schedule the code within this larger block. The process of reordering instructions within the superblock is called *global acyclic scheduling*. *Global* means that the compiler moves code across basic block boundaries. For example, the compiler has moved instructions 3 and 4, which were in block L2 of the original code, before instruction 2, which was in block L1 in the original code. *Acyclic* means that the scheduling is done within an acyclic region of the control flow graph. In the example, instructions have been moved within the superblock, but no instructions have been moved across the back-edge of the loop from one iteration to another.

Note that when the compiler moved instructions 3 and 4 to before instruction 2, code was moved from after to before a branch. This is known as *speculative code motion* [13][14][15][16]. In the original program, instructions 3 and 4 are executed only if the *then* path is executed and are said to be *control dependent* on instruction 2. Speculative code motion breaks this control dependence. When the *then* path is executed, two cycles are saved because the early execution of instructions 3 and 4 allows their dependent instructions (5 and 6) to also execute early. However, when the infrequent *else* path is executed, instructions 3 and 4 are executed needlessly. The compiler must ensure that the extra execution of instructions 3 and 4 does not cause incorrect program results when the *else* path is executed. The execution time on the *then* path is decreased from 6 cycles to 4 cycles as a result of the increased opportunities to execute instructions in parallel.

Traces and superblocks are most effective if a single frequently executed path through the loop body exists. The compiler can focus on optimizing that path without being limited by the constraints of the infrequent paths. Also, when there is a single dominant path, the branches in the loop body tend to be very predictable. If more than one important path exists, the compiler needs to jointly optimize both paths in order to produce efficient code. Also, when there is more than one important path, the branches can be less predictable, leading to more performance problems at run time.

If more than one important path exists, a larger region can be formed using *predicated execution* [17] [18]. This technique merges several paths into a single block and eliminates from the instruction stream the branches associated with those paths. Predicated or guarded execution refers to the conditional execution of an instruction based on the value of a boolean source operand, referred to as the *predicate*. Through a process known as *if-conversion*, the compiler converts conditional branches into predicate define instructions, and assigns predicates to instructions along alternative paths of each branch [19] [20] [21]. Predicated instructions are fetched regardless of their predicate value. Instructions whose predicate is true are executed normally. Conversely, instructions with false predicates are nullified, and thus are prevented from modifying the processor state. Transformation of conditional branches into predicate define instructions converts the control dependences associated with the branches into data dependences associated with the predicate define instructions. Predicated execution allows the compiler to trade instruction fetch efficiency for the capability to expose ILP to the hardware along multiple execution paths.

Figure 4 shows the example loop after both paths have been merged into a single block. Instead of branching based on a comparison, instruction 2 now does the same comparison and writes the result in predicate register p1. The *then* path is executed if predicate p1 is false; the *else* path is executed if predicate p1 is true. Initially, instructions 3-6 are all conditionally executed on predicate p1. However, a technique referred to as *predicate promotion* allows instructions 3, 4, and 5 from the *then* path to be executed before the result of the comparison is known [21] by removing the predicates from those instructions. This is another example of speculative code motion and again helps the performance of the *then* path.[4] The ILP along the *then* path has been increased to two instructions per cycle.[5] The *else* path is now included in the block and treated along with the *then* path during later optimizations instead of being excluded. The examples in the rest of the section will continue to optimize this predicated code.

In Figure 4, the compiler is limited to a single iteration of the loop when looking for independent instructions. *Loop unrolling* [22] and *software pipelining* [23][24][25] are two techniques which allow the compiler to overlap the execution of multiple iterations. A loop is unrolled by placing several copies of the loop body sequentially in memory. This forms a much larger block in much the same way as forming a superblock. A global acyclic scheduling algorithm can then be applied to the unrolled loop body. When code is moved between copies of the loop body, the iterations become

[4]Assuming that the *peq* instruction takes 1 cycle, executing instruction 5 before its predicate has been computed reduces the execution time of the loop body by 1 cycle.

[5]When the *then* path is executed, instructions 8 and 28 are not executed. Thus, they are not counted in the calculation of instructions/cycle.

overlapped.

However, with loop unrolling, only the copies of the original loop body within the unrolled loop body are overlapped; all overlap is lost when the loop-back branch for the unrolled loop body is taken. Software pipelining generates code that continuously maintains the overlap of the original loop iterations throughout the execution of the loop. Software pipelining is also called *cyclic scheduling* because it moves instructions across the back-edge in the control flow graph from one loop iteration to another.

2.3 Overcoming Register and Memory Dependences

Figure 5(a) shows the example loop from Figure 4 after unrolling to form two copies of the original loop body. Instruction 10, the loop-back branch in the original loop, is now a loop-exit branch which is taken if the second copy of the loop body should not be executed. There are now two problems which limit the overlap of the two copies of the loop body. The first is the reuse of the same registers for the second copy of the loop body. This reuse creates an anti-dependence because the instruction which writes the new value cannot be executed until all the instructions which read the old value have been executed. In addition, a register output dependence is created between the write of the old value and the write of the new value.

For the example loop, the anti-dependences associated with the index variable (r3) are particularly problematic. The code scheduler would like to arrange the instructions so that instructions from the second copy of the loop body execute concurrently with instructions from the first copy. However, the load instructions in the second copy depend on the update of the index variable done by instruction 9. Instruction 9 in turn cannot be executed until the stores which use r3 have been executed. These register dependences effectively serialize the two copies of the loop body, but can be removed by using different registers for the second copy of the loop body. This process is called compile-time *register renaming*.

The second impediment to ILP in the unrolled loop is possible memory dependences [26][27]. Given a pair of memory references, the compiler tries to determine whether or not the two memory references ever access the same memory location. If the compiler cannot determine this, it must conservatively assume that a dependence exists to guarantee the correctness of the compiled code.

In Figure 5(a), possible memory flow dependences exist from instructions 6 and 8, the highlighted stores of $B[i]$ in the first copy of the original loop body, to each of the highlighted load instructions 21 (i.e., $A[i]$), 23 (i.e., $B[i]$), and 24 (i.e., $B[i+1]$), in the second copy.[6] To determine if the loads are independent of the stores, the compiler must determine whether the loads ever access the same memory location as the stores. For example, if the compiler can determine that A and B are the labels for the start of two non-overlapping arrays, it can determine that the load of $A[i]$ cannot access the same memory location as the stores of $B[i]$. Determining that instructions 23 and 24 are independent of the stores is more difficult. The compiler must analyze the address operands to determine the relationship between the value in r1 and B and the relationship between the value in r3 in the first copy of the loop body and the value in r3 in the second copy. This process is referred to *data dependence analysis*, and will be discussed further in section 4.

It is possible for data dependence analysis techniques to determine that the loads in Figure 5(a) are independent of the stores, and Figure 5(b) shows the code after the compiler performs optimizations and scheduling assuming this independence. The register renaming discussed earlier has also been performed. The loads from the second copy can now be executed before the stores from the first copy. In addition, the compiler is able to determine, using data dependence analysis, that the load of $B[i]$ (i.e., instruction 23) in the second copy accesses the same location as the load of $B[i+1]$ (i.e., instruction 4) in the first copy; that is, a memory input dependence exists between the two instructions. Thus, the load of $B[i]$ is redundant, and can be removed. After applying data dependence analysis and register renaming, the unrolled loop body is executed in four cycles

[6]Note that the value of i increases by one from the first to the second copy.

instead of eight cycles.

2.4 Summary

Overall, ILP compilation reduced the execution time from 6 cycles for **one** iteration of the loop to 4 cycles for **two** iterations of the loop. Note that by using predication, the execution of the two copies of the loop body is overlapped even if the first copy takes the *else* path, which is an advantage of predication over traces or superblocks. Using superblocks, the code of Figure 3(b) would have been unrolled. However, in that case, if the *else* path in the first copy is taken, the unrolled superblock would be exited, and the overlap with the second copy would be lost. Using superblocks can cause significant performance degradation if the *else* path is executed often enough relative to the *then* path.

Also note that after the limitations imposed by control flow instructions and register dependences were removed, memory dependences became the only bottleneck. Data dependence analysis is important for traditional optimizations such as loop invariant code removal and redundant load elimination. However, as shown in this section, data dependence analysis is also important for ILP compilation because conservative memory dependences can prohibit the scheduler from overlapping independent sections of the code. The next two sections present case studies on the predication and data dependence analysis technologies, including their performance benefits.

3 Case Study 1: Compiling for Predicated Execution

A critical issue in high performance microprocessors is control flow. Control flow corresponds to constructs such as *switch* and *if-then-else* statements in programming languages. In a traditional instruction set architecture (ISA), control flow results in branch instructions that often account for a significant percentage of the instructions executed. For example, if one examines the execution record of the program *wc*, a frequently-used UNIX^{TM} utility program that counts the number of characters, words, and lines in a text file, more than 40% of the instructions executed by the processor are branch instructions.[7] The source code for the inner loop of *wc*, in which the program spends nearly 100% of its execution time, is shown in Figure 6.

