Compiler Code Transformations for Superscalar-Based High-Performance Systems

Scott A. Mahlke William Y. Chen John C. Gyllenhaal Wen-mei W. Hwu

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

Pohua P. Chang

Intel Corporation
Hillsboro, OR 97124

Tokuzo Kiyohara

Matsushita Electric Industrial Co., LTD.
Osaka, Japan

Abstract

Exploiting parallelism at both the multiprocessor level and the instruction level is an effective means for supercomputers to achieve high-performance. The amount of instruction-level parallelism available to superscalar or VLIW node processors can be limited, however, with conventional compiler optimization techniques. In this paper, a set of compiler transformations designed to increase instruction-level parallelism is described. The effectiveness of these transformations is evaluated using 40 loop nests extracted from a range of supercomputer applications. This evaluation shows that increasing execution resources in superscalar/VLIW node processors yields little performance improvement unless loop unrolling and register renaming are applied. It also reveals that these two transformations are sufficient for DOALL loops. However, more advanced transformations are required in order for serial and DOACROSS loops to fully benefit from the increased execution resources. The results show that the six additional transformations studied satisfy most of this need.

1 Introduction

Supercomputers can potentially achieve large performance improvements by exploiting parallelism at both the multiprocessor and instruction level. Instruction-level parallelism (ILP) refers to executing low level machine instructions, such as memory loads and stores, integer adds, and floating point multiplies, in parallel [1]. In a supercomputer, ILP is often exploited within superscalar or VLIW node processors. For example, the Alliant FX/2800 system supports parallel execution among its 28 Intel i860 node processors and each processor is capable of completing two instructions per clock cycle. The Intel Touchstone system and the Thinking Machines CM5 system provide additional examples of program parallelism exploited at both the multiprocessor and instruction level. The importance of exploiting ILP to improve the performance of superscalar or VLIW node processors will continue to grow in future supercomputer systems.

The amount of ILP available to superscalar or VLIW node processors, though, can be limited with conventional compiler optimization techniques, which are designed for scalar processors. The primary objective of a traditional optimizer is to reduce the number and complexity of the instructions executed by the processor [2]. Due to their limited amount of execution resources, scalar processors benefit little from increased ILP. Superscalar and VLIW processors, on the other hand, achieve higher performance by exploiting ILP with multiple data paths and functional units. As the amount of parallel hardware within node processors continues to grow, optimizers will be required to expose increasing levels of ILP to effectively utilize the parallel hardware.

In this paper, a set of compiler transformations designed to increase the ILP for superscalar and VLIW node processors are presented. We assume an execution model where multiprocessor parallelism is exploited in outer loops and instruction-level parallelism in inner loops. This model is also assumed in the Alliant FX/2800. We further assume that transformations to adjust the granularity and dependence patterns of outer loops are done by a high level restructurer such as KAP [3]. The objective of the transformations discussed in this paper is to increase the ILP within each inner loop. This is achieved by removing dependences between instructions within each iteration as well as across iterations. This removal of dependences makes more instructions independent from each other, thereby increasing the number of instructions that can be executed concurrently.

The transformations presented in this paper have been incorporated in the IMPACT-I prototype compiler developed at the University of Illinois. This prototype has made it possible to quantify the effectiveness of these transformations. In particular, the effect of these transformations on loop nests collected from supercomputer applications is evaluated for superscalar/VLIW processors. Section 3

shows that the performance of these processors is severely limited with only conventional optimizations. It also shows that performance improves drastically with the ILP increasing transformations described in this paper. The performance improvement, however, does not come without cost. To fully exploit the increased ILP, the hardware must provide sufficient registers. Consequently, the register file requirements for these transformations are reported.

1.1 Related Work

Many researchers have addressed compile-time transformations to remove dependences between instructions. Kuck *et al.* discussed transformations, such as scalar expansion and variable renaming, to eliminate anti and output dependences [4]. Nakatani and Ebcioglu presented an operation combining method that removes flow dependences between pairs of instructions [5]. Anantha and Long described a parallelizing compiler that employs loop unrolling, loop peeling, and variable renaming to assist Aiken and Nicolau's percolation scheduling [6]. Techniques to eliminated dependences between unrolled iterations were implemented in the Bulldog, Multiflow trace and Cydra 5 compilers [7] [8] [9]. Several height reduction techniques that reduce the length of dependence chains in arithmetic calculations have been proposed [10] [11]. The transformations discussed in this paper utilize the concepts presented in these previous studies.

The use of code scheduling schemes that can take advantage of eliminated dependences is necessary to get performance improvement from these transformations. Many previous studies have focused on such code scheduling strategies for VLIW and superscalar processors. Fisher developed trace scheduling to effectively schedule instructions across basic blocks which are likely to execute in sequence (trace) [12]. Trace scheduling has been utilized in both the Bulldog compiler [7] and the Multiflow Trace compiler [8] [13]. Superblock scheduling is an extension of trace scheduling which removes some of the bookkeeping complexity associated with code reordering in a trace [14]. Superblock scheduling is the scheduling method used for this study.

Software pipelining is an effective scheduling method to overlap the execution of loop iterations on VLIW and superscalar processors. Rau *et al.* derived and productized a software pipelining technology which takes advantage of special hardware support in the Cydra 5 compiler [9]. Lam presented a software pipelining algorithm that does not require special hardware support [15]. Aiken and Nicolau derived an optimal algorithm for software pipelining for a given set of dependences across loop iterations [16]. These methods also benefit from dependence elimination but the effect of the transformations on these methods is not evaluated in this study.

1.2 Organization of this Paper

The remainder of this paper is organized into 3 sections. Section 2 presents the compiler code transformations for VLIW and superscalar node processors. An experimental evaluation of the code transformations is presented in Section 3. Finally, some concluding remarks are offered in Section 4.

