The Superblock: An Effective Technique for VLIW and Superscalar Compilation

Wen-mei W. Hwu Scott A. Mahlke William Y. Chen Pohua P. Chang

Nancy J. Warter Roger A. Bringmann Roland G. Ouellette Richard E. Hank

Tokuzo Kiyohara Grant E. Haab John G. Holm Daniel M. Lavery *

Abstract

A compiler for VLIW and superscalar processors must expose sufficient instruction-level parallelism (ILP) to effectively utilize the parallel hardware. However, ILP within basic blocks is extremely limited for control-intensive programs. We have developed a set of techniques for exploiting ILP across basic block boundaries. These techniques are based on a novel structure called the *superblock*. The superblock enables the optimizer and scheduler to extract more ILP along the important execution paths by systematically removing constraints due to the unimportant paths. Superblock optimization and scheduling have been implemented in the IMPACT-I compiler. This implementation gives us a unique opportunity to fully understand the issues involved in incorporating these techniques into a real compiler. Superblock optimizations and scheduling are shown to be useful while taking into account a variety of architectural features.

Index terms - code scheduling, control-intensive programs, instruction-level parallel processing, optimizing compiler, profile information, speculative execution, superblock, superscalar processor, VLIW processor.

*The authors are with the Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign, Illinois, 61801. Pohua Chang is now with Intel Corporation. Roland Ouellette is now with Digital Equipment Corporation. Tokuzo Kiyohara is with Matsushita Electric Industrial Co., Ltd., Japan. Daniel Lavery is also with the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign.

1 Introduction

VLIW and superscalar processors contain multiple data paths and functional units, making them capable of issuing multiple instructions per clock cycle [Colwell et al. 1987; Fisher 1981; Horst et al. 1990; Intel 1989; Rau et al. 1989; Warren 1990]. As a result, the peak performance of the coming generation of VLIW and superscalar processors will range from two to eight times greater than their scalar predecessors which execute at most one instruction per clock cycle. However, previous studies have shown that using conventional code optimization and scheduling methods, superscalar and VLIW processors cannot produce a sustained speedup of more than two for control-intensive programs [Jouppi and Wall 1989; Schuette and Shen 1991; Smith et al. 1989]. For such programs, conventional compilation methods do not provide enough support to utilize these processors.

Traditionally, the primary objective of code optimization is to reduce the number and complexity of executed instructions. Few existing optimizing compilers attempt to increase the instruction-level parallelism, or ILP [1], of the code. This is because most optimizing compilers have been designed for scalar processors, which benefit little from the increased ILP. Since VLIW and superscalar processors can take advantage of the increased ILP, it is important for the code optimizer to expose enough parallelism to fully utilize the parallel hardware.

The amount of ILP within basic blocks is extremely limited for control-intensive programs. Therefore, the code optimizer must look beyond basic blocks to find sufficient ILP. We have developed a set of optimizations to increase ILP across basic block boundaries. These optimizations are based on a novel structure called the *superblock*. The formation and optimization of superblocks increases the ILP along the important execution paths by systematically removing constraints due to the unimportant paths. Because these optimizations increase the ILP within superblocks, they are collectively referred to as *superblock ILP optimizations*.

Unlike code optimization, code scheduling for VLIW processors has received extensive treatment in the literature [Aiken and Nicolau 1988; Bernstein and Rodeh 1991; Ellis 1985; Fisher 1981; Gupta and Soffa 1990]. In particular, the *trace scheduling* technique invented by Fisher has been shown to be very effective for rearranging instructions across basic blocks [Fisher 1981]. An important issue for trace scheduling is the compiler implementation complexity incurred by the need to maintain correct program execution after

[1] One can measure the ILP as the average number of simultaneously executable instructions per clock cycle. It is a function of the data and control dependences between instructions in the program as well as the instruction latencies of the processor hardware. It is independent of all other hardware constraints.

moving instructions across basic blocks. The code scheduling technique described in this paper, which is derived from trace scheduling, employs the superblock. Superblock ILP optimizations remove constraints, and the code scheduler implementation complexity is reduced. This code scheduling approach will be referred to as *superblock scheduling*.

In order to characterize the cost and effectiveness of the superblock ILP optimizations and superblock scheduling, we have implemented these techniques in the IMPACT-I compiler developed at the University of Illinois. The fundamental premise of this project is to provide a complete compiler implementation that allows us to quantify the impact of these techniques on the performance of VLIW and superscalar processors by compiling and executing large control-intensive programs. In addition, this compiler allows us to fully understand the issues involved in incorporating these optimization and scheduling techniques into a real compiler. Superblock optimizations are shown to be useful while taking into account a variety of architectural parameters.

Section 2 of this paper introduces the superblock. Section 3 gives a concise overview of the superblock ILP optimizations and superblock scheduling. Section 4 presents the cost and performance of these techniques. The concluding remarks are made in Section 5.

2 The Superblock

The purpose of code optimization and scheduling is to minimize the execution time while preserving the program semantics. When this is done globally, some optimization and scheduling decisions may decrease the execution time for one control path while increasing the time for another path. By making these decisions in favor of the more frequently executed path, an overall performance improvement can be achieved.

Trace scheduling is a technique that was developed to allow scheduling decisions to be made in this manner [Ellis 1985; Fisher 1981]. In trace scheduling the function is divided into a set of traces that represent the frequently executed paths. There may be conditional branches out of the middle of the trace (side exits) and transitions from other traces into the middle of the trace (side entrances). Instructions are scheduled within each trace ignoring these control-flow transitions. After scheduling, bookkeeping is required to ensure the correct execution of off-trace code.

Code motion past side exits can be handled in a fairly straightforward manner. If an instruction I is moved from above to below a side exit, and the destination of I is used before it is redefined when the side

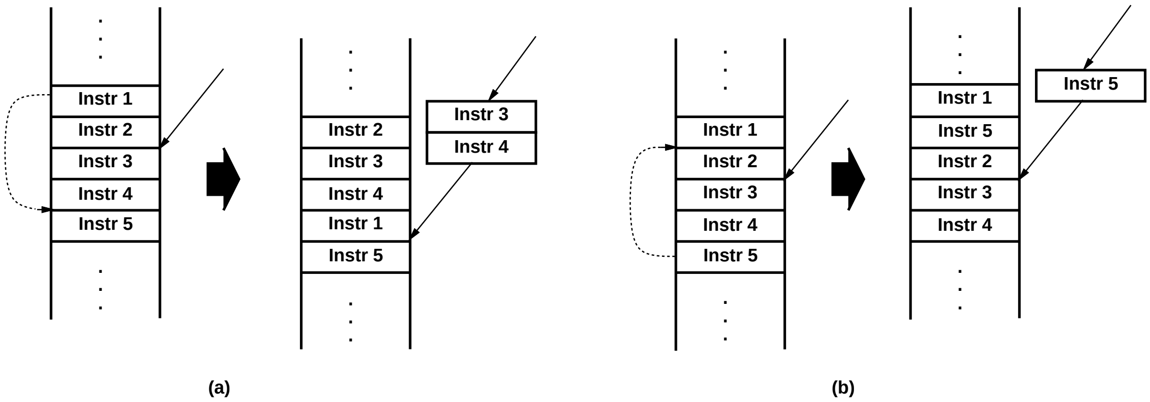

Figure 1: Instruction scheduling across trace side entrances. (a) Moving an instruction below a side entrance. (b) Moving an instruction above a side entrance.

exit is taken, then a copy of I must also be placed between the side exit and its target. Movement of an instruction from below to above a branch can also be handled without too much difficulty. The method for doing this is described in section 3.4.