A closer look at the *wc* inner loop provides some insight into the frequent occurrence of branch instructions. The assembly code for the loop is shown in Figure 7(a) and the corresponding control flow graph is shown in Figure 7(b). The weights on the arcs are the number of times each control flow arc is traversed when the program is executed with a particular input, and are obtained from run-time execution profiling. For each iteration of the loop, a *path* is effectively traversed through the control flow graph. One can verify that there are 22 alternative paths through this loop, some with much higher probability of traversal than the others. However, no one path through the inner loop is clearly dominant. For example, the most frequent path through the loop, $A \rightarrow B \rightarrow D \rightarrow E \rightarrow F \rightarrow H \rightarrow A$, is traversed 58% of the time. This path corresponds to the case where the current character is not the last character in the word; no special action is needed except for incrementing the character counter. Note that nine instructions are executed along this path and five of them are branches, resulting in a very high percentage of branch instructions among executed instructions.

In a processor that exploits instruction-level parallelism, branches limit ILP in two principal ways. First, branches force high performance processors to perform branch prediction at the cost of large performance penalty if prediction fails [28] [29] [30]. For example, if the execution takes path $A \rightarrow B \rightarrow D \rightarrow E \rightarrow F \rightarrow H \rightarrow A$, it will encounter five branch instructions. If the processor wants to fetch all nine instructions on this path in one cycle, it must be able to fetch instructions across five branch instructions in the same clock cycle. This means predicting all five instructions and hoping that the execution will follow that path. If any of the five branches is predicted incorrectly, the processor must disrupt the execution, recover from the incorrect prediction, and

[7] The frequency of branches varies across instruction set architectures and compilers. The example is based on HP PA-RISC^{TM} code generated by IMPACT C Compiler version 2.6.

initiate instruction fetch along the correct path. This costly disruption makes it necessary to predict branches as accurately as possible. Unfortunately, branches such as those at the end of E and H in Figure 7(b) do leave the path frequently in an unpredictable manner. This makes it difficult for any branch prediction scheme to attain a high degree of accuracy.

The second way branch instructions limit ILP comes from the limited number of processor resources available to handle branches. For example, most of the high performance microprocessors introduced in 1995 are capable of issuing four instructions every clock cycle, but only one of these four instructions can be a branch [6] [7]. Frequent occurrences of branches can seriously limit the execution speed of programs. For example, if the execution takes path $A \rightarrow B \rightarrow D \rightarrow E \rightarrow F \rightarrow H \rightarrow A$, a processor that executes one branch per clock cycle must spend at least five cycles to execute instructions along the path. With only nine instructions on the path, the processor is limited to executing 1.8 instructions per clock cycle.

One approach to removing the limitation due to branch handling hardware is to increase the number of branch instructions that can be handled in parallel during each clock cycle. This would require the processor to predict multiple branches each clock cycle. It would also require the processor to determine the joint outcome of multiple branches executed in the same cycle. The problem with this brute force approach is that it may negatively impact the clock frequency of the processor. An alternative technique allows the compiler to judiciously remove branches. Eliminating branches helps to solve both of the above problems, and removes the possibility of incorrect branch prediction associated with the branches. This can greatly reduce the performance penalty due to incorrect predictions. Furthermore, the reduced frequency of branches will increase the performance of processors with limited branch handling resources.

3.1 Predicate-Based Compilation

A suite of compiler techniques to effectively use predicated execution has been developed based on the hyperblock structure [21]. A hyperblock is a collection of connected basic blocks in which control may only enter at the first block, designated as the entry block. Control flow may leave from one or more blocks in the hyperblock. All control flow between basic blocks in a hyperblock is removed by eliminating branch instructions and introducing conditional instructions through if-conversion [19] [20].

While forming hyperblocks, the compiler must trade off branch performance with resource usage, hazard conditions, and critical path length. Selecting blocks on either side of a branch instruction eliminates the branch and any associated misprediction penalties. However, combining both paths may over-utilize available processor resources, introduce hazard conditions that inhibit optimizations, or increase the critical path length. The compiler must take all of these factors into account when determining which basic blocks are desirable to include within a hyperblock. In Figure 7(b), a hyperblock highlighted by the dashed box is formed by including the desirable blocks. For example, Block C is not included because of its low execution frequency and the fact that it calls the *fillbuf* function in the I/O library. Such library calls present hazard conditions that can reduce the effectiveness of parallelization. Blocks D through M are duplicated so that the control flow does not reenter the middle of the hyperblock. This is done to eliminate the effect the hazard conditions in C can have on the parallelization of the resulting hyperblock.

The Hewlett-Packard Laboratories PlayDoh architecture [31] helps us illustrate the compiler extensions required to support predicated execution. Within the PlayDoh architecture, all instructions have an operand that serves as a predicate specifier, that is, all instructions are predicated. The predicates are located in a centralized predicate register file. The ISA contains a set of specialized predicate define instructions that write their results into the predicate register file. The instruction format of a predicate define instruction is shown below.

$$\mathbf{p}\text{<}cmp\text{>}\ \mathbf{P}_{out1}(<type>),\ \mathbf{P}_{out2}(<type>),\ \mathbf{src1},\ \mathbf{src2}\ (\mathbf{P}_{in})$$

This instruction assigns values to P_{out1} and P_{out2} according to a comparison of *src1* and *src2* specified by <*cmp*>. The comparison <*cmp*> can be: equal (eq), not equal (ne), greater than (gt), etc. A predicate <*type*> is specified for each of the destination predicates P_{out1} and P_{out2}. Predicate define instructions may also be predicated, as specified by P_{in}.

The assembly code for the resulting hyperblock is shown in Figure 7(c). Notice that all branches except for instructions 2 and 5 have been replaced by predicate define instructions with corresponding compare conditions and input operands. All non-branch instructions are predicated with the predicate assigned to their basic block during if-conversion. The predicate define instructions in this hyperblock utilize two of the predicate definition types available in the PlayDoh ISA, unconditional (U) and OR-types. Unconditional-type predicates are useful for instructions that execute based upon a single condition. For example, instruction 17 from basic block I can only be reached from basic block G. Thus, the assigned predicate, p7, is defined by instruction 13 as unconditional. OR-type predicates are useful when execution of a block can be enabled by multiple conditions. An example use of an OR type predicate is provided by instruction 15, which is predicated on predicate p6. This instruction was originally located in block M and is executed only if control passes through blocks G, J, or L. OR-type predicates must be preset to zero and are set to one only if the result of an OR-type comparison evaluates to true. If an OR-type comparison evaluates to false, the destination predicate register is unchanged. This characteristic allows multiple OR-type definitions of the same predicate register to be executed in the same cycle. Thus, predicate p6 will allow instruction 15 to execute, if set to one by any of the predicate define instructions 13', 14, or 19.

The elimination of most branches within the inner loop of *wc* has two significant effects on the dynamic branch characteristics of the program. First, the number of dynamic branches in the program is reduced from 572K to 315K by if-conversion. Second, the number of branch mispredictions is significantly reduced; the two remaining exit branches are only taken a combined total of 15 times, making them easily predictable. The loop-back branch is also easy to predict since the loop iterates frequently. As a result, the number of branch mispredictions in *wc* using a branch target buffer with a 2-bit counter is reduced from 52K to 56. This example illustrates the ability of predicated execution to significantly improve the dynamic branch behavior of programs for architectures with high branch misprediction penalties and limited branch handling resources.

3.2 Predicate-Based Optimizations

Predicated execution provides benefits beyond improved program branch behavior. Hyperblock formation combines basic blocks from multiple control flow paths into a single block for optimization and scheduling. The presence of multiple control flow paths within a hyperblock exposes more opportunities for the compiler to apply classical optimizations, such as common subexpression elimination and copy propagation [8]. Hyperblock formation also increases the applicability of more aggressive ILP techniques by transforming complex control flow constructs into constructs that are better understood. For example, if-conversion transformed the complex control flow found in the *wc* example of Figure 7 into a single-block loop which greatly facilitates loop optimizations.

The hyperblock loop formed within *wc* has two desirable properties. First, it is a frequently iterated loop, making it a prime candidate for loop unrolling or software pipelining. Second, the hyperblock contains only two exit branches so execution will not be constrained on a processor with limited branch resources. However, not all loops are as well-behaved. Several additional predicate-based compiler optimizations have been developed to deal with these less well-behaved code segments.