Function	Latency	Function	Latency
Int ALU	1	FP ALU	3
Int multiply	3	FP conversion	3
Int divide	10	FP multiply	3
branch	1 / 1 slot	FP divide	10
memory load	2	memory store	1

Table 1: Instruction latencies. These latencies are used for the examples and the experimental evaluation presented in this paper.

2 Compiler Code Transformations

Eight compiler transformations to increase ILP for superscalar and VLIW processors are described in this section. Two widely used transformations, loop unrolling and register renaming, are first briefly explained. Then more specialized transformations, which expose increasing levels of ILP, are discussed. These transformations consist of accumulator variable expansion, induction variable expansion, search variable expansion, operation combining, strength reduction, and tree height reduction. The first three of these expose increasing levels of ILP of unrolled loops. The remaining three increase the utilization of a superscalar or VLIW processor's arithmetic hardware. A side effect of most of these transformations is to substantially increase the number of processor registers utilized by the program. This effect will be experimentally evaluated in Section 3.

Loop Unrolling. Loop unrolling is a technique commonly used to overlap the execution of multiple iterations of a loop. A loop unrolled N times has $N-1$ copies of the loop body are appended to the original loop. The control transfers to the beginning of the loop are adjusted to account for the unrolling. If the iteration count is known on loop entry, it is possible to remove many of these control transfers by using a preconditioning loop to execute the first Mod N iterations. All of the loop examples used in this paper are of this type. After loop unrolling, the loop body contains N iterations of the loop which can be scheduled as a single unit by the compiler.

Register Renaming. Reuse of registers by the compiler and variables by the programmer introduces artificial anti and output dependences and restricts the effectiveness of a superscalar or VLIW processor. Many of these artificial dependences can be eliminated with register renaming [4]. Register renaming assigns unique 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 loop. An example to illustrate this is shown in Figure 1. The instruction latencies shown in Table 1 are used for all examples in this section.

(a) Original Loop

```
   do 10 j = 1,n
   C(j) = A(j)+B(j)
10 continue
```

(b) Assembly Code

	Assembly	IT
L1:	r2f = MEM(A+r1i)	0
	r3f = MEM(B+r1i)	0
	r4f = r2f+r3f	2
	MEM(C+r1i) = r4f	5
	r1i = r1i + 4	5
	blt (r1i r5i) L1	6

7 cycles / 1 iteration

(c) After Loop Unrolling

	Assembly	IT
L1:	r2f = MEM(A+r1i)	0
	r3f = MEM(B+r1i)	0
	r4f = r2f+r3f	2
	MEM(C+r1i) = r4f	5
	r1i = r1i + 4	5
	r2f = MEM(A+r1i)	6
	r3f = MEM(B+r1i)	6
	r4f = r2f+r3f	8
	MEM(C+r1i) = r4f	11
	r1i = r1i + 4	11
	r2f = MEM(A+r1i)	12
	r3f = MEM(B+r1i)	12
	r4f = r2f+r3f	14
	MEM(C+r1i) = r4f	17
	r1i = r1i + 4	17
	blt (r1i r5i) L1	18

19 cycles / 3 iterations

(d) After Register Renaming

	Assembly	IT
L1:	r21f = MEM(A+r11i)	0
	r31f = MEM(B+r11i)	0
	r41f = r21f+r31f	2
	MEM(C+r11i) = r41f	5
	r12i = r11i + 4	0
	r22f = MEM(A+r12i)	1
	r32f = MEM(B+r12i)	1
	r42f = r22f+r32f	3
	MEM(C+r12i) = r42f	6
	r13i = r12i + 4	1
	r23f = MEM(A+r13i)	2
	r33f = MEM(B+r13i)	2
	r43f = r23f+r33f	4
	MEM(C+r13i) = r43f	7
	r11i = r13i + 4	2
	blt (r11i r5i) L1	7

8 cycles / 3 iterations

Figure 1: Example of loop unrolling and register renaming. All examples in this paper assume a superscalar processor with infinite resources and no register renaming hardware. Also, the issue times (IT) shown are for the code after code scheduling but in order to preserve clarity, the unscheduled code is shown. Sorting by issue time yields the scheduled code.

A simple inner loop (Figure 1a) and its corresponding assembly code (Figure 1b) are shown. Without either unrolling or renaming, each loop iteration requires 7 cycles to execute. If the loop is unrolled 3 times (Figure 1c), each loop iteration requires an average of 6.3 cycles. Finally, if register renaming is applied in addition to loop unrolling (Figure 1d), the average execution time of each loop iteration is reduced to 2.7 cycles.

These two transformations alone are often expose sufficient ILP for low issue machines. However, for higher issue machines additional transformations are needed to utilize hardware effectively. The following transformations fulfill this additional demand.

Accumulator Variable Expansion. An accumulator variable accumulates a sum or product in each iteration of a loop. A common example is the computation of a dot product between two vectors. For loops of this type, the accumulation operation often defines the critical path within the loop. Accumulator variable expansion eliminates redefinitions of an accumulator variable within an unrolled loop by creating k (k refers to the number of accumulation instructions for that accumulation register in the unrolled loop body) temporary accumulators. Each temporary accumulator replaces one definition of the original accumulator in the loop. In this manner all flow, anti, and output dependences are eliminated between the accumulation instructions. To recover the value of the original accumulator variable, the temporary accumulators are summed at all exit points of the loop.

```
accumulator_variable_expansion(loop)
{
  for each variable V in loop:
    Determine if V is an accumulator variable by checking if
    the following conditions are satisfied:
      1. All instructions modifying V are increment/decrement
         instructions
      2. V is only referenced in the above inc/dec instructions
      3. The loop contains more than one of the above
         inc/dec instructions
    If V is an accumulator variable, do this transformation
    to make the inc/dec instructions independent:
      1. Let k be the number of inc/dec instructions
      2. Allocate k new virtual registers (temp accumulators)
      3. Insert initialization code for the above k registers
         into the loop preheader as follows:
         a. Initialize the first register to V's value
         b. Initialize the other k-1 registers to zero
      4. For each inc/dec instruction, replace V by one of
         the above k temp accumulators using each
         accumulator exactly once
      5. At all loop exit points, insert a summation of the k
         temporary accumulators to generate V's exit value
}
```

Figure 2: Algorithm for accumulator variable expansion.

