Improving Balanced Scheduling with Compiler Optimizations that Increase Instruction-Level Parallelism

Jack L. Lo and Susan J. Eggers
Department of Computer Science and Engineering
University of Washington
{jlo,eggers}@cs.washington.edu

Abstract

Traditional list schedulers order instructions based on an optimistic estimate of the load latency imposed by the hardware and therefore cannot respond to variations in memory latency caused by cache hits and misses on non-blocking architectures. In contrast, balanced scheduling schedules instructions based on an estimate of the amount of instruction-level parallelism in the program. By scheduling independent instructions behind loads based on what the program can provide, rather than what the implementation stipulates in the best case (i.e., a cache hit), balanced scheduling can hide variations in memory latencies more effectively.

Since its success depends on the amount of instruction-level parallelism in the code, balanced scheduling should perform even better when more parallelism is available. In this study, we combine balanced scheduling with three compiler optimizations that increase instruction-level parallelism: loop unrolling, trace scheduling and cache locality analysis. Using code generated for the DEC Alpha by the Multiflow compiler, we simulated a non-blocking processor architecture that closely models the Alpha 21164. Our results show that balanced scheduling benefits from all three optimizations, producing average speedups that range from 1.15 to 1.40, across the optimizations. More importantly, because of its ability to tolerate variations in load interlocks, it improves its advantage over traditional scheduling. Without the optimizations, balanced scheduled code is, on average, 1.05 times faster than that generated by a traditional scheduler; with them, its lead increases to 1.18.

1 Introduction

Traditional instruction schedulers order load instructions based on an optimistic assumption that all loads will be cache hits. On most machines, this optimistic estimate is accurate, because the processors block, i.e., stall the pipeline, on cache misses. Blocking processors simplify the design of the code scheduler, by enabling all load instructions to be handled identically, whether they hit or miss in the cache.

The traditional blocking processor model has recently been challenged by processors that do not block on loads [ER94] [McL93] [Asp93] [DA92] [Gwe94a] [Gwe94b] [Mip94] [CS95]. Rather than stalling until a cache miss is satisfied, they use lockup-free caches [Kro81] [FJ94] to continue executing instructions to hide the latency of outstanding memory requests. On these non-blocking architectures, instruction latency is exposed to the compiler and becomes uncertain: not only will the processor see both cache hits and misses, but each level in the memory hierarchy will also introduce a new set of latencies. Instead of handling all loads identically, a non-blocking processor's code scheduler could arrange instructions behind loads more intelligently to produce more efficient code schedules. Unfortunately, a traditional instruction scheduler fails to exploit this opportunity because of its optimistic and architecture-based estimates; its resulting schedules may be therefore intolerant of variations in load latency.

Balanced scheduling [KE93] is an algorithm that can generate schedules that adapt more readily to the uncertainties in memory latency. Rather than being determined by a predefined, architecturally-based value, load latency estimates are based on the number of independent instructions that are available to hide the latency of a particular load. Using these estimates as load weights, the balanced scheduler then schedules instructions normally. Previous work has demonstrated that balanced schedules show speedups averaging 8% for several Perfect Club benchmarks for two different cache hit/miss ratios, assuming a workstation-like memory model in which cache misses are normally distributed [KE93].

Since its success depends on the amount of instruction-level parallelism in a program, balanced scheduling should perform better when more parallelism is available. In this study, we combine balanced scheduling with three compiler optimizations that increase instruction-level parallelism. Two of them, loop unrolling and trace scheduling, do so by giving the scheduler a larger window of instructions from which to select. The third, locality analysis, more effectively utilizes the amount of instruction-level parallelism that is currently available.