Branch Combining. A hyperblock may contain branches which exit the hyperblock to handle execution sequences involving one or more basic blocks which were not selected for inclusion in the hyperblock. These basic blocks typically correspond to handling infrequent execution scenarios such as special cases, boundary conditions, and invalid input. In the *wc* example presented earlier (see Figure 7), the hyperblock contained two exit branches, instructions 2 and 5. These exit branches handle the special cases of refilling the input buffer and detecting the end of the input file. In many cases, code segments contain a large number of these infrequent execution scenarios. Thus, the corresponding hyperblocks often contain a large number of exit branches.

An example of such a hyperblock is the loop segment from the benchmark *grep* shown in Figure 8(a). The code segment consists of a loop body which has been unrolled twice. Each iteration of the loop consists of a load, a store, and four exit branches. This loop contains no loop-carried dependences, so the amount of parallelism achieved will be determined by resource constraints. In this example, the unrolled loop contains nine branches. Assuming the processor can execute one branch per cycle, the minimal schedule length of this hyperblock is 9/1 or nine cycles. With only 15 instructions in the loop body, the best case schedule length of nine cycles yields an average of only 1.67 instructions per cycle.

In this example, if-conversion alone was not sufficient for eliminating branches from the code. The targets of the remaining branches were not included in the hyperblock because the contents of those blocks would have resulted in a less efficient hyperblock. For these cases, the compiler can employ a transformation, referred to as branch combining [32], to further eliminate exit branches from the hyperblock. Branch combining replaces a group of exit branches with a corresponding group of predicate define instructions. All of the predicate defines write into the same predicate register using the OR-type semantics. As a result, the resultant predicate will be true if any of the exit branches were to be taken. Not exiting the hyperblock is the most common case, so the predicate will usually be false. The use of OR-type semantics allows the predicate define instructions to be executed in parallel, which significantly reduces the dependence height of the loop.

Branch combining is illustrated in Figure 8(b). Each of the exit branches, instructions 1, 3, 4, 5, 7, 9, 10, and 11 in Figure 8(a), is replaced by a corresponding predicate define instruction in Figure 8(b) based on the corresponding comparison condition. All predicate define instructions target the same predicate register, p1. The predicate is initially cleared, then each predicate define instruction sets p1 if the corresponding exit branch would have been taken in the original code. A single, combined exit branch (instruction 16) is then inserted which is taken whenever any of the exit branches are taken. The correct exiting condition is achieved by creating an unconditional branch predicated on p1. In the cases where p1 is false, the remainder of the unrolled loop is executed and the next iteration is invoked.

In the cases where an exit branch was indeed taken, instruction 16 transfers control to the block labeled Decode. This block of code serves two purposes. First, exit branches are re-executed in their original order to determine the first branch which was taken. Since the conditions of multiple exit branches could be true, the first such branch needs to be determined since that branch would have been taken in the original code sequence. The second purpose of the decode block is to execute any instructions in the hyperblock that originally resided between exit branches but have not yet been executed when the exit branch is taken. Memory stores are the most common instructions in this category. For example, instruction 6 (Figure 8(a)) is moved below the combined exit in the transformed hyperblock and copied to the decode block after the fourth exit branch. By positioning the store as such, it is guaranteed to execute exactly the same number of times as it did in the original code sequence. For instructions which may be executed speculatively, such as loads or arithmetic instructions, this duplication is unnecessary.

Overall, for an 8-issue processor which can execute at most one branch per cycle, the execution time of the example loop from *grep* is reduced from 584K cycles to 106K cycles with branch combining.

Additional Predicate-Based Optimizations. Two other important optimizations which take advantage of predicated execution support are loop peeling and control height reduction. Loop peeling targets inner loops that tend to iterate infrequently [32]. For these loops, loop unrolling and software pipelining are usually ineffective for exposing sufficient ILP since the number of iterations available to overlap is small. Loop peeling is a technique whereby the compiler "peels" away the first several iterations of a loop. The peeled iterations are then combined with the code surrounding the inner loop (oftentimes an outer loop) using predication to create a single block of code for ILP optimization and scheduling. By combining the inner loop code with the surrounding code, the scheduler can then overlap the execution of the peeled iterations with the surrounding code. Furthermore, when the surrounding code is an outer loop, the combined outer loop and peeled iterations of the inner loop may be unrolled or software pipelined to expose large levels of ILP.

Control height reduction, is a technique which reduces the dependence chain length to compute the execution conditions of instructions [33]. In the control flow domain, an execution path is a sequence of directions taken by the branches leading to a particular instruction. In the predicate domain, the execution path is a sequence of predicate values used to compute the predicate of a particular instruction. Since predicates are a series of data values computed using arithmetic instructions, data height reduction techniques such as symmetric back substitution may be applied. Using height reduction, a compiler can significantly shorten dependence chain lengths to compute predicates, thereby enhancing ILP. A complete description of these and other predicate optimizations are beyond the scope of this paper. The interested reader is referred to [33] and [32] for more details.

3.3 Experimental Evaluation

To illustrate the effect of full predication on processor performance, experiments were conducted on a set of ten *C* benchmark programs, including programs from SPEC CFP92 (052.alvinn, 056.ear) SPEC CINT92 (008.espresso, 023.eqntott, 072.sc), and common UNIX^{TM} utilities (cmp, eqn, grep, lex, wc). The processor model used for these experiments is an 8-issue extension of the HP PA-RISC^{TM} architecture. The processor may issue eight instructions of any type each cycle except branches, which are restricted to one per cycle. The processor has 64 integer, 64 floating-point, and 64 predicate registers. Branch prediction is done using a 2-bit counter based branch target buffer, which is direct-mapped and contains 1K entries. In addition, perfect instruction and data caches are assumed. The results are based on detailed simulation of dynamic execution of the benchmark programs.

Figure 9 compares the performance of these programs with and without support for predicated execution on a 8-issue processor. The base configuration for this experiment is a single-issue processor without predicate support. The *Best Non-Predicated* bars represent the speedup over the base processor of the most aggressive ILP compilation techniques for an architecture without support for predicated execution. The *Predicated* bars represent the performance of the predicate-based compilation techniques on an architecture with full predicate support. As Figure 9 shows, the addition of predicate support affords a significant improvement in performance.

The ability to remove branches by combining multiple execution paths is extremely beneficial to several of the benchmarks. The performance improvements in *023.eqntott*, *cmp*, and *wc* result from the elimination of almost all of the branch mispredictions. The largest overall speedup, 12.5, is observed for *cmp*. The apparent super-linear speedup for *cmp* is a combination of optimizations exposed through hyperblock formation and the increased issue width of the processor. The branch combining technique presented in Section 3.2 is the source of the performance improvements seen in *grep* and *lex*. The performance improvements in *008.espresso* and *072.sc* are primarily due to loop peeling.

Predicated execution support has several effects on the benchmarks that are not apparent from the data shown in Figure 9. The number of dynamic instructions that must be fetched by the processor may change because multiple execution paths are being combined. Also, the elimination of difficult-to-predict branches improves the performance of the branch prediction mechanism used by the processor. A detailed discussion of these effects is beyond the scope of this paper; the interested reader is referred to [34][35][36] for more details.

4 Case Study II: Memory Dependence Information for ILP Compilation

Various techniques have been proposed to provide an accurate analysis of memory references. Data dependence analysis attempts to determine the nature of the dependence relationship between pairs of memory references at compile time. This information can then be used to safely direct subsequent code transformations. This case study is organized as follows: The first section gives an overview of data dependence analysis. Later sections discuss the dependence information needed by the optimizing and scheduling phases of the compiler, and how that information can be maintained during code transformations. Finally, results are shown which quantify the benefit of accurate data dependence analysis for ILP processor performance.

4.1 Data Dependence Analysis

Traditionally, data dependence analysis is performed on the source-level code, and is used to facilitate source-to-source code transformations. In-depth dependence analysis has seldom been applied to assist compilation of low-level code. In most commercial compilers, rudimentary data dependence analysis for low-level code is performed using only symbol table information and inexpensive memory address analysis. For example, commercial compilers can often determine the independence of references to separate global variables or to separate locations on the stack. However, they often have trouble analyzing array and pointer references. There are two exceptions that the authors are aware of. The compiler for the Cydrome Cydra-5 [37] performed detailed memory dependence analysis for inner loops and used that information to support optimization of memory reference instructions and software pipelining. The Multiflow Trace Scheduling Compiler [22] also performed detailed memory dependence analysis for low-level code, but within traces instead of inner loops, and used that information to support optimization and scheduling for the TRACE series of VLIW computers.

Research has focused on data dependence analysis to support source-level transformations; however, many of the dependence algorithms proposed can also support ILP compilation. A few representative works are cited here. Numerous algorithms have been proposed for dependence analysis of array references [38][39][40]. These algorithms generate a set of equations and inequalities which represent the conditions that must be met for two references to access the same array element. A dependence exists between the two references if an integer solution to the equations exists. The problem of *variable aliasing*, which occurs when two or more logical variables can reference the same memory location, has also been addressed [41][42]. For languages like C, analysis across procedure boundaries, referred to as *inter-procedural analysis*, is often required to identify the aliases due to the use of pointers [43][44][27].