An algorithm to perform accumulator variable expansion for a loop is shown in Figure 2. Note that accumulator variables are only allowed to be modified in the loop by increment or decrement instructions. Also, since the total value of the accumulator is not computed until the loop is exited, no instructions in the loop other than accumulation instructions may use the accumulator variable.

An example to illustrate the application of accumulator variable expansion is presented in Figure 3. This loop (Figure 3a) is the inner most loop for matrix multiplication. After conventional compiler optimization (Figure 3b), each iteration of the loop requires 8 cycles to execute. Loop unrolling and register renaming (Figure 3c) further improves the average execution time to 4.7 cycles per iteration. However, it is clear that the accumulation of the sum into $r1$ limits the ILP in the loop. Accumulator variable expansion (Figure 3d), removes the dependences between instructions which increment/decrement $r1$ by introducing 3 temporary accumulators, $r11$, $r12$, and $r13$. With this transformation, an average of 3.3 cycles is required for each iteration. Also, the application of the induction variable expansion, described next, would improve this time to 2.7 cycles per iteration.

Induction Variable Expansion. Induction variables are used within loops to index through loop iterations and through regular data structures such as arrays. The value

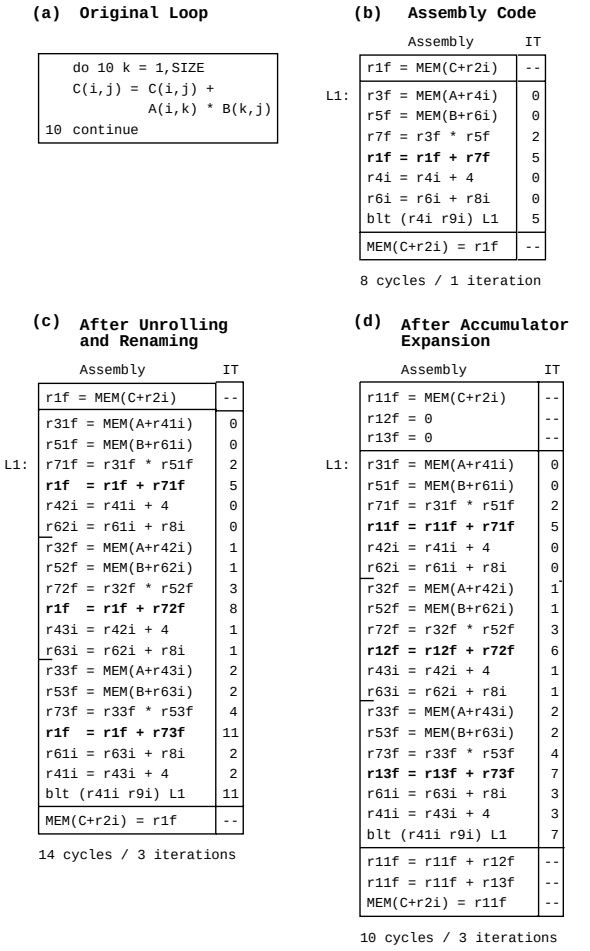

Figure 3: Example of accumulator variable expansion.

of an induction variable is used to compute the address of data structures, and therefore must be computed before the data access is performed. When data access is delayed due to dependences on induction variable computation, ILP is typically limited. Induction variable expansion eliminates flow, anti, and output dependences between definitions of induction variables and their uses within an unrolled loop body by creating k (k is the number of instructions which update this induction variable in the unrolled loop body) temporary induction variables. Each temporary induction variable replaces one definition of the original induction variable in the loop. Also, the increments of each temporary induction variable are moved to the end of the unrolled loop body to eliminate the flow dependences between each definition of a temporary induction variable and its uses. Each temporary induction variable is initialized to its correct value in the loop preheader.

An algorithm to perform induction variable expansion for a loop is presented in Figure 4. Note that there are two major distinctions between induction variables and accumulator variables. First, the value of an accumulator variable may only be used by accumulation instructions in

```
induction_variable_expansion(loop)
{
  for each variable V in loop:
    Determine if V is an induction variable by checking if
    the following conditions are satisfied:
      1. All instructions modifying V are increment/decrement
         instructions
      2. The inc/dec value is the same for all of the above
         inc/dec instructions and it is invariant in the loop
      3. The loop contains more than one of the above
         inc/dec instructions
    If V is an induction variable, do this transformation
    to make the inc/dec instructions independent:
      1. Let k be the number of inc/dec instructions
      2. Let m be the loop invariant inc/dec value
      3. Allocate k+2 new virtual registers (k+1 temporary
         induction registers and one modified inc/dec value z)
      4. Let the k+1 temp ind registers be numbered
         p = 0 to k
      5. Insert initialization code for the above registers
         into the loop preheader as follows:
         a. Initialize register p with V's value + p * m
         b. Initialize register z with k * m
      6. Replace all references to V before the first inc/dec
         instruction by references to temp ind register 0
      7. Replace all references to V between the pth inc/dec
         instruction and the (p+1)th inc/dec instruction by
         references to temp ind register p
      8. Remove all the inc/dec instructions from the loop
      9. Before each branch back to the start of the loop,
         increment the k+1 registers by register z's value
}
```

Figure 4: Algorithm for induction variable expansion.

the loop, whereas induction variables are used by at least one other instructions in the loop. Second, the increment of an accumulator variable may vary with each iteration of the loop, however induction variables must be incremented by a loop invariant amount.