Loop unrolling generates more instruction-level parallelism by duplicating loop iterations a number of times equal to the unrolling factor. The technique increases basic block size by eliminating branch overhead instructions for all iterations but the last. Consequently, more instructions are available to hide load latencies and more flexibility is offered to the scheduler. Trace scheduling increases the amount of instruction-level parallelism by permitting instruction scheduling beyond basic blocks [FERN84] [LFK+93]. It does this by assigning expected execution frequencies to edges of the control flow graph, and optimizing the schedules along the execution paths with the highest frequencies. Basic blocks along these paths (or traces) are grouped together, and instructions are scheduled, sometimes speculatively, as though the trace were a single basic block. In order to preserve data dependencies and correctness, some code motion is restricted, or may only occur if additional compensation code is added. The third optimization, locality analysis, identifies potential temporal and spatial reuse in array accesses within a loop and transforms the code to exploit it[MLG92]. When used in conjunction with balanced scheduling, it enables load instructions to be selectively balanced scheduled. If the compiler can determine that a load instruction is a hit, then the optimistic latency estimate is correct, and should be used. This frees more instructions to hide the latencies of other loads (compiler-determined cache misses, as well as loads with no reuse information), which are balanced scheduled.

The original work on balanced scheduling [KE93], which compared it to the traditional approach and without ILP-enhancing optimizations, relied on a stochastic model to simulate cache behavior and network congestion. It included load latencies for a single-level cache hierarchy, but assumed single-cycle execution for all other multi-cycle instructions, and modeled an instruction cache that always hit and an ideal, pipelined floating point unit. By examining the effects of balanced scheduling independent of the particular latencies of other multi-cycle instructions, the simple model allowed us to understand the ramifications of changing just one aspect of code scheduling.

However, executions on a realistic processor and memory architecture are needed to see how well balanced scheduling will perform in practice. Real machines have longer load latencies (i.e., more cycles), both because of their more complete memory system (a cache hierarchy and a TLB) and their faster processor cycle time, relative to the memory system; they also execute instructions that require multiple, fixed latencies. The former should help balanced scheduling relative to traditional scheduling, as long as there is sufficient instruction-level parallelism in the code, because the balanced technique is more likely to hide the additional load latencies. However, the latter, should dilute its benefits.

To study the effects of ILP-enhancing optimizations on balanced scheduling and to evaluate the impact of processor and memory features that affect its ability to hide load latencies, we incorporated balanced scheduling and locality analysis into the Multiflow compiler (which already does loop unrolling and trace scheduling). Using code generated for the DEC Alpha and a simulator that models the AXP 21164, we simulated the effects of all three optimizations on both balanced and traditional scheduling, using Perfect Club [BCK+89] and SPEC92 [Dix92] benchmarks.

Our results confirm the hypothesis. Balanced scheduling interacts well with all three optimizations, producing average speedups that range from 1.15 to 1.40. More importantly, its performance advantage relative to traditional scheduling increases when combined with the optimizations. Without ILP-enhancing optimizations, balanced scheduled programs execute 1.05 times faster than those scheduled traditionally; with them, its performance advantage rises as high as 1.18.

The rest of this paper provides more detail and analysis to support these results. Section 2 briefly reviews the balanced scheduling algorithm and contrasts it to traditional instruction scheduling. Section 3 contains a discussion of the three compiler optimizations, in more detail than was presented above, and with examples for both trace scheduling and locality analysis. Section 4 describes the methodology and experimental framework for the studies. In section 5, we present the results from applying all three compiler optimizations and analyze the factors that account for the performance improvements. Finally, in section 6, we conclude.

2 Balanced Scheduling

Although balanced scheduling is a new algorithm for calculating load instruction weights, it fits cleanly within the framework of a traditional list scheduler. This section describes the behavior of a traditional scheduler, and then explains how balanced scheduling differs.

Instruction scheduling seeks to minimize pipeline stalls due to data and control hazards. Given the data and control dependences present in the instruction stream and the estimates for instruction latencies, the scheduler statically determines an instruction order that avoids as many interlocks as possible. The list scheduling approach [HG83] [GM86] [Sit78] to this minimization problem relies on a code DAG, instruction latencies, and heuristics for instruction selection. The code DAG represents the instructions to be scheduled and the data dependences between them. Instruction latencies (also known as instruction weights) correspond to the number of cycles required before the results of the instructions become available. In traditional list scheduling, these latencies have architecturally-fixed and optimistic values. (For example, as mentioned above, the weight of load instructions is the time of a cache hit.) Given the code DAG and the instruction weights, the scheduler traverses the DAG and generates an ordering for the instructions by applying the set of heuristics for instruction selection, many of which are based on the instruction weights.