More complex bookkeeping must be done when code is moved above and below side entrances. Figure 1 illustrates this bookkeeping. In Figure 1(a), when *Instr 1* is moved below the side entrance (to after *Instr 4*), the side entrance is moved below *Instr 1*. *Instr 3* and *Instr 4* are then copied to the side entrance. Likewise, in Figure 1(b), when *Instr 5* is moved above the side entrance, it must also be copied to the side entrance.

Side entrances can also make it more complex to apply optimizations to traces. For example, Figure 2 shows how copy propagation can be applied to the trace and the necessary bookkeeping for the off-trace code. In this example, in order to propagate the value of *r1* from *I1* to *I3*, bookkeeping must be performed. Before propagating the value of *r1*, the side entrance is moved to below *I3* and instructions *I2* and *I3* are copied to the side entrance.

The bookkeeping associated with side entrances can be avoided if the side entrances are removed from the trace. A *superblock* is a trace which has no side entrances. Control may only enter from the top but may leave at one or more exit points. Superblocks are formed in two steps. First, traces are identified using execution profile information [Chang and Hwu 1988]. Second, a process called tail duplication is performed to eliminate any side entrances to the trace [Chang et al. 1991b]. A copy is made of the tail portion of the trace from the first side entrance to the end. All side entrances into the trace are then moved to the corresponding duplicate basic blocks. The basic blocks in a superblock need not be consecutive in the code. However, our implementation restructures the code so that all blocks in a superblock appear in consecutive

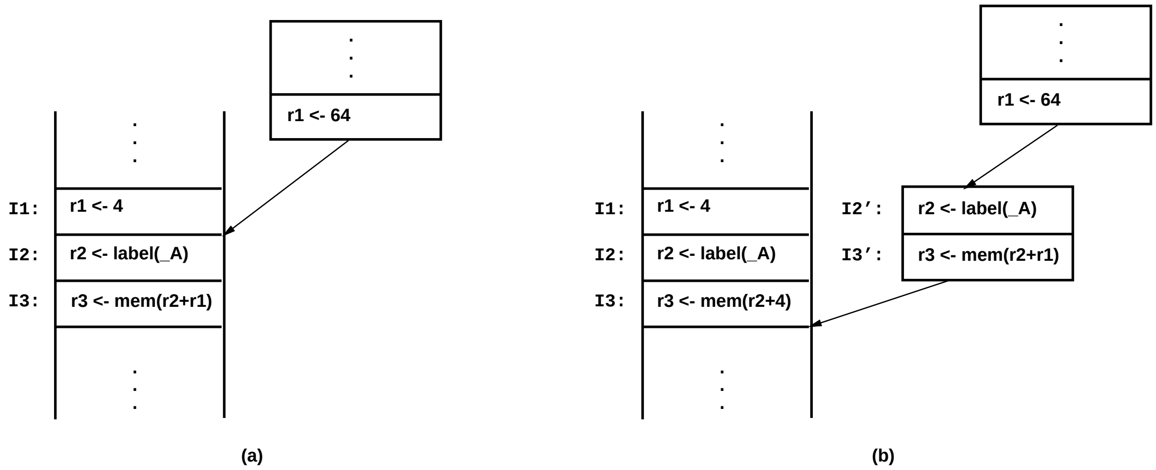

Figure 2: Applying copy propagation to an instruction trace. (a) Before copy propagation. (b) After copy propagation with bookkeeping code inserted.

order for better cache performance.

The formation of superblocks is illustrated in Figure 3. Figure 3(a) shows a weighted flow graph which represents a loop code segment. The nodes correspond to basic blocks and arcs correspond to possible control transfers. The *count* of each basic block indicates the execution frequency of that basic block. In Figure 3(a), the *count* of $\{A, B, C, D, E, F\}$ is $\{100, 90, 10, 0, 90, 100\}$, respectively. The *count* of each control transfer indicates the frequency of invoking these control transfers. In Figure 3(a), the *count* of $\{A \rightarrow B, A \rightarrow C, B \rightarrow D, B \rightarrow E, C \rightarrow F, D \rightarrow F, E \rightarrow F, F \rightarrow A\}$ is $\{90, 10, 0, 90, 10, 0, 90, 99\}$, respectively. Clearly, the most frequently executed path in this example is the basic block sequence $< A, B, E, F >$. There are three traces: $\{A, B, E, F\}$, $\{C\}$, and $\{D\}$. After trace selection, each trace is converted into a superblock. In Figure 3(a), we see that there are two control paths that enter the $\{A, B, E, F\}$ trace at basic block F. Therefore, we duplicate the tail part of the $\{A, B, E, F\}$ trace starting at basic block F. Each duplicated basic block forms a new superblock that is appended to the end of the function. The result is shown in Figure 3(b). Note that there are no longer side entrances into the most frequently traversed trace, $< A, B, E, F >$; it has become a superblock.

Superblocks are similar to the extended basic blocks. An extended basic block is defined as a sequence of basic blocks $B_1 \ldots B_k$ such that for $1 \leq i < k$, B_i is the only predecessor of B_{i+1} and B_1 does not have a unique predecessor [Aho et al. 1986]. The difference between superblocks and extended basic blocks lies mainly in how they are formed. Superblock formation is guided by profile information and side entrances are removed to increase the size of the superblocks. It is possible for the first basic block in a superblock to

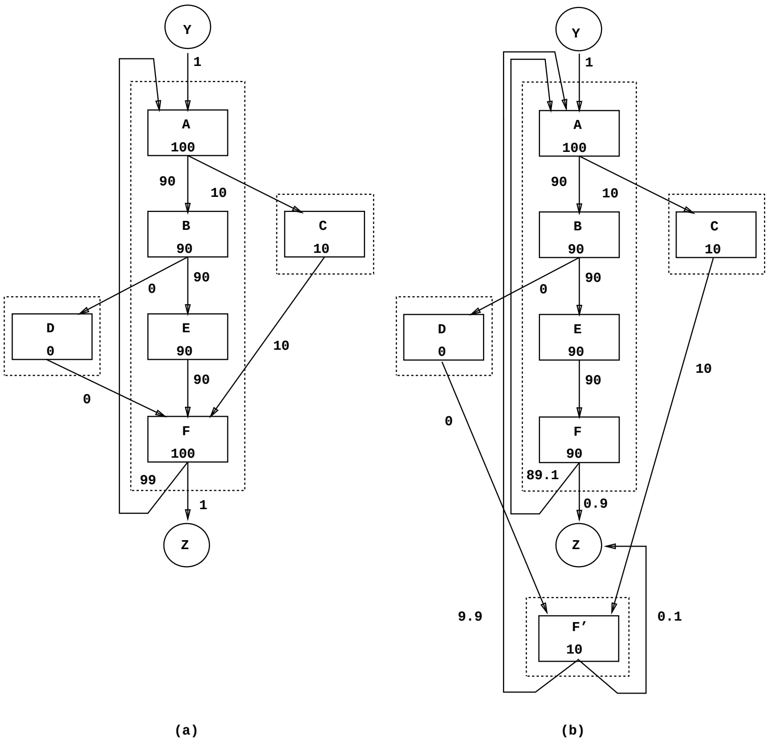

Figure 3: An example of superblock formation.

have a unique predecessor.