An example loop to illustrate the application of induction variable expansion is shown in Figure 5a. After conventional compiler optimization (Figure 5b), each loop iteration requires 6 cycles to execute. With loop unrolling and register renaming (Figure 5c), this is reduced to 2.7 cycles per iteration. However, the increments of $r2i$ changed by register renaming are still flow dependent. Induction variable expansion (Figure 5d) changes the increments of the three registers renaming created for $r2i$ so that the definitions are independent. This further reduces the number of cycles to 2 per iteration. The improvement becomes more pronounced the more the loop is unrolled. For example, the same loop unrolled 8 times would require 1.6 cycles per iteration after renaming but only 0.8 cycles per iteration after induction variable expansion.

Search Variable Expansion. A single value, such as a maximum or minimum, is often determined for matrices or arrays. Such values will be referred to as search values. Within an unrolled loop body, the chain of flow dependences between successive tests and updates of a search variable often defines a critical path. Similar to accumulator variable expansion and induction variable expansion,

(a) Original Loop

```
   do 10 i = 1,n
   C(j) = A(j)*B(j)
   j = j + K
10 continue
```

(b) Assembly Code

	Assembly	IT
L1:	r3f = MEM(A+r2i)	0
	r4f = MEM(B+r2i)	0
	r5f = r3f * r4f	2
	MEM(C+r2i) = r5f	5
	r2i = r2i + r7i	5
	r1i = r1i + 1	0
	blt (r1 r6) L1	5

6 cycles / 1 iteration

(c) After Unrolling and Renaming

	Assembly	IT
L1:	r31f = MEM(A+r21i)	0
	r41f = MEM(B+r21i)	0
	r51f = r31f * r41f	2
	MEM(C+r21i) = r51f	5
	r22i = r21i + r7i	0
	r32f = MEM(A+r22i)	1
	r42f = MEM(B+r22i)	1
	r52f = r32f * r42f	3
	MEM(C+r22i) = r52f	6
	r23i = r22i + r7i	1
	r33f = MEM(A+r23i)	2
	r43f = MEM(B+r23i)	2
	r53f = r33f * r43f	4
	MEM(C+r23i) = r53f	7
	r21i = r23i + r7i	2
	r1 = r1 + 3	0
	blt (r1 r6) L1	7

8 cycles/ 3 iterations

(d) After Induction Expansion

	Assembly	IT
	r21i = r2i	--
	r22i = r21i + r7i	--
	r23i = r22i + r7i	--
	r71i = r7i * 3	--
L1:	r31f = MEM(A+r21i)	0
	r41f = MEM(B+r21i)	0
	r51f = r31f * r41f	2
	MEM(C+r21i) = r51f	5
	r32f = MEM(A+r22i)	0
	r42f = MEM(B+r22i)	0
	r52f = r32f * r42f	2
	MEM(C+r22i) = r52f	5
	r33f = MEM(A+r23i)	0
	r43f = MEM(B+r23i)	0
	r53f = r33f * r43f	2
	MEM(C+r23i) = r53f	5
	r21i = r21i + r71i	5
	r22i = r22i + r71i	5
	r23i = r23i + r71i	5
	r1 = r1 + 3	0
	blt (r1 r6) L1	5

6 cycles / 3 iterations

Figure 5: Example of induction variable expansion.

search variable expansion eliminates this chain of dependences by creating k temporary search variables. Each temporary search variable replaces one definition of the original search variable in the loop. In this manner, each loop body in the unrolled loop determines a value for the search variable during execution. When the loop is exited, the value of the original search variable is obtained by comparing the values of all temporary search variables.

Operation Combining. Flow dependences between pairs of instructions each with a compile-time constant source operand can be eliminated with operation combining [5]. Flow dependences which can be eliminated with operation combining often arise between address calculation instructions and memory access instructions along with loop variable increments and loop exit branches. To illustrate the application of operation combining, consider the following two flow dependent instructions:

I1: r1 = r2 op1 C1
I2: r3 = r1 op2 C2 $\Rightarrow$ r3 = r2 op2 (C1 op3 C2)

The instructions are combined in two steps. First the non-constant source operand of $I2$ is replaced by the non-constant source operand of $I1$. Therefore, $r1$ is replaced

(a) Original Loop

```
10 i = i + 1
   t = A(i+2) - 3.2
   if (t.LT.10.0) goto 10
```

(b) Assembly Code

	Assembly	IT
L1:	r1i = r1i + 4	0
	r2f = MEM(r1i+8)	1
	r3f = r2f - 3.2	3
	blt (r3f 10.0) L1	6

7 cycles / 1 iteration

(c) After Combining

	Assembly	IT
L1:	r2f = MEM(r1i+12)	0
	r1i = r1i + 4	0
	r3f = r2f - 3.2	2
	blt (r2f 13.2) L1	2

5 cycles / 1 iteration

Figure 6: Example of operation combining.

by $r2$. If the destination and source registers for $I1$ are the same, the positions of $I1$ and $I2$ are exchanged rather than switching source operands. Second, the constant source operands are evaluated according to the $op1$ and $op2$ operations. For this case, if both $op1$ and $op2$ are add operations, $C2$ is replaced by $C1 + C2$.

Clearly, the combination of operations is limited to those of the same precedence and data type (e.g., an add and a multiply operation cannot be combined). The current implementation allows the following operations on the left to be combined with the operations on the right (i indicates integer and f indicates floating point):[1]

(add_i, sub_i)	$\Rightarrow$	(add_i, sub_i, compare_i, load, store, branch_i)
(mul_i)	$\Rightarrow$	(mul_i)
(add_f, sub_f)	$\Rightarrow$	(add_f, sub_f, compare_f, branch_f)
(mul_f, div_f)	$\Rightarrow$	(mul_f, div_f)

An example code segment to illustrate the effectiveness of operation combining is presented in Figure 6a. Without operation combining (Figure 6b), each iteration of the loop requires 7 cycles to execute. However, the flow dependence between the first two instructions and the last two instructions can be eliminated with operation combining. Note that the first 2 instructions must exchange positions when operation combining is performed. After operation combining, the execution time of each loop iteration is reduced to 5 cycles.