The traditional list scheduler can be very effective for an architecture with a blocking processor. In this model, the weights accurately reflect the actual latencies that a processor must hide. Variations in actual load latencies (due to either cache misses in the various levels of the memory hierarchy or congestion in the interconnect) are masked from the compiler, since the processor stalls after a number of cycles equal to the optimistic, architecturally-defined latency.

This is not the case for non-blocking processors. Because the processor is free to continue instruction execution while a load access is in progress, instruction schedulers have an opportunity to hide load latencies by carefully selecting a number of instructions to place behind loads. Traditional schedulers, with their fixed assumptions about the time to execute loads, are unable to take advantage of this opportunity. The balanced scheduler, however, can produce schedules that tolerate variations in memory latency by making no architectural assumptions about load weights.

Instead of using fixed (and architecturally-optimistic) latency estimates for all load instructions, the balanced scheduler varies the instruction latency for loads by basing it on the amount of instruction-level parallelism available in the code. We characterize this measurement as *load-level parallelism*, and we use it for the weight of the load instruction instead of the fixed instruction latency. Working with one basic block at a time, the balanced scheduler calculates the weight for each load separately, as a function of the number of independent instructions that may initiate execution while the load is being serviced and the number of other loads that could hide their latencies with these same instructions. When loads are independent and therefore can be scheduled in sequence without stalling, independent instructions can be used to hide the latencies of them all. For example, in the code DAG of Figure 1, loads L0 and L1 are independent of non-load instructions X1 and X2, and the scheduler could issue them in an L0 L1 X1 X2 order. However, when loads appear in series, there is less load-level parallelism to hide their latencies. For example, X1 and X2 can be used to hide the latency of either L2 or L3, but not both. The overall effect of the balanced scheduler is to schedule the independent instructions behind loads in a balanced fashion, and proportional to their ability to hide the latency of particular loads.

The balanced scheduling approach has three potential advantages over traditional schedulers. First, it hides more latency for loads that have more load-level parallelism available to them. Second, it distributes load-level parallelism across all the loads in order to be more tolerant of memory latency variability. Third, since it bases its load weight estimates on the code rather than the architecture, it produces schedules that are independent of the memory system implementation.

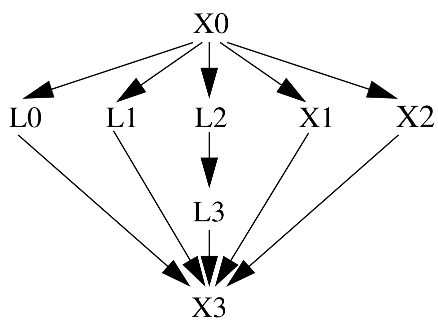

Figure 1

Initial results [KE93] indicated that balanced scheduling produced speedups averaging 8% for selected programs in the Perfect Club suite, when simulated on a stochastic model of a microprocessor with 1990 processor and memory latencies, a bus interconnect and cache hit rates of 80% and 95%. Code for these studies was generated using gcc [Sta].

3 The Compiler Optimizations

Balanced scheduling utilizes load-level parallelism to hide the longer load latencies exposed by non-blocking processors. By applying techniques to increase load-level parallelism, the balanced scheduler should be able to generate schedules that are even more tolerant of uncertain memory latency. In this study, we analyzed the effect of three such techniques: loop unrolling, trace scheduling, and locality analysis.

3.1 Loop Unrolling

Loop unrolling increases the size of basic blocks by duplicating iterations a number of times equal to the unrolling factor. It contributes to increased performance in two ways. First, by creating multiple copies of the loop body, it reduces conditional branch and loop indexing overhead from all but the last copy. Second, the consequent increase in the size of the basic block can expose more instruction-level parallelism, thereby providing more opportunities for code scheduling.