3 Superblock ILP Optimization and Scheduling

Before superblock scheduling is performed, superblock ILP optimizations are applied which enlarge the superblock and increase instruction parallelism by removing dependences.

3.1 Superblock Enlarging Optimizations

The first category of superblock ILP optimizations is superblock enlarging optimizations. The purpose of these optimizations is to increase the size of the most frequently executed superblocks so that the superblock scheduler can manipulate a larger number of instructions. It is more likely the scheduler will find independent instructions to schedule at every cycle in a superblock when there are more instructions to choose from. An important feature of superblock enlarging optimizations is that only the most frequently executed parts of a program are enlarged. This selective enlarging strategy keeps the overall code expansion under control [Chang et al. 1991b]. Three superblock enlarging optimizations are described as follows.

Branch Target Expansion. Branch target expansion expands the target superblock of a likely taken control transfer which ends a superblock. The target superblock is copied and appended to the end of the original superblock. Note that branch target expansion is not applied for control transfers which are loop backedges. Branch target expansion continues to increase the size of a superblock until a predefined superblock size is reached, or the branch ending the superblock does not favor one direction.

Loop Peeling. Superblock loop peeling modifies a superblock loop (a superblock which ends with a likely control transfer to itself) which tends to iterate only a few times for each loop execution. The loop body is replaced by straight-line code consisting of the first several iterations of the loop. [2] The original loop body is moved to the end of the function to handle executions which require additional iterations. After loop peeling, the most frequently executed preceding and succeeding superblocks can be expanded into the peeled loop body to create a single large superblock.

Loop Unrolling. Superblock loop unrolling replicates the body of a superblock loop which tends to iterate many times. To unroll a superblock loop N times, $N-1$ copies of the superblock are appended to the original superblock. The loop backedge control transfers in the first $N-1$ loop bodies are adjusted or removed if possible to account for the unrolling. If the iteration count is known on loop entry, it is possible to remove these transfers by using a preconditioning loop to execute the first mod N iterations. However, preconditioning loops are not currently inserted for unrolled loops.

3.2 Superblock Dependence Removing Optimizations

The second category of superblock ILP optimizations is superblock dependence removing optimizations. These optimizations eliminate data dependences between instructions within frequently executed superblocks, which increases the ILP available to the code scheduler. As a side effect, some of these optimizations increase the number of executed instructions. However, by applying these optimizations only to frequently executed superblocks, the code expansion incurred is regulated. Five superblock dependence removing optimizations are described as follows.

Register Renaming. Reuse of registers by the compiler and variables by the programmer introduces artificial anti and output dependences and restricts the effectiveness of the scheduler. Many of these artificial dependences can be eliminated with register renaming [Kuck et al. 1981]. Register renaming assigns unique

[2] Using the profile information, the loop is peeled its expected number of iterations.

registers to different definitions of the same register. A common use of register renaming is to rename registers within individual loop bodies of an unrolled superblock loop.

Operation Migration. Operation migration moves an instruction from a superblock where its result is not used to a less frequently executed superblock. By migrating an instruction, all of the data dependences associated with that instruction are eliminated from the original superblock. Operation migration is performed by detecting an instruction whose destination is not referenced in its home superblock. Based on a cost constraint, a copy of the instruction is placed at the target of each exit of the superblock in which the destination of the instruction is live. Finally, the original instruction is deleted.

Induction Variable Expansion. Induction variables are used within loops to index through loop iterations and through regular data structures such as arrays. When data access is delayed due to dependences on induction variable computation, ILP is typically limited. Induction variable expansion eliminates redefinitions of induction variables within an unrolled superblock loop. Each definition of the induction variable is given a new induction variable, thereby eliminating all anti, output, and flow dependences among the induction variable definitions. However an additional instruction is inserted into the loop preheader to initialize each newly created induction variable. Also, patch code is inserted if the induction variable is used outside the superblock to recover the proper value for the induction variable.

Accumulator Variable Expansion. An accumulator variable accumulates a sum or product in each iteration of a loop. For loops of this type, the accumulation operation often defines the critical path within the loop. Similar to induction variable expansion, anti, output, and flow dependences between instructions which accumulate a total are eliminated by replacing each definition of accumulator variable with a new accumulator variable. Unlike induction variable expansion, though, the increment or decrement value is not required to be constant within the superblock loop. Again, initialization instructions for these new accumulator variables must be inserted into the superblock preheader. Also, the new accumulator variables are summed at all superblock exit points to recover the value of the original accumulator variable. Note that accumulator variable expansion applied to floating point variables may not be safe for programs which rely on the order which arithmetic operations are performed to maintain accuracy. For programs of this type, an option is provided for the user to disable accumulator variable expansion of floating point variables.

Operation Combining. Flow dependences between pairs of instructions with the same precedence each with a compile-time constant source operand can be eliminated with operation combining [Nakatani and Ebcioglu 1989]. Flow dependences which can be eliminated with operation combining often arise between

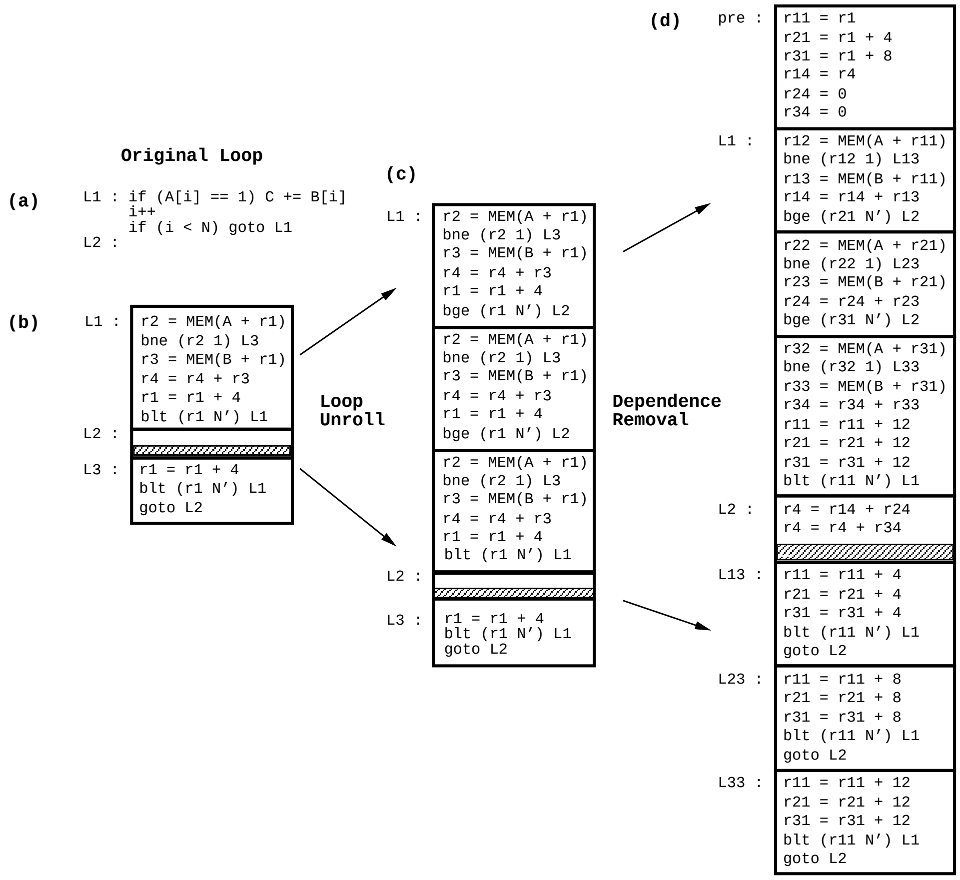

Figure 4: An application of superblock ILP optimizations, (a) original program segment, (b) assembly code after superblock formation, (c) assembly code after superblock loop unrolling, (d) assembly code after superblock dependence removal optimizations.

address calculation instructions and memory access instructions. Also, similar opportunities occur for loop variable increments and loop exit branches. The flow dependence is removed by substituting the expression of the first instruction into the second instruction and evaluating the constants at compile time.