Strength Reduction. Strength reduction is a common technique employed by compilers to replace long latency instructions with one or more instructions which collectively require less time. In many existing compilers, integer multiply by a compile-time constant is replaced by a sequence of left shifts and adds [17]. For example, $r2 = r1 * 10$ can be replaced by:

temp1 = r1 $\ll$ 3
temp2 = r1 $\ll$ 1
r2 = temp1 + temp2

[1] Note that if the evaluation of the constants during operation combining results in an overflow or underflow the compiler does not perform the transformation

(a) Original Source Code

```
A = B * (C + D) * E * F / G
```

(b) Assembly Code

Assembly	IT
r1f = rCf + rDf	0
r1f = r1f * rBf	3
r1f = r1f * rEf	6
r1f = r1f * rFf	9
rAf = r1f / rGf	12

22 cycles

(C) After Height Reduction

Assembly	IT
r1f = rCf + rDf	0
r2f = rBf * rEf	0
r3f = rFf / rGf	0
r1f = r1f * r2f	3
rAf = r1f * r3f	10

13 cycles

Figure 7: Example of tree height reduction.

The applicability of this transformation depends on whether the sequence of instructions generated by strength reduction will execute in fewer cycles than the original instruction. The application of strength reduction is typically limited in a scalar processor by this restriction. However, many of the instructions generated during strength reduction are independent and can be executed concurrently on a superscalar or VLIW processor. Therefore, there are more opportunities to apply strength reduction for superscalar and VLIW processors. In addition to applying strength reduction for integer multiply, superscalar and VLIW processors may benefit from reduction of integer divide and integer remainder by a compile-time constant.

Tree Height Reduction. Scalar processor compilers typically generate code for arithmetic expressions by minimizing both the total number of instructions and the total number of temporary registers required. For superscalar and VLIW processors, however, these methods often limit performance by restricting parallel computation of individual components of an arithmetic expression. Tree height reduction exposes ILP in the computation of an arithmetic expression [10] [11]. Tree height reduction first constructs an expression tree to represent the arithmetic expression. The tree is then balanced to reduce the height. The height represents the number of cycles to compute the expression using a specific processor model.

The compiler used in this study uses a modified version of the algorithm proposed by Baer and Bovet [10] that examines intermediate code rather than source code. This tree height reduction algorithm utilizes communicativity and associativity to reduce the height of expressions using -, +, *, /, (,). Note that this algorithm does not apply distributive property. It also assumes all operations have the same latency which is not true for the processor model studied and therefore limits its effectiveness in this case. We are currently examining more advanced techniques for height reduction which utilize the distributive property and allow different latencies for operations.

An example arithmetic computation to illustrate the effectiveness of tree height reduction is presented in Figure 7a. With conventional code optimization techniques (Figure 7b), the computation of the expression requires 22 cycles. With tree height reduction (Figure 7b), two multiply instructions and an add instruction can be executed in parallel with the divide, thereby reducing the execution time to 13 cycles.

3 Experimental Evaluation

In this section, the effectiveness of the compile-time transformations to increase ILP for superscalar and VLIW node processors is analyzed. The performance analysis is done using 40 loop nests extracted from a range of supercomputer applications.

3.1 Methodology

The compiler transformations previously discussed have been implemented in the IMPACT-I compiler. The IMPACT-I compiler is a prototype optimizing compiler designed to generated efficient code for superscalar and VLIW systems [14]. The code generation strategy used by the compiler consists of superblock scheduling [18] and graph-coloring-based register allocation. Superblock scheduling is a global scheduling technique similar to trace scheduling [12] [7] which reduces some of the compiler complexity associated with inter basic block code reordering. The compiler also utilizes a machine description file to generate code for a parameterized superscalar/VLIW node processor.

For this study, the underlying processor microarchitecture is assumed to have in-order execution with register interlocking and deterministic instruction latencies (Table 1 in Section 2). The instruction set is a RISC assembly language similar to the MIPS R2000 instruction set. The processor is assumed to support non-excepting loads and floating point instructions to enable the compiler to schedule these instructions before previous branches which they are control dependent on. The processor that is modeled has an unlimited supply of registers, however the register allocator attempts utilize the least number of registers required for a given loop. Therefore, registers are reused as soon as they become available. The effect of the transformations on register utilization is analyzed in Section 3.2.

For each machine configuration, the execution time of each loop nest, assuming a 100% cache hit rate, is derived using execution-driven simulation. In this analysis, 40 loop nests extracted from the PERFECT club benchmark suite [19], SPEC benchmark suite, and vector library routines are used. Loop nests were selected from the highest execution frequency loop nests of each benchmark. A description of each loop nest is presented in Table 2. The *Name* column specifies from which benchmark the loop nest originated. The remaining information is specified for the innermost loop in the loop nest. The *Size* column indicates the number of lines of FORTRAN source code in the innermost loop. The *Iters* column shows the average number of iterations executed in the innermost loop. The *Nest* column specifies the nesting depth of the innermost loop. The *Type* column indicates whether the innermost loop is DOALL, DOACROSS, or serial. This information