3.2 Trace Scheduling

Trace scheduling [FERN84] [LFK+93] enables more aggressive scheduling by permitting code motion across basic block boundaries. Guided by estimated or profiled execution frequencies for each basic block, it creates traces of paths through each procedure. The trace scheduler then picks a trace, in order of decreasing execution frequencies, and schedules the basic blocks in the trace as if they were a single basic block. Code motion takes into account the effects of scheduling instructions across conditional jumps (splits) and merges (joins), following specific rules for when instructions must be copied or cannot be moved. The final schedule effectively combines into a single block what would have been multiple basic blocks if generated by a traditional scheduler.

Figure 2 illustrates a simple example of trace scheduling. The trace scheduler identifies basic blocks 1, 2, 4, and 5 as a single trace (trace A), and block 3 forms its own trace (trace B). Assume that trace A is scheduled first and therefore is known as the on-trace path (trace B is the off-trace path). Trace A is viewed by the scheduler as a single basic block: it uses traditional basic block code motion, as long as data dependences are maintained. However, in order to move instructions across splits or joins, compensation code may need to be inserted. For example, if an instruction from block 5 is placed in the schedule above the join, then a copy of that instruction must also be placed in the off-trace path, to ensure that the instruction is executed regardless of the path taken. Code motion can also be done speculatively. For example, the scheduler might wish to move an instruction in block 4 above the split in block 2. This operation becomes speculative, because it is not supposed to be executed in the off-trace path. Rather than adding instruction overhead in the off-trace path to undo its effects, speculative motion is restricted to safe operations only, i.e., it is not permitted if the instruction writes memory or sets a variable which is live in the off-trace path.

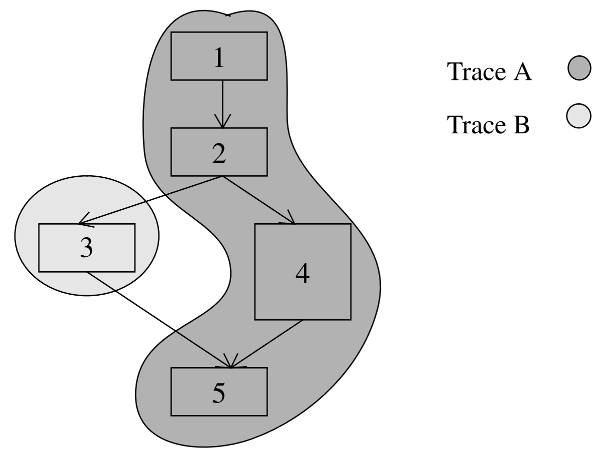

Figure 2

3.3 Locality Analysis

Although memory latency can be uncertain because of cache behavior, this variability is not entirely random. For example, many numeric programs have regular memory access patterns within loops. On different loop iterations, an array reference may access two elements of an array that reside in the same cache line; the first will miss in the cache, but the second will hit because of spatial locality.

If the compiler can determine cache behavior, it can treat cache hits and misses differently. Cache hits can be scheduled using the traditional scheduling scheme, because their actual load latency matches the optimistic, architecture-defined estimate. Then, because we don't need to hide the latency of hits, more independent instructions are available for loads that will miss. These loads, as well as other loads for which locality analysis cannot determine cache behavior (e.g., those that are not array references within loops), can be scheduled via balanced scheduling.

We identify spatial and temporal locality by using the locality analysis algorithm that Mowry, Lam and Gupta developed for selective cache prefetching [MLG92]. We apply their algorithm to determine cache hit and miss behavior for load instructions in inner loops. When the indices of array accesses are linear functions of the loop indices, their algorithm can statically determine the relationship between array references with respect to cache behavior, identifying both spatial and temporal reuse within cache blocks.

Assume that the arrays in our examples (Figures 3-5) are laid out in row-major order. An array reference has spatial reuse if, on consecutive iterations of the inner loop, the reference accesses memory locations that map to the same cache line. For example, in the code sequence of Figure 3, reference A[i][j] has spatial reuse in the inner loop. In order to exploit this locality, however, the scheduler needs to identify the first reference to the cache line as a cache miss and the others as cache hits. This cannot be done in the loop in Figure 3, where there is exactly one load instruction corresponding to the A[i][j] reference. However, if the inner loop is unrolled, the load in the first unrolled copy can be marked as a cache miss, and the loads in the other copies as cache hits.