Example. An example to illustrate loop unrolling, register renaming, induction variable expansion, and accumulator variable expansion is shown in Figure 4. This example assumes that the condition of the *if* statement within the loop is likely to be true. After superblock formation, the resulting assembly code is shown in Figure 4(b). To enlarge the superblock loop, loop unrolling is applied. The loop is unrolled three times in this example (Figure 4(c)). After loop unrolling, data dependences limit the amount of ILP in the superblock loop.

The dependences among successive updates of $r4$ (Figure 4(c)) are eliminated with accumulator variable expansion. In Figure 4(d), three temporary accumulators, $r14$, $r24$, and $r34$, are created within the superblock loop. After accumulator expansion, all updates to $r4$ within one iteration of the unrolled loop are independent of one another. In order to maintain correctness, the temporary accumulators are properly initialized in the loop preheader (pre), and the values are summed at the loop exit point (L2). The dependences among successive updates of $r1$ along with updates of $r1$ and succeeding load instructions (Figure 4(c)) are eliminated with induction variable expansion. In Figure 4(d), three temporary induction variables, $r11$, $r21$, and $r31$, are created within the superblock loop. After induction variable expansion, the chain of dependences created by the induction variable is eliminated within one iteration of the unrolled loop. Finally, register renaming is applied to the load instructions to eliminate output dependences. After all superblock ILP optimizations are applied, the execution of the original loop bodies within the unrolled superblock loop may be completely overlapped by the scheduler.

3.3 Superblock Scheduling

After the superblock ILP optimizations are applied, superblock scheduling is performed. Code scheduling within a superblock consists of two steps: dependence graph construction followed by list scheduling. A dependence graph is a data structure which represents the control and data dependences between instructions. A control dependence is initially assigned between each instruction and every preceding branch in the superblock. Some control dependences may then be removed according to the available hardware support described in the following section. After the appropriate control dependences have been eliminated, list scheduling using the dependence graph, instruction latencies, and resource constraints is performed on the superblock.

3.4 Speculative Execution Support

Speculative execution refers to the execution of an instruction before it is known that its execution is required. Such an instruction will be referred to as a *speculative instruction*. Speculative execution occurs when the superblock scheduler moves an instruction J above a preceding conditional branch B. During run time, J will be executed before B, i.e., J is executed regardless of the branch direction of B. However, according to the original program order, J should be executed only if B is not taken. [3] Therefore, the execution result of J must not be used if B is taken, which is formally stated as follows:

[3]Note that the blocks of a superblock are laid out sequentially by the compiler. Each instruction in the superblock is always on the fall-through path of its preceding conditional branch.

Restriction 1. The destination of J is not used before it is redefined when B is taken.

This restriction usually has very little effect on code scheduling after superblock ILP optimization. For example, in Figure 4(d), after dependence removal in block L1, the instruction that loads $B[i]$ into $r13$ can be executed before the direction the preceding *bne* instruction is known. Note that the execution result of this load instruction must not be used if the branch is taken. This is trivially achieved because $r13$ is never used before defined if the branch is taken; the value thus loaded into $r13$ will be ignored in the subsequent execution.

A more serious problem with speculative execution is the prevention of premature program termination due to exceptions caused by speculative instructions. Exceptions caused by speculative instructions which would not have executed on a sequential machine must be ignored, which leads to the following restriction:

Restriction 2. J will never cause an exception that may terminate program execution when branch B is taken.

In Figure 4(d), the execution of the instruction that loads $B[i]$ into r13 could potentially cause a memory access violation fault. If this instruction is speculatively scheduled before its preceding branch and such a fault occurs during execution, the exception should be ignored if the branch was taken. Reporting the exception if the branch is taken would have incorrectly terminated the execution of the program.

Two levels of hardware support for Restriction 2 will be examined in this paper: *restricted percolation model* and *general percolation model*. The restricted percolation model includes no support for disregarding the exceptions generated by the speculative instructions. For conventional processors, memory load, memory store, integer divide, and all floating point instructions can cause exceptions. When using the restricted percolation model, these instructions may not be moved above branches unless the compiler can prove that those instructions can never cause an exception when the preceding branch is taken. The limiting factor of the restricted percolation model is the inability to move potential trap-causing instructions with long latency, such as load instructions, above branches. When the critical path of a superblock contains many of these instructions, the performance of the restricted percolation model is limited.

The general percolation model eliminates Restriction 2 by providing a non-trapping version of instructions that can cause exceptions [Chang et al. 1991a]. Exceptions that may terminate program execution are avoided by converting all speculative instructions which can potentially cause exceptions into their non-trapping versions. For programs in which detection of exceptions is important, the loss of exceptions can be recovered with additional architectural and compiler support [Mahlke et al. 1992]. However, in this

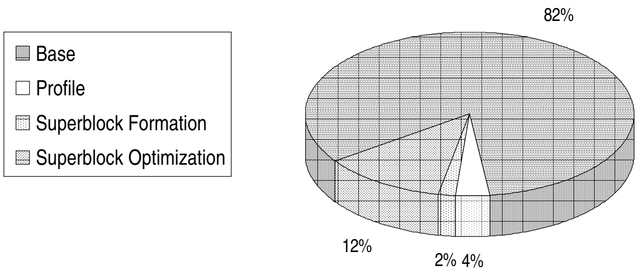

Figure 5: Compiler code size break down. The entire compiler consists of approximately 92,000 lines of C code with one code generator.

paper only the non-trapping execution support is utilized. In Section 4.6, we will show how the general percolation model allows the superblock scheduler to exploit much more of the ILP exposed by superblock ILP optimizations than the restricted percolation model.

4 Implementation Cost and Performance Results

In this section, we report the implications of superblock ILP optimization and scheduling on compiler size, compile time, and output code performance. We additionally examine three architectural factors that directly affect the effectiveness of superblock ILP optimizations and scheduling: speculative execution support, instruction cache misses, and data cache misses.

4.1 Compiler Size

The IMPACT-I C compiler serves two important purposes. First, it is intended to generate highly optimized code for existing commercial microprocessors. We have constructed code generators for the MIPS R2000^{TM}, SPARC^{TM}, Am29000^{TM}, i860^{TM}, and HP PA-RISC^{TM} processors. Second, it provides a platform for studying new code optimization techniques for instruction-level parallel processing. New code optimization techniques, once validated, can be immediately applied to the VLIW and superscalar implementations of existing and new commercial architectures.