Sophisticated algorithms for array dependence analysis and inter-procedural analysis are time consuming. Thus, it is desirable to do the analysis only once. To make the most use of the information, it is desirable to do the analysis as early in the compilation process as possible. The information must then be maintained through subsequent code transformations.[8] Figure 10 gives an overview of this process. Dependence analysis is performed using the compiler's high-level intermediate representation (IR), which typically represents the program source code in a tree-type structure. The results of this dependence analysis, dependence relations between memory references, are maintained throughout subsequent compilation in the form of explicit dependence arcs. While compiling the low-level IR, the dependence arcs must be accurately maintained through any code transformations. The dependence arcs can then be used to provide accurate memory dependence information to support optimization and scheduling.

[8]The alternative to this approach is to maintain the source-level information necessary to do the analysis, and use this information to perform the analysis on the low-level code. This alternative approach was used in the Cydra-5 and Multiflow compilers.

4.2 Desired Dependence Information

Source-level dependence analysis has most often been used to support source-level loop transformations. In contrast, the approach outlined in Figure 10 applies the results of the source-level analysis to support low-level transformations. Because the dependence information must be maintained explicitly throughout the remaining compilation, only dependence information which is directly useful for supporting subsequent transformations should be propagated to the low-level code. In this section, examples involving acyclic scheduling, cyclic scheduling, and code optimization show the types of dependence information utilized in the low-level compilation phases.

4.2.1 Dependence Information to Support Code Scheduling

For acyclic scheduling, the legality of reordering two memory instructions is based on whether a dependence exists between the two instructions during any single execution of the block being scheduled. If the block is a loop body, two memory instructions can be reordered only if they never reference the same memory location during any single iteration of the loop; i.e., there exists no loop-independent dependence between them. For example, the array references for the code shown in Figure 11(a) cannot be legally reordered within a single loop iteration because they access the same element of the array A during each iteration of the loop.

Since cyclic scheduling attempts to overlap the execution of multiple, consecutive iterations of the loop body, loop-carried dependences must also be taken into account. For the code shown in Figure 11(b), the store of $A[i+2]$ will reference the same location as the load of $A[i]$ which occurs two loop iterations later. The number of loop iterations between two dependent memory references is called the *dependence distance*.[9] To continue the example, a flow dependence of distance two exists from the store to the load for the array A. During cyclic scheduling, this distance-two

[9] Loop-independent dependences, such as the one shown in Figure 11(a) are sometimes called *distance-zero* dependences since both references occur in the same loop iteration.

dependence prevents scheduling the load of $A[i]$ from two iterations later before the store of $A[i+2]$ in the current iteration. Note that the load and store of array A could be reordered during acyclic scheduling of the loop body for this code, because no loop-independent dependence exists.

A characteristic of dependence relations useful for cyclic scheduling is the loop which *carries* the dependence. In Figure 11(c), note that the inner-loop index variable is i and the array references are indexed by only the outer-loop index variable, j. During each single iteration of the j loop, the load and store instructions reference the same two non-intersecting locations for all iterations of the i loop. Thus, an output dependence of distance one, carried by the i loop, exists between the store in the current iteration of the i loop and the store in the next iteration of the i loop (but the same iteration of the j loop), as shown in Figure 11(c).[10] This distance-one output dependence prevents scheduling the store of $A[j+2]$ from the next iteration of the i loop before the store of $A[j+2]$ in the current iteration. In addition, a flow dependence of distance two, carried by the j loop, exists between the store and the load, which prevents scheduling the store of $A[j+2]$ in the current iteration of the j loop before the load of $A[j]$ two iterations later. Although both dependences are loop carried, only the distance-one output dependence must be honored during cyclic scheduling of the i loop, and only the distance-two flow dependence must be honored during cyclic scheduling of the j loop. Because cyclic scheduling of a particular loop need only honor dependences which are carried by that loop or are loop-independent, the loop which carries the dependence is useful information to include in the dependence representation.

From the requirements of acyclic and cyclic scheduling, it follows that the dependence information should include the dependence distance and an identifier indicating which loop carries the dependence (or none if the dependence is loop-independent). For some cases, the dependence analysis may determine that the dependence distance varies, or it may be unable to determine the distance. For these cases, the dependence representation should at least be able to specify that the

[10] Although an input dependence also exists from the load to the load in the next iteration of the i loop, this dependence is not shown in Figure 11(c) because input dependences do not affect code scheduling.

dependence distance is unknown, implying that dependences of any distance may exist.

4.2.2 Dependence Information to Support Optimizations

Memory dependence analysis is critical not only for code scheduling, but also for code optimization. The *certainty* of the existence of the dependence is a characteristic of the dependence relationship useful to support code optimization. One optimization requiring information about the certainty of the dependence is redundant load elimination, illustrated in Figure 12. This optimization attempts to eliminate the second load if it is loading the same data as the first load during each loop iteration. Thus, a *definite* loop-independent dependence must exist between the two loads for all iterations of the i loop.

Another condition for valid redundant load elimination is that no intervening store instructions may exist that possibly reference the same address as the load instructions. Even a possible dependence, called a *maybe* dependence, between the store and either load is sufficient to prevent the optimization. For example, if the dependence analyzer cannot determine that the memory locations accessed by $A[i]$ and $A[j]$ are different within each iteration of the i loop in Figure 12, then it must report that maybe dependences exist from the first load to the store and the store to the second load, which prevent the optimization. Information regarding the certainty of the dependence is useful to include in the dependence representation to support this optimization. Note that although only flow, output, and anti-dependence relations are useful for code scheduling, this example demonstrates that input dependence relations are useful for optimization as well.

Table 1 summarizes the dependence information which is useful for scheduling and optimization of the low-level code.

4.3 Maintaining Dependence Arcs

One limitation of using source-level analysis results for low-level code compilation is that dependence analysis cannot be re-applied if the dependence information is lost or becomes less accurate as a result of code transformations. For this approach to be viable, the dependence arcs must be accurately maintained through transformations.

Maintaining dependence arcs when memory instructions are relocated within a loop body is trivial. If two memory instructions are legally reordered by a code transformation such as acyclic scheduling, no characteristic of the dependence is altered. For example, if a loop-carried flow dependence exists from a store instruction to a subsequent load, and scheduling reorders the instructions, the dependence will still be a flow dependence with the same dependence distance.

The maintenance of the arcs is less trivial when the structure of a loop is changed. One of the most interesting optimizations with regard to dependence arc maintenance is loop unrolling. To preserve the accuracy of the dependences, the optimizer must utilize the dependence distance when unrolling loops. If the compiler does not utilize the dependence distance, it must conservatively assume that all dependences carried by the loop could have any distance greater than or equal to one. Consequently, if a loop-carried dependence exists between a store and a load in the original loop, then the compiler places a loop-carried dependence between all possible combinations of the copies of that store and load in the unrolled loop. The resulting loss of certainty in the dependence relationships restricts code scheduling and can significantly decrease available ILP. A simple method for maintaining dependence arcs with full precision during unrolling is demonstrated in Figure 13.

Figure 13(a) shows a loop body with dependence arcs annotated by the dependence distance in loop iterations. For example, a flow dependence arc exists between the store and the first load with a distance of 2. Figure 13(b) shows the loop body after unrolling, with updated dependence arcs. After unrolling, there will be a flow dependence arc from the store in copy 0 of the unrolled loop, but the compiler must determine to which loads (the first load in copy 0, the first load in copy 1, or both) the arc should go. The copy of the loop to which the arc should go ($Copy_{dest}$) can be identified as follows:

$$Copy_{dest} = (Copy_{src} + Dist_{old}) \bmod n$$

where $Copy_{src}$ is the copy of the loop containing the arc's source instruction, $Dist_{old}$ is the distance in original loop iterations, and n is the number of copies of the original loop body after unrolling (two for the example in Figure 13). The example flow dependence arc would go to the first load in copy $(0 + 2) \bmod 2 = 0$. In addition, the dependence distance is updated so that it represents the difference in iterations of the unrolled loop. The new distance is calculated as follows using integer division instead of modulo arithmetic on the same values:

$$Dist_{new} = \frac{Copy_{src} + Dist_{old}}{n}$$

The example flow dependence would have a new distance of $(0 + 2)/2 = 1$. Use of the formulas to update all the arcs in the unrolled loop body results in a precise update of the dependence relations, adding only the required arcs.