Name	Size†	Iters	Nest	Type	Conds
PERFECT					
APS-1	2	64	2	doall	no
APS-2	8	31	2	doall	no
APS-3	2	776	1	doall	no
CSS-1	6	67	1	serial	yes
LWS-1	2	343	2	serial	no
LWS-2	1	3087	2	serial	no
MTS-1	2	423	2	serial	yes
MTS-2	2	24	3	serial	yes
NAS-1	22	1500	1	doall	no
NAS-2	5	1520	1	doall	no
NAS-3	6	6000	1	doall	no
NAS-4	2	1204	1	serial	no
NAS-5	71	1500	2	serial	no
NAS-6	24	635	2	doacross	no
SDS-1	1	25	2	serial	no
SDS-2	1	32	3	serial	no
SDS-3	1	25	2	serial	no
SDS-4	3	25	2	doacross	no
SRS-1	3	287	1	doall	no
SRS-2	5	287	2	doacross	no
SRS-3	1	287	2	doall	no
SRS-4	9	87	3	doall	no
SRS-5	21	287	2	doall	no
SRS-6	1	287	2	serial	no
TFS-1	11	89	2	doall	no
TFS-2	7	120	2	doacross	no
TFS-3	2	49	3	doall	no
WSS-1	1	96	2	doall	no
WSS-2	4	39	2	doacross	no
SPEC					
doduc-1	38	13	1	serial	yes
matrix300-1	1	300	1	doall	no
nasa7-1	1	256	3	doall	no
nasa7-2	3	1000	3	doacross	no
tomcatv-1	21	255	2	doall	no
tomcatv-2	8	255	2	serial	yes
VECTOR					
add	1	1024	1	doall	no
dotprod	1	1024	1	serial	no
maxval	3	1024	1	serial	yes
merge	4	1024	1	doall	yes
sum	1	1024	1	serial	no

† Specifies number of lines of FORTRAN source code.

Table 2: Description of loop nests.

was derived by parallelizing each application with KAP [3]. Finally the *Conds* column states whether the innermost loops contains conditional branches.

3.2 Results

The performance of varying levels of compiler transformations is compared for superscalar/VLIW node processors with issue rates 2, 4 and 8. The issue rate is the maximum number of instructions the processor can fetch and issue per cycle. No limitation has been placed on the combination of instructions that can be issued in the same cycle. Five levels of compiler transformations are evaluated: conventional scalar processor optimizations (Conv); loop unrolling (Lev1); register renaming (Lev2); operation combining, strength reduction, and height reduction (Lev3);

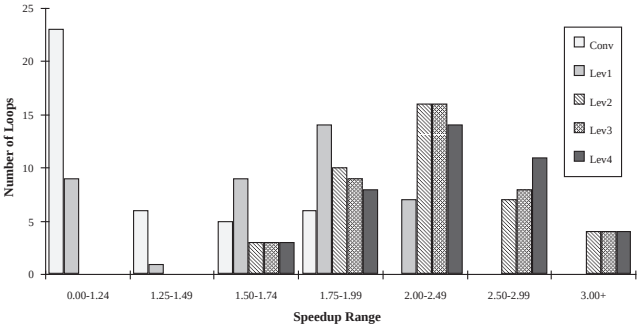

Figure 8: Speedup distribution for an issue-2 superscalar/VLIW processor.

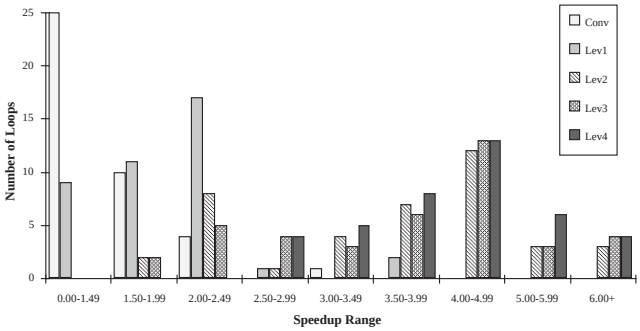

Figure 9: Speedup distribution for an issue-4 superscalar/VLIW processor.

accumulator variable expansion, induction variable expansion, and search variable expansion (Lev4). Each successive level includes all transformations from previous levels. The conventional scalar transformations consist of a complete set of classical local, global, and loop transformations, including constant propagation, copy propagation, common subexpression elimination, constant folding, operation folding, redundant memory access elimination, dead code removal, loop invariant code removal, loop induction variable strength reduction, and loop induction variable elimination [2].

The base configuration for all speedup calculations is an issue-1 processor with conventional compiler transformations. Note that since we are varying both compiler transformation level and issue rate of a superscalar processor, super-linear speedups may be reported. Thus, for example an issue-4 processor may achieve greater than 4x speedup over the base configuration.

Effect of Transformations on Speedup. The speedup distributions for the loop nests achieved by varying the degree of compiler transformations for an issue-2, issue-4, and issue-8 superscalar/VLIW processor are shown in Figures 8, 9, and 10, respectively. The number of loops with a particular transformation level whose speedup falls within the given range is plotted.

For all figures, the effectiveness of increasing degrees of compiler transformations is clearly shown. With conven-

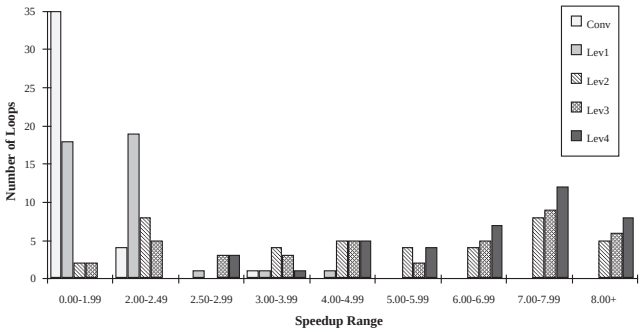

Figure 10: Speedup distribution for an issue-8 superscalar/VLIW processor.

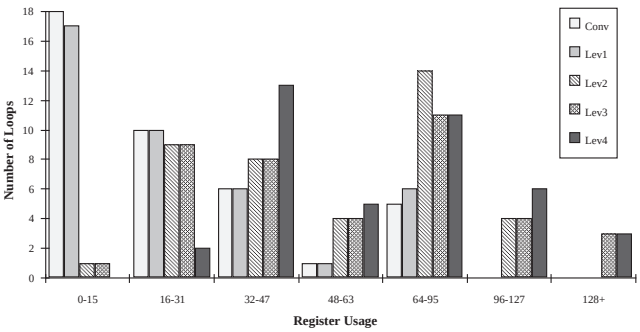

Figure 11: Register usage distribution for an issue-8 superscalar/VLIW processor.

tional compiler optimization, few loops achieve speedup beyond the first several speedup ranges. Loop unrolling results in small speedup increases, but the inability to remove data dependences among instructions in different unrolled bodies limits the effectiveness of loop unrolling alone. (It should be noted loops are unrolled a maximum of 8 times or until a maximum loop body size is reached, whichever limit is reached first.) Register renaming applied in addition to loop unrolling results in substantial speedup increases for all processors. For example with an issue-4 processor, 29 loops achieve 3x or greater speedup and 18 loops achieve 4x or greater speedup. Moderate performance improvements are observed with operation combining, strength reduction, and height reduction additionally applied. Accumulator variable expansion, induction variable expansion, and search variable expansion provide further large improvements in the performance of superscalar/VLIW processors. For example with an issue-4 processor, 36 loops achieve 3x or larger speedup and 23 of those 4x or larger speedup.