Figure 3

```
for (i=0; i < n; i++)
    for (j=0; j < n; j++)
        C[i][j] = A[i][j]+B[i][0];
```

Figure 4

```
for (i=0; i < n; i++) {
    for (j=0; j < n-(n%4); j+=4) {
        C[i][j] = A[i][j] + B[i][0];
        C[i][j+1] = A[i][j+1] + B[i][0];
        C[i][j+2] = A[i][j+2] + B[i][0];
        C[i][j+3] = A[i][j+3] + B[i][0];
    }
    if (j < n) {
        C[i][j] = A[i][j] + B[i][0];
        j++;
        if (j < n) {
            C[i][j] = A[i][j] + B[i][0];
            j++;
            if (j < n)  {
                C[i][j] = A[i][j] + B[i][0];
                j++;
            }
        }
    }
}
```

Figure 5

```
for (i=0; i < n; i++) {
    C[i][0] = A[i][0] + B[i][0];
    for (j = 1; j < n; j++)
        C[i][j] = A[i][j] + B[i][0]
}
```

We align the arrays on cache-line boundaries, and assume the 32 byte cache line size of the Alpha 21164 first-level cache. For arrays with double-word elements, each cache line contains four elements of the array. Because of spatial reuse, references to A[i][0], A[i][4], etc., will be cache misses, while references A[i][1], A[i][2], A[i][3], A[i][5], A[i][6], A[i][7], etc., will hit. When the number of loop iterations is unknown statically, we postcondition the loop to ensure that the first unrolled copy still corresponds to a cache miss. This means that the code to handle extra loop iterations is placed after the code for unrolling the loop, as in Figure 4. Note that we cannot simply use another for loop for the extra iterations, because we must be able to mark the load instructions as cache hits or misses.

For temporal reuse, a code transformation is also applied. An array reference has temporal reuse when the reference accesses the same location in different iterations of the loop. In Figure 3, array reference B[i][0] has temporal reuse in the inner loop. The first iteration of the loop will cause a cache miss, while the others will result in hits. By peeling off the first iteration of the loop, we identify the load instruction for B[i][0] in the first iteration as a cache miss. The rest of the loop is unchanged, and since all successive references to B[i][0] in the loop will now be cache hits, we do not need to apply balanced scheduling to its load. Figure 5 shows the peeled version of the original loop of Figure 3.

In Figure 3, the inner loop actually exhibits both types of reuse, so the two transformations would be applied in conjunction. First, the loop would be peeled, and then the remaining iterations would be unrolled.

4 Methodology

To examine the interactions between balanced scheduling and the three compiler optimizations, we ran a set of experiments that measured the effects of individual optimizations on balanced and traditionally scheduled code. We also compared the balanced scheduled code with that produced when combining the optimizations with a traditional list scheduler. The experiments simulate program execution of all code on an architectural model that closely resembles the DEC Alpha 21164. The code was generated using the Multiflow compiler. Section 4.1 describes our workload, section 4.2 the compiler changes and section 4.3 the simulator and the metrics it produced.

Program	Lang.	Description
ARC2D	Fortran	Two-dimensional fluid flow problem solver using Euler equations
BDNA	Fortran	Simulation of hydration structure and dynamics of nucleic acids
DYFESM	Fortran	Structural dynamics benchmark to solve displacements and stresses
MDG	Fortran	Molecular dynamic simulation of flexible water molecules
QCD2	Fortran	Lattice-gauge QCD simulation
TRFD	Fortran	Two-electron integral transformation
alvinn	C	Trains a neural network using back propagation
dnasa7	Fortran	Matrix manipulation routines
doduc	Fortran	Monte Carlo simulation of the time evolution of a nuclear reactor component
ear	C	Simulates the propagation of sound in the human cochlea
hydro2d	Fortran	Solves hydrodynamical Navier Stokes equations to compute galactical jets
mdljdp2	Fortran	Chemical application program that solves equations of motion for atoms
ora	Fortran	Traces rays through an optical system composed of spherical and planar surfaces
spice2g6	Fortran	Circuit simulation package
su2cor	Fortran	Computes masses of elementary particles in the framework of the Quark-Gluon theory
swm256	Fortran	Solves shallow water equations using finite difference equations
tomcatv	Fortran	Vectorized mesh generation program

Table 1: The workload.