4.4 Experimental Evaluation

A suite of 29 benchmarks was compiled and subjected to aggressive ILP optimizations. The benchmark suite consists of 15 integer programs and 14 floating-point programs. Each benchmark was compiled in two different ways: once using a base level of dependence analysis and once using aggressive memory dependence analysis. The base-level dependence analysis uses only symbol table information and inexpensive memory address analysis, and thus cannot determine the independence of many pointer and array references. In contrast, the aggressive memory dependence analysis includes both the sophisticated array dependence analysis performed by the Omega Test [45], as well as inter-procedural analysis of pointer aliases and function side effects. For this case, memory dependence information is propagated to the low-level code in the form of dependence arcs.

Both versions of the compiled code were then evaluated using a detailed simulation of the processor and memory system. For more details of the experimental evaluation presented here, see [27]. An experimental evaluation of the Multiflow TRACE 14/300 computer, including the bottlenecks due to memory dependences, has been published elsewhere [46].

4.4.1 Integer Benchmark Results

Figure 14 shows performance results for the 15 integer benchmarks. The first six benchmarks in the figure are from the SPEC CINT92 suite and the rest are UNIX^{TM} utilities. The speedup of code compiled with and without memory dependence arcs is shown for an 8-issue architecture compared to a baseline single-issue architecture. The code compiled without memory dependence arcs and the baseline single-issue code use the base-level dependence analysis.

The data demonstrates that maintaining the dependence arcs derived from the aggressive memory dependence analysis resulted in much better optimization and scheduling than the information provided by the base-level dependence analysis. The use of memory dependence arcs resulted in more than 20% speedup over the same architecture without the arcs for six of the benchmarks, and even significantly greater speedup in a few cases. The use of memory dependence analysis significantly impacted overall ILP; for most benchmarks, the code compiled with the dependence arcs provided greater than three times speedup over the baseline single-issue architecture.

The most impressive result occurs for the benchmark *cmp*, which obtained a 17.8 times speedup over the single-issue processor and an 11.8 times speedup over the 8-issue processor without memory dependence arcs. This large speedup results because the compiler is able to both optimize and schedule the code much more effectively with the aid of dependence arcs.

4.4.2 Floating-point Benchmark Results

The benchmarks from SPEC CFP92 were also evaluated to measure the performance increase due to accurate memory dependence information. Floating-point programs generally display significantly different characteristics than integer programs. Floating-point programs are usually much less control intensive; they tend to have larger basic blocks than integer C programs. They also tend to spend most of the execution time in loops with a large number of iterations and make frequent use of array data structures. These factors tend to make ILP compilation for floating-point code more effective, assuming good memory dependence analysis is available for array references.

Figure 15 presents the speedup results for floating-point code compiled with and without memory dependence arcs on an 8-issue processor, relative to the single-issue processor. Because the effectiveness of the base-level dependence analysis is limited for array references, the results for the code without arcs are relatively poor. A speedup of less than two is achieved for most programs, despite the ability to issue eight instructions per cycle. With the benefit of aggressive memory dependence analysis, the code with arcs provides substantially better performance for most benchmarks. However, for a few benchmarks such as *013.spice2g6*, *015.doduc*, and *048.ora*, overall parallelism is low and the aggressive dependence analysis provides limited benefit. A comparison of the data presented in Figures 14 and 15 confirms that the benefit of the aggressive dependence analysis is more pronounced for floating-point code than for integer code.

5 Conclusions

In the coming decade, microprocessor designers will continue to increase the hardware parallelism, allowing execution of 16 or more instructions each clock cycle. In order to exploit the performance potential of such parallel hardware, compiler technology will assume greater responsibility in exposing and enhancing instruction-level parallelism in the executable code. The quality of compilation will become one of the most important distinguishing factors among microprocessor products. Therefore, one can expect that commercial compilers will go through rapid, fundamental changes to include technology comparable to that described in this paper.

We would like to emphasize that abundant research opportunities exist in the continuing revolution of compiler technology. Microprocessor architects will continue to devise innovative features such as predicated execution that require new compiler support. Compiler researchers will continue to develop innovations to meet the demands for increased performance. Because of these innovations, the phenomenal growth in processor performance should continue into the foreseeable future.

Acknowledgments

This paper and the underlying research have benefited from discussions with Mike Schlansker, Bob Rau, Vinod Kathail, and Sadun Anik. The authors would like to thank Sadun Anik, Scott Breach, Po-Yung Chang, Yale Patt, Guri Sohi, Aad van Moorsel, and Cliff Young for reviewing the various versions of the paper, and Brian Deitrich, Teresa Johnson and Jim McCormick for their contributions to the paper. We would also like to acknowledge all the members of the IMPACT research group for their support.

This research has been supported by the National Science Foundation (NSF) under grant MIP-9308013. Grant Haab was supported by a Fannie and John Hertz Foundation Graduate Fellowship. David August was supported by an Office of Naval Research (ONR) Graduate Fellowship.

References

[1] T. Asprey, G. Averill, E. DeLano, R. Mason, B. Weiner, and J. Yetter, "Performance features of the PA7100 microprocessor," *IEEE Micro*, vol. 13, pp. 22–35, June 1993.

[2] E. McLellan, "The Alpha AXP architecture and 21064 processor," *IEEE Micro*, vol. 13, pp. 36–47, June 1993.

[3] J. Emer and D. Clark, "A characterization of processor performance in the VAX-11/780," in *Proceedings of the 11th International Symposium on Computer Architecture*, June 1984.

[4] J. H. Crawford, "The i486 CPU: Executing instructions in one clock cycle," *IEEE Micro*, pp. 27–36, February 1990.

[5] M. Forsyth, S. Mangelsdorf, E. Delano, C. Gleason, and J. Yetter, "CMOS PA-RISC processor for a new family of workstations," in *Proceedings of COMPCON*, pp. 202–207, February 1991.

[6] J. H. Edmondson, P. Rubinfeld, R. Preston, and V. Rajagopalan, "Superscalar instruction execution in the 21164 Alpha microprocessor," *IEEE Micro*, pp. 33–43, April 1995.

[7] P. Wayner, "SPARC strikes back," *Byte*, vol. 19, pp. 105–112, Nov. 1994.

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

[9] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176–188, April 1991.

[10] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," in *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 46–57, May 1992.

[11] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, pp. 478–490, July 1981.

[12] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The

Superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[13] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 180–192, April 1987.

[14] M. D. Smith, M. A. Horowitz, and M. S. Lam, "Efficient superscalar performance through boosting," in *Proceedings of the Fifth International Conference on Architecture Support for Programming Languages and Operating Systems*, pp. 248–259, October 1992.

[15] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling: A model for compiler-controlled speculative execution," *Transactions on Computer Systems*, vol. 11, November 1993.

[16] R. A. Bringmann, S. A. Mahlke, R. E. Hank, J. C. Gyllenhaal, and W. W. Hwu, "Speculative execution exception recovery using write-back suppression," in *Proceedings of 26th Annual International Symposium on Microarchitecture*, December 1993.

[17] P. Y. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 386–395, June 1986.

[18] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, pp. 12–35, January 1989.

[19] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189, January 1983.

[20] J. C. Park and M. S. Schlansker, "On predicated execution," Tech. Rep. HPL-91-58, Hewlett Packard Laboratories, Palo Alto, CA, May 1991.

[21] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 45–54, December 1992.

[22] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donell, and J. C. Ruttenberg, "The Multiflow trace scheduling compiler," *The Journal of Supercomputing*, vol. 7, pp. 51–142, January 1993.

[23] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 63–74, December 1994.

[24] K. Ebcioglu and T. Nakatani, "A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture," in *Languages and Compilers for Parallel Computing*, pp. 213–229, 1989.

[25] P. Tirumalai, M. Lee, and M. Schlansker, "Parallelization of loops with exits on pipelined architectures," in *Proceedings of Supercomputing '90*, November 1990.

[26] W. Y. Chen, *Data Preload for Superscalar and VLIW Processors.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[27] D. M. Gallagher, *Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[28] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135–148, May 1981.

[29] J. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, pp. 6–22, January 1984.

[30] T. Y. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 257–266, May 1993.

[31] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: Version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA, February 1994.

[32] S. A. Mahlke, *Exploiting Instruction Level Parallelism in the Presence of Conditional Branches.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[33] M. Schlansker, V. Kathail, and S. Anik, "Height reduction of control recurrences for ILP processors," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 40–51, December 1994.

[34] G. S. Tyson, "The effects of predicated execution on branch prediction," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 196–206, December 1994.

[35] D. N. Pnevmatikatos and G. S. Sohi, "Guarded execution and branch prediction in dynamic ILP processors," in *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 120–129, April 1994.

[36] S. A. Mahlke, R. E. Hank, J. McCormick, D. I. August, and W. W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," in *Proceedings of the 22th International Symposium on Computer Architecture*, pp. 138–150, June 1995.

[37] J. C. Dehnert and R. A. Towle, "Compiling for the Cydra 5," *The Journal of Supercomputing*, vol. 7, pp. 181–227, January 1993.