Figure 5 shows the percentage of compiler size due to each level of compiler sophistication. The base level accounts for the C front-end, traditional code optimizations, graph-coloring-based register allocation, basic block scheduling, and one code generator [Chaitin 1982; Chow and Hennessy 1990]. The traditional optimizations include classical local and global code optimizations, function inline expansion, instruction placement optimization, profile-based branch prediction, and constant preloading [Aho et al. 1986; Hwu and

Chang 1989a; Hwu and Chang 1989b]. The profile level generates dynamic execution frequency information and feeds this information back to the compiler. The superblock formation level performs trace selection, tail duplication, and superblock scheduling. The superblock ILP optimization level performs branch expansion, loop unrolling, loop peeling, register renaming, induction variable expansion, accumulator expansion, operation migration, and operation combining. As shown in Figure 5, the compiler source code dedicated to the superblock techniques is only about 14% of the entire IMPACT-I compiler.

4.2 Benchmark Programs

Benchmark	Size	Benchmark Description	Profile Description	Input Description
cccp	4787	GNU C preprocessor	20 C source files (100 - 5000 lines)	1 C source file (4000 lines)
cmp	141	compare files	20 similar/dissimilar files	2 similar files (4000 lines)
compress	1514	compress files	20 C source files (100 - 5000 lines)	1 C source file (4000 lines)
eqn	2569	typeset math formulas	20 ditroff files (100 - 4000 lines)	1 ditroff file (17000 lines)
eqntott	3461	boolean minimization	5 files of boolean equations	standard SPEC 92 input
espresso	6722	boolean minimization	20 original espresso benchmarks	opa
grep	464	string search	20 C source files with search strings	1 C source file (4000 lines)
lex	3316	lexical analyzer generator	5 lexers for C, lisp, pascal, awk, pic	C lexer
li	7747	lisp interpreter	5 gabriel benchmarks	queens 7
tbl	2817	format tables for troff	20 ditroff files (100 - 4000 lines)	1 ditroff file (5000 lines)
wc	120	word count	20 C source files (100 - 5000 lines)	1 C source file (4000 lines)
yacc	2303	parser generator	10 grammars for C, pascal, pic, eqn	C grammar

Table 1: The benchmarks.

Table 1 shows the characteristics of the benchmark programs to be used in our compile time and output code performance experiments. The *Size* column indicates the sizes of the benchmark programs measured in numbers of lines of C code excluding comments. The remaining columns describe the benchmarks, the input files used for profiling, and the input file used for performance comparison. In a few cases, such as lex and yacc, one of the profile inputs was used as the test input due to an insufficient number of realistic test cases. Most of the benchmark programs are large and complex enough that it would be virtually impossible to conduct experimentation using these programs without a working compiler.

4.3 Base Code Calibration

All the superblock ILP optimization and scheduling results will be reported as speedup over the code generated by a base compilation. For the speedup measures to be meaningful, it is important to show that the base compilation generates efficient code. This is done by comparing the execution time of our base compilation output against that produced by two high quality production compilers. In Table 2, we compare the output code execution time against that of the MIPS^{TM} C compiler (release 2.1, -O4) and the GNU^{TM} C compiler (release 1.37.1, -O), on a DECstation3100^{TM} which uses a MIPS R2000 processor. The *MIPS.O4* and *GNU.O* columns show the normalized execution time for code generated by the MIPS and GNU C compilers with respect to the IMPACT base compilation output. The results show that our base compiler performs slightly better than the two production compilers for all benchmark programs. Therefore, all the speedup numbers reported in the subsequent sections are based on an efficient base code.

Benchmark	IMPACT.O5	MIPS.O4	GNU.O
cccp	1.00	1.08	1.09
cmp	1.00	1.04	1.05
compress	1.00	1.02	1.06
eqn	1.00	1.09	1.10
eqntott	1.00	1.04	1.33
espresso	1.00	1.02	1.15
grep	1.00	1.03	1.23
lex	1.00	1.01	1.04
li	1.00	1.14	1.32
tbl	1.00	1.02	1.08
wc	1.00	1.04	1.15
yacc	1.00	1.00	1.11

Table 2: Execution times of benchmarks on DECstation3100

4.4 Compile Time Cost

Due to the prototype nature of IMPACT-I, very little effort has been spent minimizing compile time. During the development of the superblock ILP optimizer, compile time was given much less concern than correctness, output code performance, clear coding style, and ease of software maintenance. Therefore, the compile time results presented in this section do not necessarily represent the cost of future implementations of superblock ILP optimizations and scheduling. Rather, the compile time data is presented to provide some initial insight into the compile time cost of superblock techniques.

Figure 6 shows the percentage increase in compile time due to each level of compiler sophistication beyond base compilation on a SPARCstation-II^{TM} workstation. To show the cost of program profiling, we separated the compile time cost of deriving the execution profile. The profiling cost reported for each benchmark in Figure 6 is the cost to derive the execution profile for a typical input file. To acquire stable profile information, one should profile each program with multiple input files.

The superblock formation part of the compile time reflects the cost of trace selection, tail duplication, and increased scheduling cost going from basic block scheduling to superblock scheduling. For our set of benchmarks, the overhead of this part is between 2% and 23% of the base compilation time.

The superblock ILP optimization part of the compile time accounts for the cost of branch expansion, loop

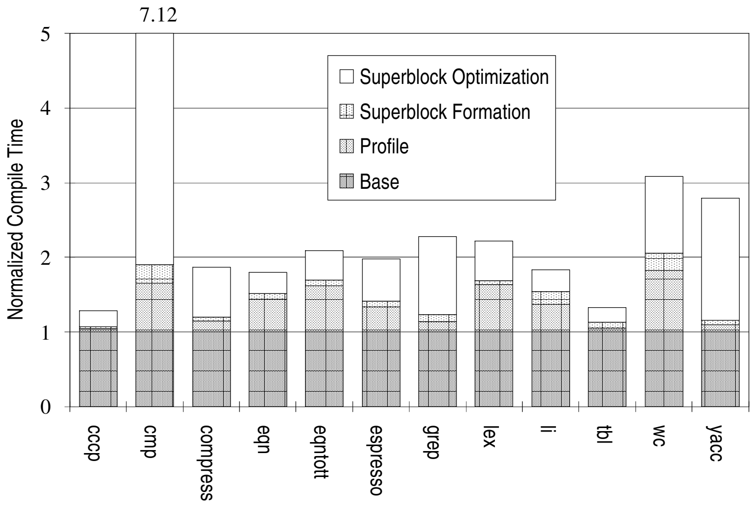

Figure 6: Compile time cost of superblock ILP optimizations.

unrolling, loop peeling, and dependence removing optimizations, as well as the further increase in scheduling cost due to enlarged superblocks. Note that the overhead varies substantially across benchmarks. Although the average overhead is about 101% of the base compilation time, the overhead is as high as 522% for *cmp*. After examining the compilation process in detail, we found out that superblock enlargement created some huge superblocks for *cmp*. Because there are several $O(N^2)$ algorithms used in ILP optimization and code scheduling, where N is the number of instructions in a superblock, the compile time cost increased dramatically due to these huge superblocks. This problem can be solved by the combination of superblock size control and more efficient optimization and scheduling algorithms to decrease the worst-case compile time overhead.