The need for higher levels of transformations increases as the processor issue rate increases. With the processor capable of executing more instructions concurrently, there is more demand on the compiler to expose ILP. For an issue-2 processor, loop unrolling and register renaming are sufficient compiler transformations to fully utilize the processor resources. For an issue-4 processor and issue-8 processor, Lev3 and Lev4 transformations provide additional ILP for loops in which unrolling and renaming alone do not expose sufficient ILP.

Induction variable expansion is the most often applied transformation. Almost all unrolled loops contain dependent induction variable computations which may be transformed with induction variable expansion. Accumulator variable expansion and search variable expansion are only applied for certain types of loops. However, these transformations result in the largest speedup increases beyond unrolling and renaming. Tree height reduction and operation combining are consistently applied in many loop bodies to reduce the dependence height within an iteration. Strength reduction is the least effective transformation. However, this is primarily due to the assumed processor model. In this study, integer multiply and divide have short latencies (3 and 10 cycles), and only the issue rate limits the number of multiply/divide instructions which can be issued in a single cycle. With a more restricted processor model, strength reduction is expected to be a more effective transformation. Overall, the average speedup for an issue-4 and issue-8 processor with Lev3 and Lev4 transformations increases from 3.73 to 4.35 and 5.10 to 6.68, respectively.

Effect of Transformations on Register File Usage. The register file usage distribution with each level of compiler transformations is shown in Figure 11 for an issue-8 superscalar/VLIW processor. Again, the number of loops with a particular transformation level which utilizes the specified range of registers is plotted. The register usage reported is the sum of the total integer and floating point registers utilized in the loop nest. The largest increase is due to register renaming. The average number of registers used in all loops increases from 28 to 57 from Lev1 to Lev2 transformations. Lev3 and Lev4 transformations provide only moderate increases in the average number of registers to 65 and 71, respectively. In terms of register usage, Lev3 and Lev4 transformations are relatively efficient transformations to expose ILP. Therefore, a production superscalar/VLIW compiler can make use of Lev3 and Lev4 transformations to achieve a desired level of performance while controlling register usage. The number of registers utilized with all transformations, however, is not unreasonable. With Lev3 and Lev4 transformations, 37 of the 40 loops utilize fewer than 128 total integer and floating point registers.

Effect on DOALL and Non-DOALL Loops. The individual speedup and register file usage distributions of an issue-8 superscalar/VLIW processor for the DOALL loops are shown in Figures 12 and 13. For the DOALL loops, loop unrolling and register renaming expose a large amount of ILP. All iterations of a DOALL loop are independent, therefore unrolling and renaming enable the scheduler to almost completely overlap the execution of multiple loop iterations. Register file usage increases accordingly with unrolling and renaming to account for the large number of temporary values which are live. The tree height reduction and induction variable expansion transformations provide for the performance increases observed with Lev3 and Lev4

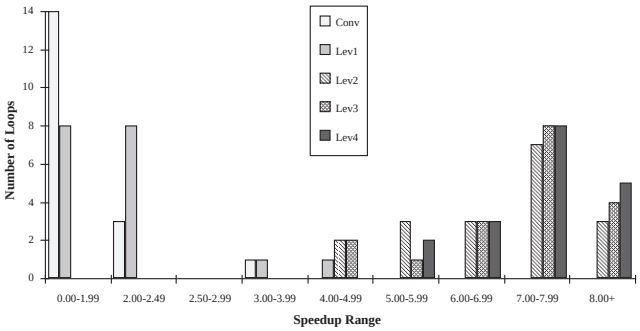

Figure 12: Speedup distribution of DOALL loops only for an issue-8 superscalar/VLIW processor.

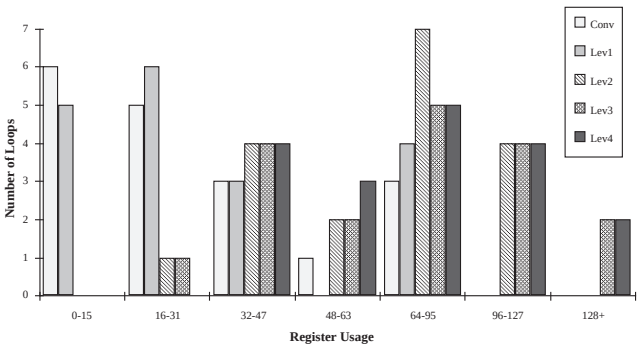

Figure 13: Register usage distribution of DOALL loops only for an issue-8 superscalar/VLIW processor.

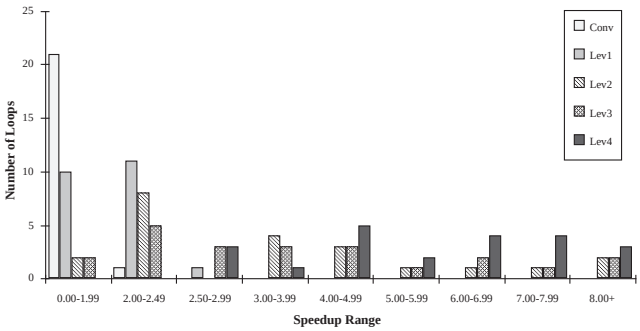

Figure 14: Speedup distribution of non-DOALL loops only for an issue-8 superscalar/VLIW processor.