[38] M. J. Wolfe, *Optimizing Compilers for Supercomputers.* PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1982.

[39] U. Banerjee, *Dependence Analysis for Supercomputing.* Boston, MA: Kluwer Academic Publishers, 1988.

[40] W. Pugh and D. Wonnacott, "Eliminating false data dependences using the omega test," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 140–151, June 1992.

[41] J. Banning, "An efficient way to find the side effects of procedure calls and the aliases of variables," in *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pp. 29–41, January 1979.

[42] K. Cooper, "Analyzing aliases of reference formal parameters," in *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pp. 281–290, January 1985.

[43] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural pointer aliasing," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 235–248, June 1992.

[44] E. Ruf, "Context-insensitive alias analysis reconsidered," in *Proceedings of the ACM SIGPLAN 95 Conference on Programming Language Design and Implementation*, pp. 13–22, June 1995.

[45] W. Pugh, "A practical algorithm for exact array dependence analysis," *Communications of the ACM*, vol. 35, pp. 102–114, August 1992.

[46] M. A. Schuette and J. P. Shen, "An instruction-level experimental evaluation of the Multiflow TRACE 14/300 VLIW computer," *The Journal of Supercomputing*, vol. 7, pp. 181–227, January 1993.

```
for (i=0; i<n; i++)
    if (A[i] != 0)
        B[i] = B[i] + B[i+1];
    else
        B[i] = 0;
```

(a)

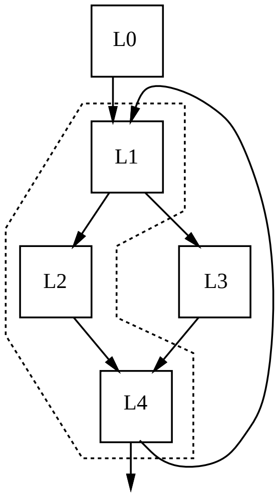
	Inst.	Assembly		IT
L0:		mov	r3, 0	
		mul	r7, n, 4	
		add	r1, B, 4	
L1:	1	ld	r2, mem(A+r3)	0
	2	**beq**	**r2, 0, L3**	2
L2:	3	ld	r4, mem(B+r3)	2
	4	ld	r5, mem(r1+r3)	2
	5	add	r6, r5, r4	4
	6	st	mem(B+r3), r6	5
	7	jmp	L4	5
L3:	8	st	mem(B+r3), 0	
L4:	9	add	r3, r3, 4	5
	10	blt	r3, r7, L1	6

1.50 instr./cycle

Register contents:

r1 = &B[1]
r3 = i*4
r4 = B[i]
r2 = A[i]
r5 = B[i+1]
r6 = B[i] + B[i+1]
r7 = n*4

(b)



(c)

Figure 1: Traditional compilation example, (a) original loop, (b) assembly code, (c) control flow graph.

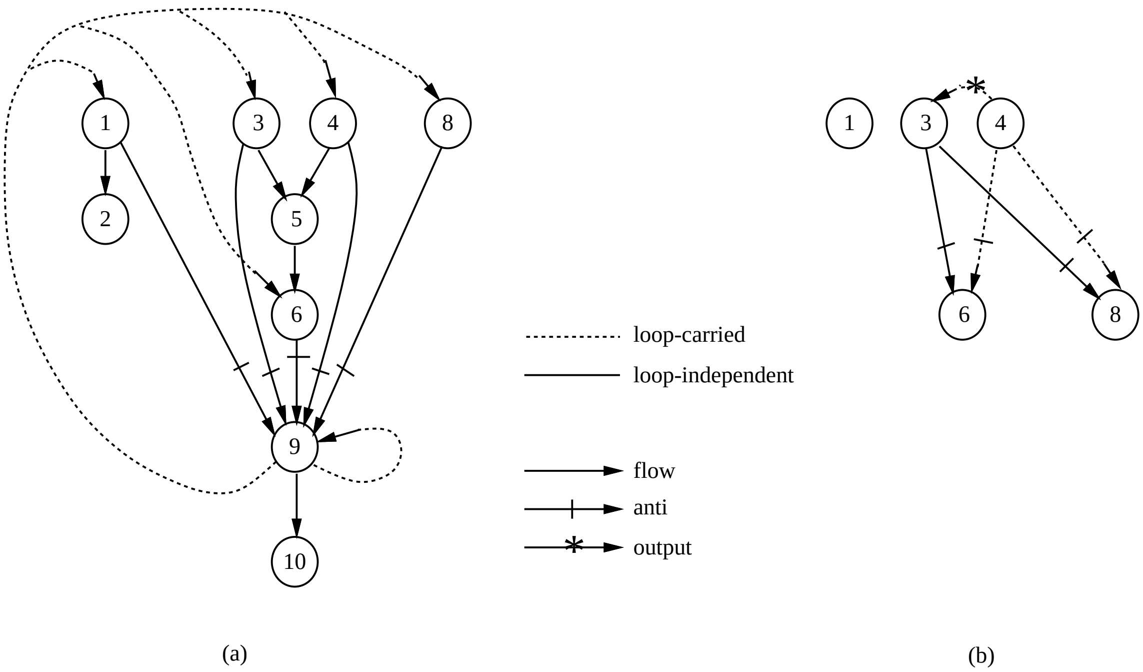
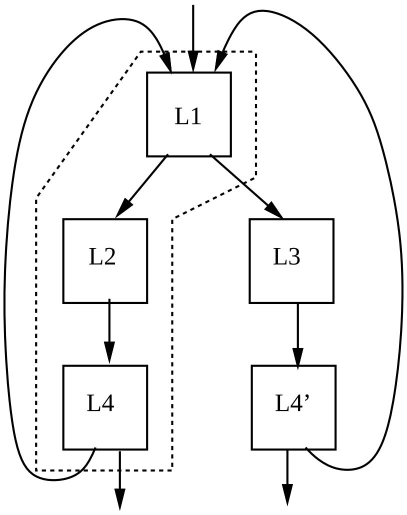


Figure 2: Dependence graphs for example loop, (a) register dependences, (b) memory dependences.



(a)

	Inst.	Assembly		IT
L1:	1	ld	r2, mem(A+r3)	0
	3	ld	r4, mem(B+r3)	0
	4	ld	r5, mem(r1+r3)	0
	2	**beq**	**r2, 0, L3**	2
	5	add	r6, r5, r4	2
	6	st	mem(B+r3), r6	3
	9	add	r3, r3, 4	3
	10	blt	r3, r7, L1	4

2.0 instr./cycle

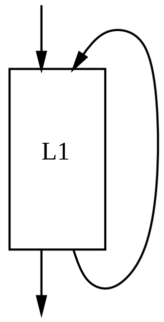
(b)

Figure 3: Forming larger blocks from traces, (a) control flow graph, (b) assembly code for the *then* path.

	Inst.	Assembly			IT
L1:	1	ld	r2, mem(A+r3)		0
	3	ld	r4, mem(B+r3)		0
	4	ld	r5, mem(r1+r3)		0
	5	add	r6, r5, r4		2
	2	**peq**	**p1, r2, 0**		2
	6	st	mem(B+r3), r6	($\overline{p1}$)	3
	8	st	mem(B+r3), 0	(p1)	3
	9	add	r3, r3, 4		3
	10	blt	r3, r7, L1		4

2.0 instr./cycle

(a)



(b)

Figure 4: Forming larger blocks using predication, (a) assembly code, (b) control flow graph.

	Inst.	Assembly			IT	
L1:	1	ld	r2, mem(A+r3)		0	
	3	ld	r4, mem(B+r3)		0	
	4	ld	r5, mem(r1+r3)		0	
	5	add	r6, r5, r4		2	
	2	peq	p1, r2, 0		2	Copy 0
	6	**st**	**mem(B+r3), r6**	**($\overline{p1}$)**	3	
	8	**st**	**mem(B+r3), 0**	**(p1)**	3	
	9	add	r3, r3, 4		3	
	10	bge	r3, r7, L100		4	
	21	**ld**	**r2, mem(A+r3)**		4	
	23	**ld**	**r4, mem(B+r3)**		4	
	24	**ld**	**r5, mem(r1+r3)**		4	
	25	add	r6, r5, r4		6	
	22	peq	p1, r2, 0		6	Copy 1
	26	st	mem(B+r3), r6	($\overline{p1}$)	7	
	28	st	mem(B+r3), 0	(p1)	7	
	29	add	r3, r3, 4		7	
	30	blt	r3, r7, L1		8	

2.0 instr./cycle

(a)