4.5 Performance of Superblock ILP Optimization and Scheduling

The compile time and base code calibration results presented in this paper have been based on real machine execution time. From this point on, we will report the performance of benchmarks based on simulation of a wide variety of superscalar processor organizations. All results will be reported as speedup over a scalar processor executing the base compilation output code.[4] The scalar processor is based on the MIPS R2000 instruction set with extensions for enhanced branch capabilities [Kane 1987]. These include squashing branches, additional branch opcodes such as BLT and BGT, and the ability to execute multiple branches per cycle [Hwu and Chang 1993]. The underlying microarchitecture stalls for flow dependent instructions,

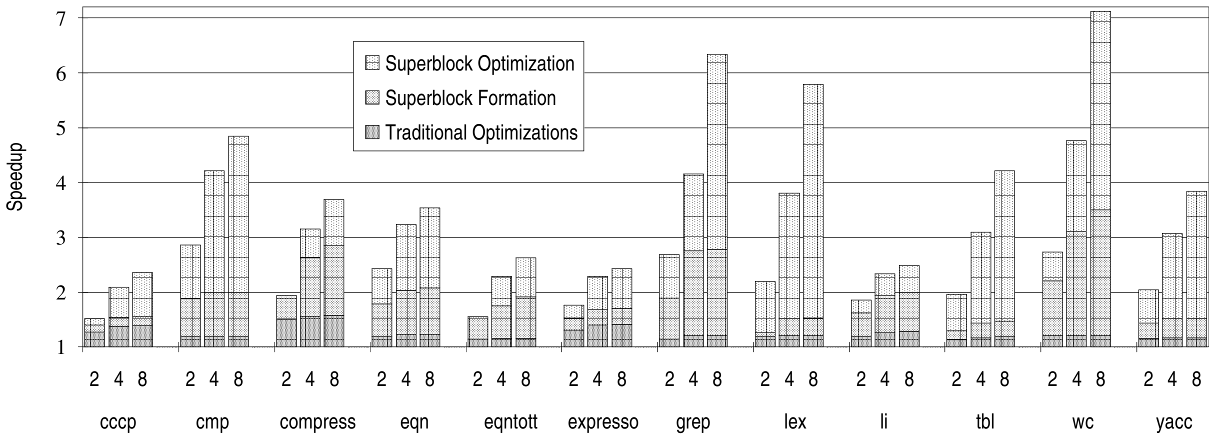

Figure 7: Performance improvement due to superblock ILP optimization. The speedup numbers are relative to the scalar processor with base level compilation.

resolves output dependences with register renaming, and resolves anti-dependences by fetching operands at an early stage of the pipeline. The instruction latencies assumed in the simulations are: 1 cycle for integer add, subtract, comparison, shift, and logic operations, 2 cycles for load in cache, 1 cycle for store, 2 cycles for branches, 3 cycles for integer multiply, 10 cycles for integer divide, 3 cycles for floating point add, subtract, multiply, comparison, and conversion, and 10 cycles for floating point divide. The load and store latency for data cache misses will be discussed in Section 4.8.

To derive the execution cycles for a particular level of compiler optimization and a processor configuration, a simulation is performed for each benchmark. Although the base scalar processor is always simulated with the output code from the base compilation, the superscalar processors are simulated with varying levels of optimization. Experimental results presented in this section assume ideal cache for both scalar and superscalar processors. This allows us to focus on the effectiveness of the compiler to utilize the processor resources. The effect of cache misses will be addressed in Sections 4.7 and 4.8.

In Figure 7, the performance improvement due to each additional level of compiler sophistication is shown for superscalar processors with varying instruction issue and execution resources. An issue K processor has the capacity to issue K instructions per cycle. In this experiment, the processors are assumed to have uniform function units, thus there are no restrictions on the permissible combinations of instructions among the K issued in the same cycle.

As shown in Figure 7, both superblock formation and superblock ILP optimization significantly increase the performance of superscalar processors. In fact, without these techniques, the superscalar processors

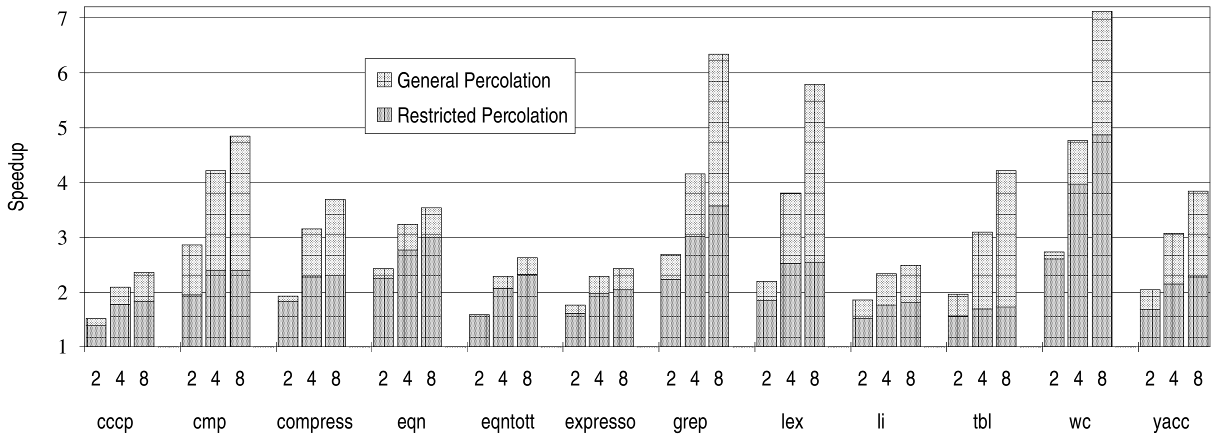

Figure 8: Effect of speculative support on superblock ILP optimization results.

achieve little speedup over the base scalar processor. Note that for *cmp*, *grep*, and *wc*, a 4-issue processor achieves more than four times speedup over the base processor. This speedup is superlinear because the 4-issue processor executes code with superblock optimization whereas the base processor only executes traditionally optimized code. For the 4-issue processor, the cumulative performance improvement due to the superblock techniques range from 53% to 293% over the base compilation. This data clearly demonstrates the effectiveness of the superblock techniques.

4.6 Effect of Speculative Execution

The general code percolation model is perhaps the most important architectural support for superblock ILP optimization and scheduling. The ability to ignore exceptions for speculative instructions allows the superblock scheduler to fully exploit the increased parallelism due to superblock ILP optimizations. This advantage is quantified in Figure 8. The general code percolation model allows the compiler to exploit from 13% to 143% more instruction level parallelism for issue 8. Without hardware support, the scheduler can take advantage of some of the parallelism exposed by superblock optimization. However, using speculative instructions in the general code percolation model, the scheduler is able to improve the performance of the 8-issue processor to between 2.36 and 7.12 times speedup. Furthermore, as the processor issue rate increases, the importance of general code percolation increases. For most benchmarks with restricted percolation, little speedup is obtained from a 4-issue processor to a 8-issue processor. However, when general percolation is used, substantial improvements are observed from a 4-issue processor to a 8-issue processor. These results confirm our qualitative analysis in Section 3.4.