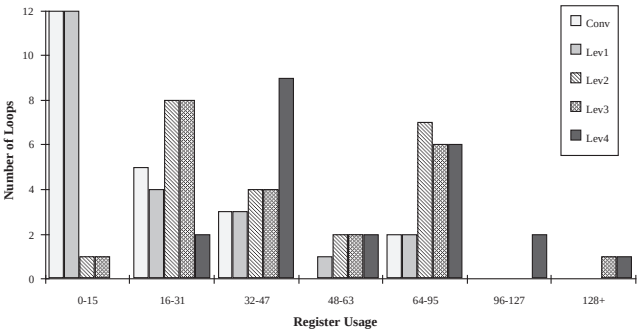

Figure 15: Register usage distribution of non-DOALL loops only for an issue-8 superscalar/VLIW processor.

transformations. Tree height reduction reduces the dependence length within a single iteration of several loops with long chains of arithmetic calculations. Induction variable expansion removes the dependences between address calculations and memory accesses to enable a more complete overlap of unrolled loop bodies. In general, though, transformations beyond loop unrolling and register renaming are not profitable for DOALL loops.

The individual speedup and register file usage distributions of an issue-8 superscalar/VLIW for the non-DOALL loops are shown in Figures 14 and 15. The non-DOALL loops consist of both the DOACROSS and serial loops shown in Table 2. For these loops, loop unrolling and register renaming can only expose limited amounts of ILP. Dependences within each loop iteration and recurrence dependences across loop iterations restrict the amount of available ILP. Lev3 transformations provide small performance improvements over Lev2 transformations. In many cases, the dependences removed with Lev3 transformations are not on the critical path of the loop, and therefore do not impact performance a large amount. Lev4 transformations, on the other hand, provide the largest performance improvements for the non-DOALL loops. For many loops, the expansion transformations remove recurrence dependences on the loops critical path. By removing these recurrence not only is ILP increased, but the effectiveness of other transformations such as operation combining and tree height reduction becomes more apparent with fewer dependences present.

Register usage of the non-DOALL loops is less than that of the DOALL loops due to the reduced overlap among unrolled loop bodies. With the exception of 3 loops, all the non-DOALL loops utilize fewer than 96 total integer and floating point registers with Lev4 transformations.

4 Conclusion

In this paper, it was shown that the amount of ILP available to superscalar or VLIW node processors in a high-performance system can be limited with conventional compiler optimization techniques. The results show that increasing execution resources in superscalar and VLIW node processors yields little performance improvement unless loop unrolling and register renaming are applied. These two transformations are found to expose sufficient ILP for DOALL loops. However, more advanced transformations are required in order for serial and DOACCROSS loops to fully benefit from the increased execution resources. The following advanced transformations were studied and found to satisfy most of this need: accumulator variable expansion, induction variable expansion, search variable expansion, operation combining, strength reduction, and tree height reduction.

With conventional optimization taken as a baseline, loop unrolling and register renaming yields an overall average speedup of 5.1 on an issue-8 processor. Broken down by loop type, the DOALL loops have an average speedup of 6.8 while the serial and DOACROSS loops have an average speedup of 3.7. These speedup numbers verify that these two transformations are sufficient for DOALL loops but that more transformations are needed for serial and DOACROSS loops. Applying the advanced transformations, the DOALL loops average speedup increases to 7.8 and the serial and DOACROSS speedup increases greatly to 5.8.

A side effect of the ILP transformations is to increase the register usage of loops. In this study, an average of 2.6 times more registers are required after all ILP code transformations are applied. However, the total number of registers required by the loops after ILP transformations is not unreasonable. For this study, 37 of 40 loops require fewer than 128 total registers after all transformations.

Acknowledgements

The authors would like to thank Sadun Anik and all members of the IMPACT research group for their comments and suggestions. Special thanks to the anonymous referees whose comments and suggestions helped to improve the quality of this paper significantly. This research has been supported by JSEP under Contract N00014-90-J-1270, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, Matsuhita Electric Industrial Corporation Ltd., Hewlett-Packard, and NASA under Contract NASA NAG 1-613 in cooperation with ICLASS.

References

[1] J. A. Fisher and B. R. Rau, "Instruction-level parallel processing," *Science*, vol. 253, pp. 1233–1241, September 1991.

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

[3] Kuck and Associates, Inc., Champaign, IL, *KAP User's Guide*, November 1988.

[4] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pp. 207–218, January 1981.

[5] T. Nakatani and K. Ebcioglu, "Combining as a compilation technique for VLIW architectures," in *Proceedings of the 22nd International Workshop on Microprogramming and Microarchitecture*, pp. 43–55, September 1989.

[6] K. Anantha and F. Long, "Code compaction for parallel architectures," *Software Practice and Experience*, vol. 20, pp. 537–554, June 1990.

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

[8] 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.

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

[10] J. L. Baer and D. P. Bovet, "Compilation of arithmetic expressions for parallel computations," in *Proceedings of IFIP Congress*, pp. 34–46, 1968.

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

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

[13] M. A. Schuette and J. P. Shen, "An instruction-level performance analysis of the Multiflow TRACE 14/300," in *Proceedings of the 24th International Workshop on Microprogramming and Microarchitecture*, pp. 2–11, November 1991.

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

[15] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318–328, June 1988.

[16] A. Aiken and A. Nicolau, "Optimal loop parallelization," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 308–317, June 1988.

[17] R. M. Stallman, *Using and porting GNU CC*. Free Software Foundation, Inc., 1989.

[18] W. W. Hwu and *et. al*, "The superblock: An effective structure for VLIW and superscalar compilation," tech. rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, February 1992.

[19] M. Berry and *et. al*, "The PERFECT club benchmarks: Effective performance evaluation of supercomputers," Tech. Rep. CSRD-827, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, May 1989.