	Inst.	Assembly			IT
L1:	1	ld	r2, mem(A+r3)		0
	21	**ld**	**r9, mem(A+r8)**		0
	3	ld	r4, mem(B+r3)		0
	4	ld	r5, mem(r1+r3)		0
	24	**ld**	**r11, mem(r1+r8)**		0
	5	add	r6, r5, r4		2
	25	add	r12, r11, r5		2
	2	peq	p1, r2, 0		2
	22	peq	p2, r9, 0		2
	6	**st**	**mem(B+r3), r6**	**($\overline{p1}$)**	3
	26	**st**	**mem(B+r8), r12**	**($\overline{p2}$)**	3
	8	st	mem(B+r3), 0	(p1)	3
	28	st	mem(B+r8), 0	(p2)	3
	9	add	r3, r8, 4		3
	10	bge	r8, r7, L100		3
	29	add	r8, r3, 4		3
	30	blt	r3, r7, L1		4

3.75 instr./cycle

(b)

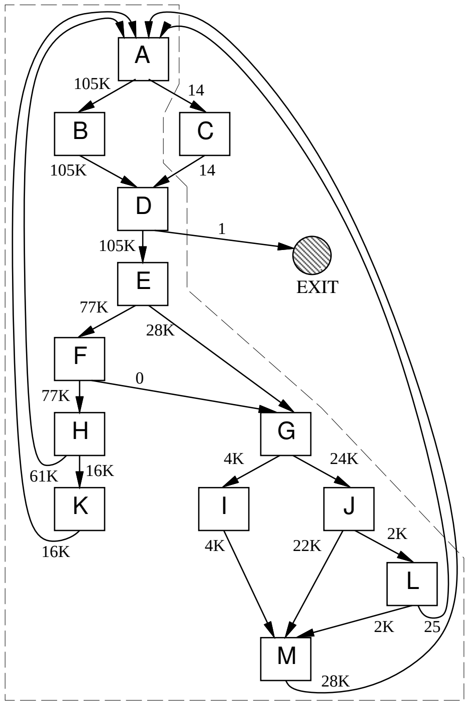
Figure 5: Effect of dependence analysis and dependence removal on performance, (a) after unrolling, (b) after renaming and memory dependence analysis.

```
        linect = wordct = charct = token = 0;
        for (;;)
        {
A:          if (−−(fp)−>cnt < 0)
C:              c = filbuf(fp);
            else
B:              c = *(fp)−>ptr++;
D:          if (c == EOF)   break;
E:          charct++;
            if ((' ' < c) &&
F:              (c < 0177))
            {
H:              if (! token)
                {
K:                  wordct++;
                    token++;
                }
                continue;
            }
G:          if (c == '\n')
I:              linect++;
J:          else if ((c != ' ') &&
L:                  (c != '\t'))   continue;
M:          token = 0;
        }
```

Figure 6: Source code for the inner loop of *wc*.

```
 1  A:  add   r6,r6,−1
 2      blt   r6,0,C
 3  B:  ld_c  r4,mem(r7+0)
 4      add   r7,r7,1
 5  D:  beq   r4,−1,EXIT
 6  E:  add   r1,r1,1
 7      bge   32,r4,G
 8  F:  bge   r4,127,G
 9  H:  bne   0,r8,A
10  K:  add   r2,r2,1
11      add   r8,r8,1
12      jmp   A
13  G:  beq   r4,10,I
14  J:  bne   r4,32,L
15  M:  mov   r8,0
16      jmp   A
17  I:  add   r3,r3,1
18      jmp   M
19  L:  bne   r4,9,A
20      jmp   M
21  C:  st    mem(r5+0),r6
22      st    mem(r5+4),r7
23      mov   Parm0,r5
24      jsr   filbuf
25      mov   r4,Ret0
26      ld    r6,mem(r5+0)
27      ld    r7,mem(r5+4)
28      jmp   D
```

(a)



(b)

```
 0   A:  pclr  p4,p6
 1       add   r6,r6,−1
 2       blt   r6,0,C
 3       ld_c  r4,mem(r7 + 0)
 4       add   r7,r7,1
 5       beq   r4,−1,EXIT
 6       add   r1,r1,1
 7       pge   p4(OR),p1(U̅),32,r4
 8       pge   p4(OR),p2(U̅),r4,127   (p1)
 9       pne   p3(U̅),−,0,r8          (p2)
10       add   r2,r2,1               (p3)
11       add   r8,r8,1               (p3)
13       peq   p7(U),−,r4,10         (p4)
17       add   r3,r3,1               (p7)
13’      peq   p6(OR),p5(U̅),r4,10    (p4)
14       pne   p6(O̅R̅),p8(U),r4,32    (p5)
19       peq   p6(OR),−,r4,9         (p8)
15       mov   r8,0                  (p6)
16       jmp   A
```

Register contents:		Predicate association:	
r1 = charct	r5 = fp	p1 ∼ F	p5 ∼ J
r2 = wordct	r6 = fp−>cnt	p2 ∼ H	p6 ∼ M
r3 = linect	r7 = fp−>ptr	p3 ∼ K	p7 ∼ I
r4 = c	r8 = token	p4 ∼ G	p8 ∼ L

(c)

Figure 7: Inner loop segment of *wc*, (a) assembly code, (b) control flow graph, (c) assembly code after if-conversion.

```
 1  A:  bge   r1, r5, EXIT1
 2      ld_c  r3, mem(r1−2)
 3      beq   r3, 10, EXIT2
 4      beq   r3, 0, EXIT3            Copy 0
 5      bge   r2, r6, EXIT4
 6      st_c  mem(r2−2), r3
 7      bge   r1, r7, EXIT5
 8      ld_c  r4, mem(r1−1)
 9      beq   r4, 10, EXIT6
10      beq   r4, 0, EXIT7            Copy 1
11      bge   r2, r8, EXIT8
12      st_c  mem(r2−1), r4
13      add   r1, r1, 2
14      add   r2, r2, 2
15      jmp   A
```

(a)

```
 0   A:  pclr  p1
 1’      pge   p1(OR), r1, r5
 2       ld_c  r3, mem(r1−2)
 3’      peq   p1(OR), r3, 10          Copy 0
 4’      peq   p1(OR), r3, 0
 5’      pge   p1(OR), r2, r6
 7’      pge   p1(OR), r1, r7
 8       ld_c  r4, mem(r1−1)
 9’      peq   p1(OR), r4, 10          Copy 1
10’      peq   p1(OR), r4, 0
11’      pge   p1(OR), r2, r8
16       jmp   Decode        (p1)
 6’      st_c  mem(r2−2), r3
12       st_c  mem(r2−1), r4
13       add   r1, r1, 2
14       add   r2, r2, 2
15       jmp   A

     Decode:
 1       bge   r1, r5, EXIT1
 3       beq   r3, 10, EXIT2
 4       beq   r3, 0, EXIT3
 5       bge   r2, r6, EXIT4
 6       st_c  mem(r2−2), r3
 7       bge   r1, r7, EXIT5
 9       beq   r4, 10, EXIT6
10       beq   r4, 0, EXIT7
11       jmp   EXIT8
```

(b)

Figure 8: Example of branch combining from *grep*, (a) original assembly code, (b) assembly code after branch combining.

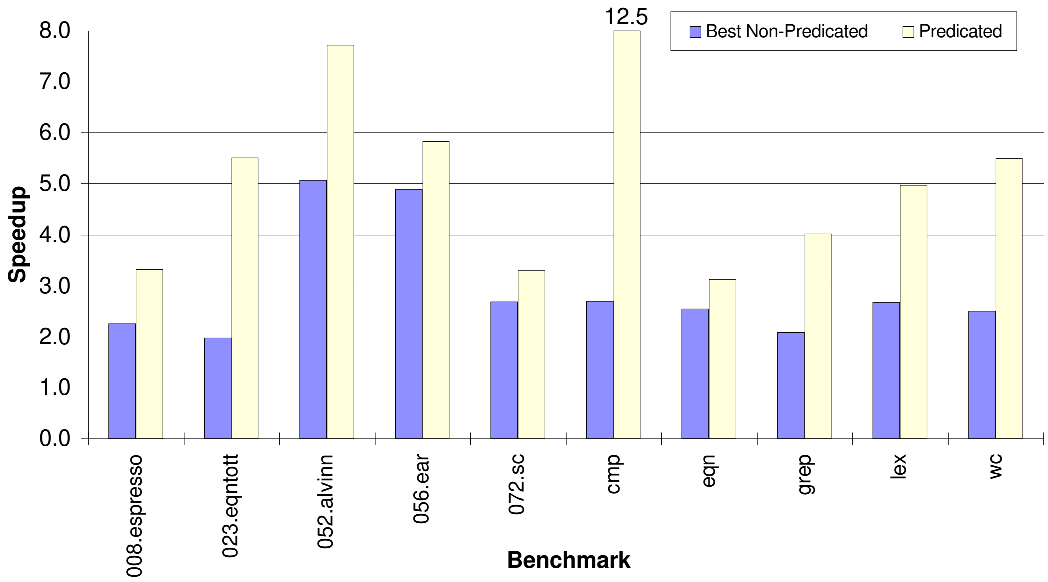


Figure 9: Performance improvement of predicated execution support for an 8-issue processor.