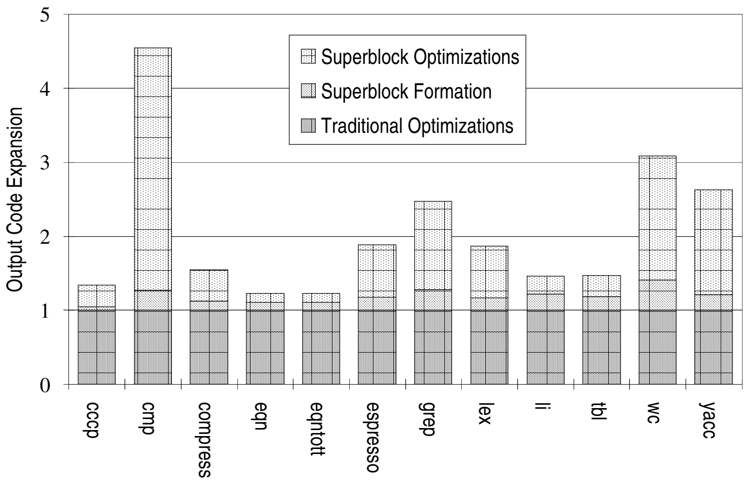

Figure 9: Output code size expansion due to superblock ILP techniques.

4.7 Instruction Cache Effects

The expansion of code from superblock formation and superblock optimizations will have an effect on instruction cache performance. Most superblock ILP optimizations rely on code duplication to enlarge the scheduling scope. Some optimizations, such as accumulator expansion, add new instructions to maintain the correctness of execution after optimization. As shown in Figure 9, the superblock ILP optimizations significantly increase the code size. The code sizes of *eqn* and *cmp* are increased by 23% and 355% respectively. Such code expansion can potentially degrade instruction cache performance. In addition, each stall cycle due to cache misses has a greater impact on the performance of superscalar processors than that of scalar processors. Therefore, it is important to evaluate the impact of instruction cache misses on the performance of superblock ILP optimizations.

Figure 10 shows the speedup of an 8-issue processor over the base scalar processor when taking instruction cache miss penalty into account. The four bars associated with each benchmark correspond to four combinations of two optimization levels and two cache refill latencies. The two cumulative optimization levels are superblock formation (A,C) and superblock ILP optimization (B,D). The two cache refill latencies are 16 and 32 clock cycles. Each bar in Figure 10 has four sections shown the relative performance of four cache sizes: 1K, 4K, 16K, and ideal. The caches are direct mapped with 32-byte blocks. Each instruction cache miss is assumed to cause the processors to stall for the cache refill latency minus the overlap cycles due to a load forwarding mechanism [Chen et al. 1991]. Since instruction cache misses affect both the base scalar processor performance and the superscalar processors performance, speedup is calculated by taking

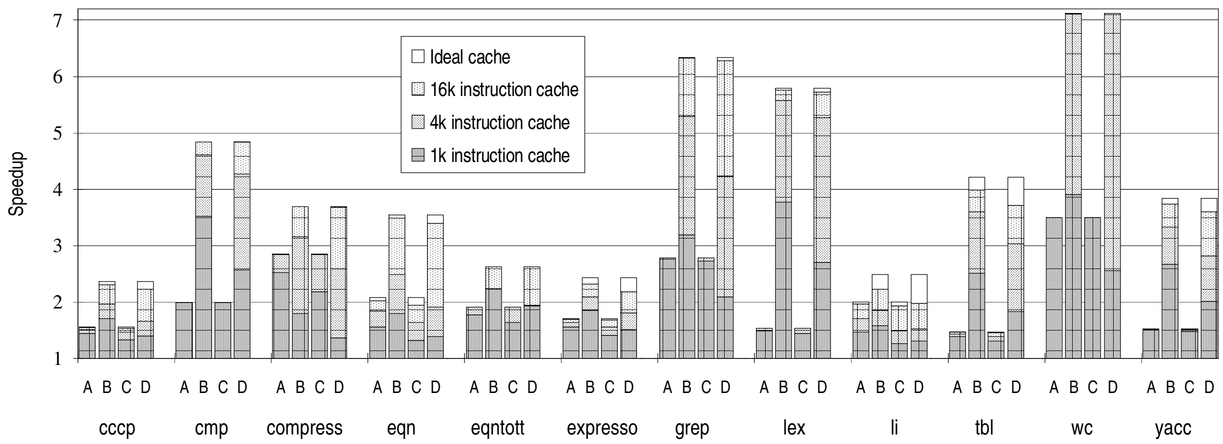

Figure 10: Instruction cache effect on superblock ILP techniques (where, *A* and *C* represent superblock formation, *B* and *D* superblock optimization, *A* and *B* have a cache refill latency of 16 cycles, *C* and *D* have a cache refill latency of 32 cycles).

instruction cache misses into account for both performance measurements.

As shown in Figure 10, for larger caches, superblock ILP optimizations increase performance despite the effect of cache misses. Even for 1K caches, superblock ILP optimizations increase performance for all but *compress*, *grep*, and *wc*. The performance approaches that of ideal cache when the instruction cache is 16K bytes or larger for both 16 cycle and 32 cycle cache refill latencies. Since most modern high performance computers have more than 64K bytes of instruction cache, the performance advantage of superblock ILP optimizations is expected to be relatively unaffected by instruction misses in future high performance computer systems.

4.8 Data Cache Effects

Because superblock optimizations do not affect the number of data memory accesses, the number of extra cycles due to data cache misses remains relatively constant across the optimization levels. However, since the superblock optimizations reduce the number of execution cycles, the overhead due to data cache misses increases. Figure 11 shows the effect of four cache configurations on the performance of an 8-issue processor. The data cache organizations have the same block size and refill latencies as those used in the instruction cache experiments, but the cache sizes are 4K, 16K, 64K, and ideal. Note that data cache misses have more influence on the performance results than instruction cache misses. This is particularly true for the *compress*, *eqntott*, and *lex* benchmarks where there is a noticeable difference between the speedups for the 64K cache

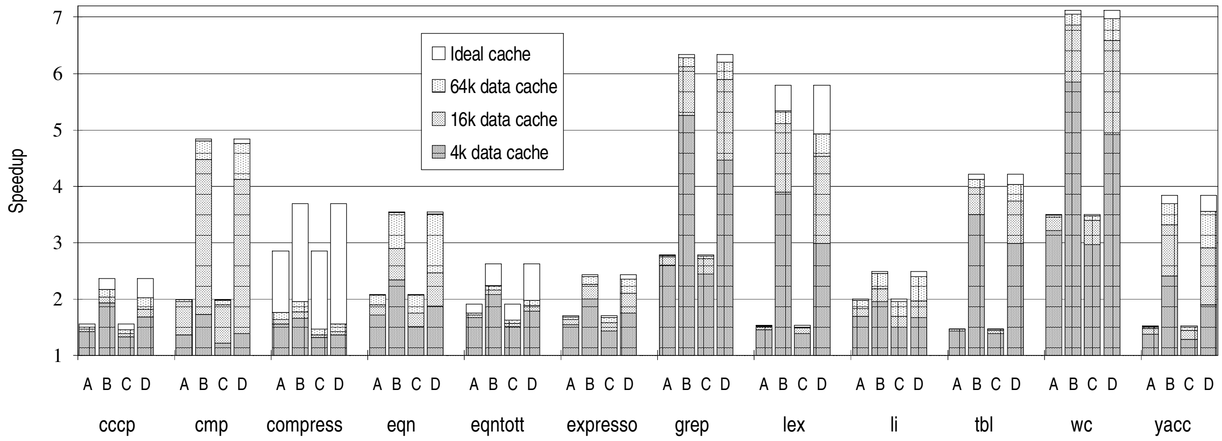

Figure 11: Data cache effect on superblock ILP techniques. (where, A and C represent superblock formation, B and D superblock optimization, A and B have a cache refill latency of 16 cycles, C and D have a cache refill latency of 32 cycles).

and the ideal cache. The poor cache performance in the case of the *compress* benchmark, can be attributed to large internal data structures. The *compress* benchmark has two large tables, each larger than 64K bytes when large input files are used. The effect of the data cache on the performance of superblock optimizations illustrates the need to include data prefetching and other load latency hiding techniques in the compiler.