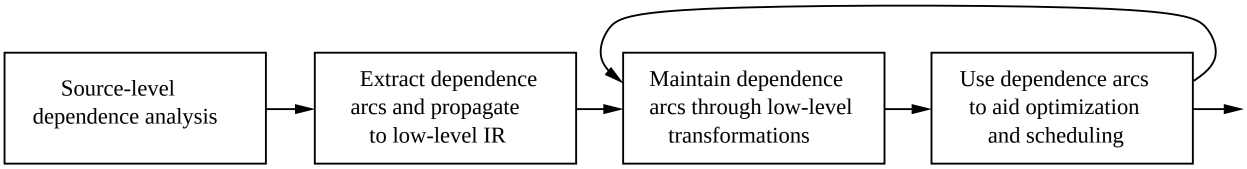


Figure 10: Overview of the generation and maintenance of dependence information.

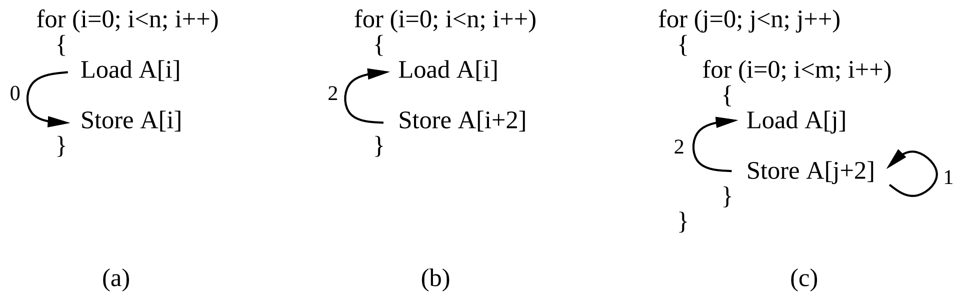


Figure 11: Code scheduling examples, (a) loop-independent dependence, (b) loop-carried dependence (c) outer-loop-carried dependence.

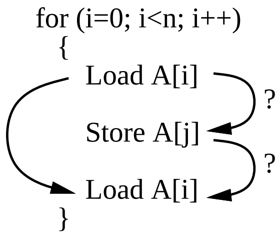


Figure 12: Redundant load elimination example.

Table 1: Desired dependence information.

Category	Possible Values
type	flow, anti, output, input
distance	(integer), unknown
carrying loop	none, (loop identifier)
certainty	definite, maybe

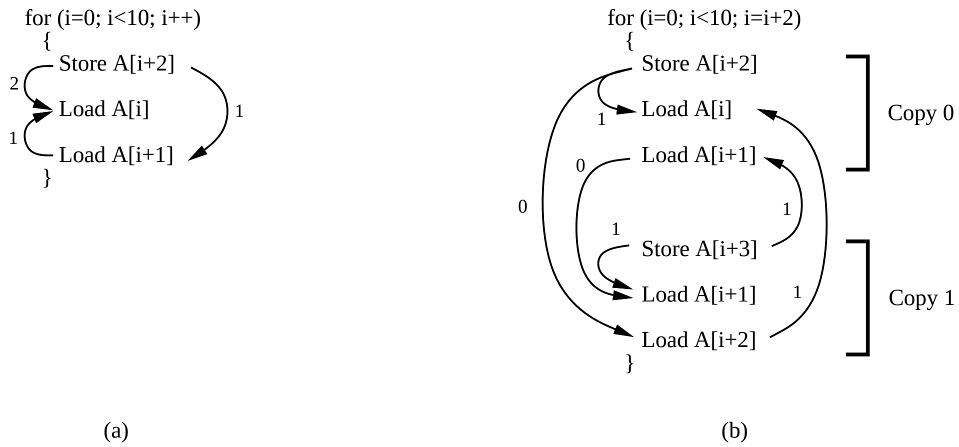


Figure 13: Loop unrolling example, (a) original code, (b) code after loop unrolling.

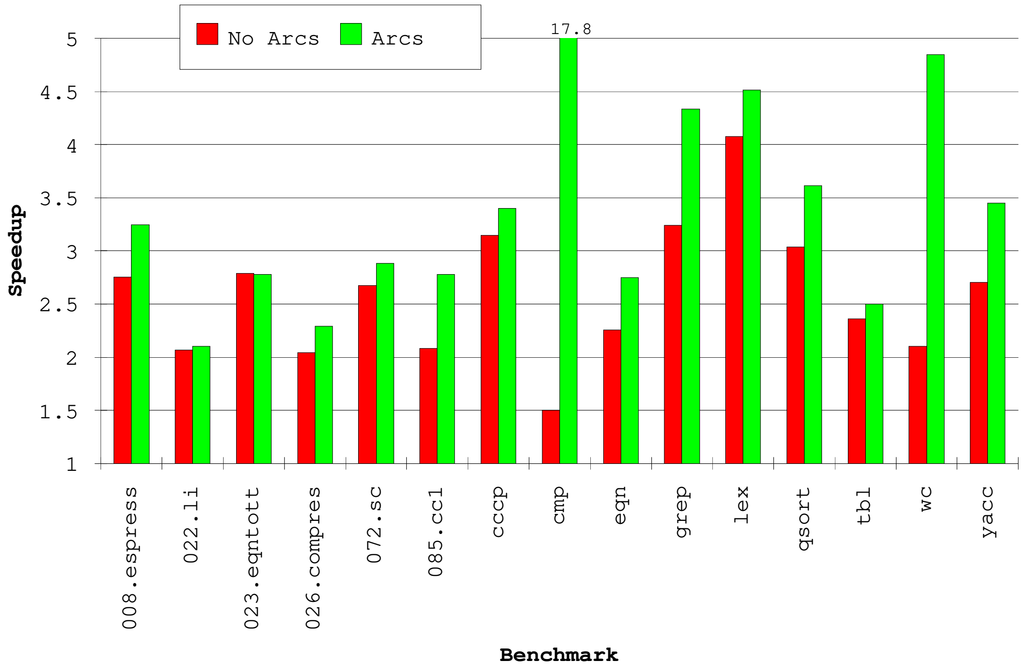


Figure 14: Integer benchmark speedups for an 8-issue processor.

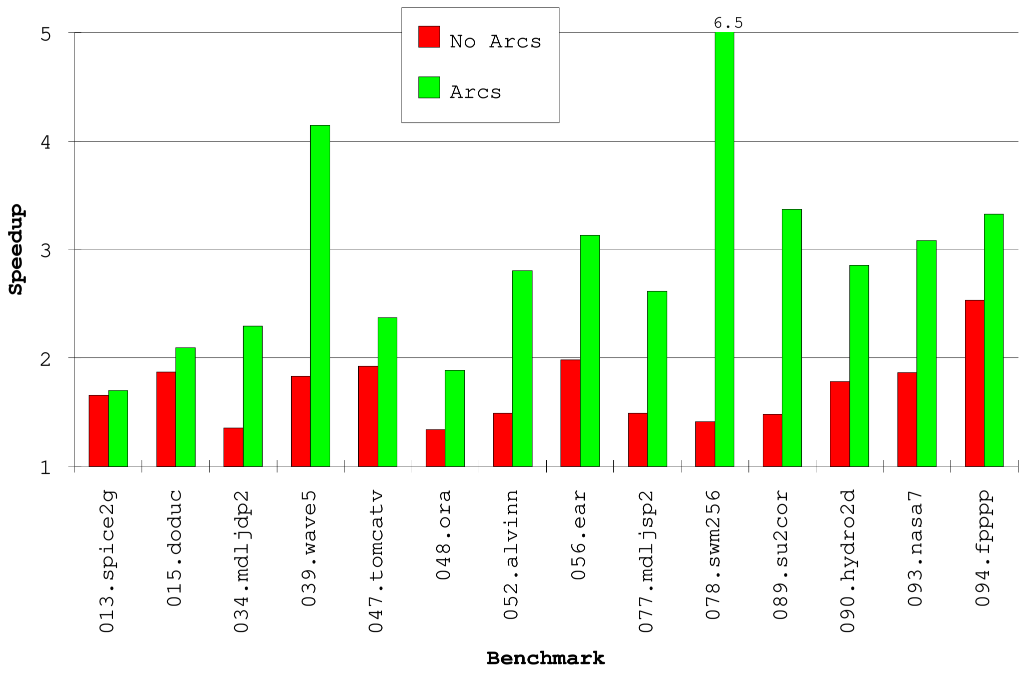


Figure 15: Floating-point benchmark speedups for an 8-issue processor.