5 Conclusion

Control intensive programs challenge instruction level parallel processing compilers with excess constraints from many possible execution paths. In order to compile these programs effectively, we have designed the superblock ILP optimizations and superblock scheduling to systematically remove constraints due to unimportant execution paths. The IMPACT-I prototype proves that it is feasible to implement superblock ILP optimization and superblock scheduling in a real compiler. The development effort dedicated to the prototype implementation is about 10 person-years in an academic environment.

The implementation of the superblock techniques accounts for approximately 14% of the compiler source code. Superblock techniques add an average overhead of 101% to the base compilation time. We would like to emphasize that the prototype is not tuned for fast compilation. The results here do not necessarily represent the compile time cost of commercial implementations. Rather, these numbers are reported to prove that the compile time overhead is acceptable in a prototypical implementation.

Using simulation, we demonstrate that superscalar processors achieve much higher performance with superblock ILP optimization and superblock scheduling. For example, the improvement for an 4-issue processor ranges from 53% to 293% across the benchmark programs.

Three architecture factors strongly influence the performance of superscalar and VLIW processors: speculative execution support, instruction cache misses, and data cache misses. We have shown that the general code percolation model allows the compiler to exploit from 13% to 143% more instruction level parallelism. Considering the moderate cost of speculative execution hardware, we expect that many future superscalar and VLIW systems will provide such support.

Although the instruction cache misses can potentially cause large performance degradation, we found that the benchmark performance results remain unaffected for instruction caches of reasonable size. Since most workstations have more than 64K bytes of instruction cache, we do not expect the instruction misses to reduce the performance advantage of superblock ILP optimizations. Similar conclusions can be drawn for the data cache. However, several benchmarks require more advanced data prefetching techniques to compensate for the effect of high cache miss rates.

In conclusion, the IMPACT-I prototype proves that superblock ILP optimization and scheduling are not only feasible but also cost effective. It also demonstrates that substantial speedup can be achieved by superscalar and VLIW processors over the current generation of high performance RISC scalar processors. It provides one important set of data points to support instruction level parallel processing as an important technology for the next generation of high performance processors.

Acknowledgments

The authors would 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-8809478, Dr. Lee Hoevel at NCR, Hewlett-Packard, the AMD 29K Advanced Processor Development Division, Matsushita Electric Corporation, Joint Services Engineering Programs (JSEP) under Contract N00014-90-J-1270, and the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS). Scott Mahlke is supported by an Intel Fellowship. Grant Haab is supported by a Fannie and John Hertz Foundation Graduate Fellowship. John Holm is supported by an AT&T Fellowship. Daniel Lavery is also supported by the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign under Grant DOE DE-FGO2-85ER25001 from the U.S. Department of Energy, and the IBM Corporation.

References

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

Aiken, A. and Nicolau, A. 1988. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering*, 14, (May), 584–594.

Bernstein, D. and Rodeh, M. 1991. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (June), 241–255.

Chaitin, G. J. 1982. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction* (June), 98–105.

Chang, P. P. and Hwu, W. W. 1988. Trace selection for compiling large C application programs to microcode. In *Proceedings of the 21st International Workshop on Microprogramming and Microarchitecture* (Nov.), 188–198.

Chang, P. P., Mahlke, S. A., Chen, W. Y., Warter, N. J., and Hwu, W. W. 1991a. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th International Symposium on Computer Architecture* (May), 266–275.

Chang, P. P., Mahlke, S. A., and Hwu, W. W. 1991b. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21, 12 (Dec.):1301–1321.

Chen, W. Y., Chang, P. P., Conte, T. M., and Hwu, W. W. 1991. The effect of code expanding optimizations on instruction cache design. Technical Report CRHC-91-17, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL.

Chow, F. C. and Hennessy, J. L. 1990. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12, (Oct.), 501–536.

Colwell, R. P., Nix, R. P., O'Donnell, J. J., Papworth, D. B., and Rodman, P. K. 1987. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems* (Apr.), 180–192.

Ellis, J. 1985. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, Cambridge, MA.

Fisher, J. A. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, c-30, (July), 478–490.

Gupta, R. and Soffa, M. L. 1990. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16, (Apr.), 421–431.

Horst, R. W., Harris, R. L., and Jardine, R. L. 1990. Multiple instruction issue in the NonStop Cyclone processor. In *Proceedings of the 17th International Symposium on Computer Architecture* (May), 216–226.

Hwu, W. W. and Chang, P. P. 1989a. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of the 16th International Symposium on Computer Architecture* (May), 242–251.

Hwu, W. W. and Chang, P. P. 1989b. Inline function expansion for compiling realistic C programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (June), 246–257.

Hwu, W. W. and Chang, P. P. Accepted for publication. Efficient instruction sequencing with inline target insertion. *IEEE Transactions on Computers*.

Intel 1989. *i860 64-Bit Microprocessor*. Santa Clara, CA.

Jouppi, N. P. and Wall, D. W. 1989. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems* (Apr.), 272–282.

Kane, G. 1987. *MIPS R2000 RISC Architecture*. Prentice-Hall, Inc., Englewood Cliffs, NJ.

Kuck, D. J. 1978. *The Structure of Computers and Computations*. John Wiley and Sons, New York, NY.

Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B., and Wolfe, M. 1981. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages* (Jan.), 207–218.

Mahlke, S. A., Chen, W. Y., Hwu, W. W., Rau, B. R., and ansker, M. S. S. 1992. Sentinel scheduling for VLIW and superscalar processors. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct.).

Nakatani, T. and Ebcioglu, K. 1989. Combining as a compilation technique for VLIW architectures. In *Proceedings of the 22nd International Workshop on Microprogramming and Microarchitecture* (Sept.), 43–55.

Rau, B. R., Yen, D. W. L., Yen, W., and Towle, R. A. 1989. The Cydra 5 departmental supercomputer. *IEEE Computer*, (Jan.), 12–35.

Schuette, M. A. and Shen, J. P. 1991. An instruction-level performance analysis of the Multiflow TRACE 14/300. In *Proceedings of the 24th International Workshop on Microprogramming and Microarchitecture* (Nov.), 2–11.

Smith, M. D., Johnson, M., and Horowitz, M. A. 1989. Limits on multiple instruction issue. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems* (Apr.), 290–302.

Warren, Jr., H. S. 1990. Instruction scheduling for the IBM RISC system/6000 processor. *IBM Journal of Research and Development*, 34, (Jan.), 85–92.