4.1 Workload

Our experiments use benchmarks (Table 1) taken from the Perfect Club [BCK+89] and SPEC92 [Dix92] suites (the Multiflow compiler has both Fortran and C front ends). We chose numeric programs because their high percentage of loops make them appropriate testbeds for optimizations targeted for loops. Arrays in the C program are laid out in row-major order; the Fortran benchmarks have a column-major data layout

4.2 The Compiler

By modifying the scheduling module (Phase 3) of the Multiflow compiler, we scheduled instructions by using either a traditional algorithm or balanced scheduling. The list scheduler for the balanced and traditional schedulers compared in this study is top-down, and selects instructions from the ready list in a prioritized order. The priority of an instruction is simply the sum of the instruction's weight and the maximum priority of its successors. Ties are broken by the following heuristics, listed in order of preference. The first heuristic aims to control register pressure by selecting the instruction with the largest difference between consumed and defined registers. The second heuristic selects the instruction that would expose the largest number of successors in the code DAG. The final heuristic selects the instruction that was generated first in the original order of instructions. As another aid in controlling register pressure when applying balanced scheduling, we limited load weights to a maximum of 50.[1] Our version of the Multiflow compiler generates code for DEC Alpha processors.

For the loop unrolling experiments, we selected two unrolling factors, 4 and 8. We disabled loop unrolling when the unrolled block reached 64 instructions (4) or 128 (8), and did not unroll loops with more than one internal conditional branch.[2]

Using the methodology established in other trace scheduling performance studies [FGL93], we first profiled the programs to determine basic block execution frequencies. This information guided the Multiflow compiler in picking traces. Since traces do not cross the back edges of loops, we unrolled them, using the same unrolling factor as in the loop unrolling studies. To gain maximum flexibility of code motion, we also permitted speculative code motion.

[1] In our machine model (described in the next section), the maximum load latency is 50 cycles, and occurs when a load must be satisfied by main memory. Hence, there is no need to hide more than 50 cycles for any load instruction.

We added a locality analysis module to Multiflow's Phase 2. As in Mowry, Lam and Gupta [MLG92], a predicate[3] is created for each array reference that exhibits reuse and is associated with the references's load instruction. Predicates are used to determine both the unrolling factor (which depends on the cache block and array element sizes)[4] and whether loads should be hits or misses. Dependence arcs were added in the code DAG between each miss load and its corresponding hit loads to prevent the latter from floating above the miss during scheduling.

4.3 Simulation Framework

The compiled benchmarks were simulated by using an execution-driven simulator that emulates the DEC Alpha 21164 in most respects. The simulator models the 21164's pipelined functional units, 3-level cache hierarchy, first-level lockup-free cache, both the instruction and data TLBs, branch and computed jump prediction and the instruction and memory hierarchy latencies. (The latter are listed in Tables 2 and 3.) Unlike the 21164, we assume single instruction issue, since we would like to understand fully balanced scheduling's ability to exploit load-level parallelism before applying it to multiple-issue processors that need ILP to execute efficiently.

The simulator produces metrics for execution cycles and number of instructions. Cycle metrics measure total cycles, interlock cycles for both loads and instructions with fixed latencies, and dynamic instruction execution. Instruction counts are obtained for long and short integers, long and short floating point operations, loads, stores, branches, and spill and restore instructions. All met-

[2] Using the Alpha's conditional move instruction, the Multiflow compiler does predicated execution on simple conditional branches. Our limit on the number of conditional branches only affects the conditions within loops that could not be predicated. Loop unrolling can have limited benefit when applied to loops with internal branches, because in the unrolled loop body, the internal branches limit the expansion of basic block size. Unrolling these loops can be more beneficial, however, when trace scheduling is also applied, because of its ability to schedule beyond basic block boundaries.

[3] Predicates contain the loop index, loop depth, stride of access and type of locality.

[4] Recall that locality analysis requires some loop unrolling to differentiate between load references that hit or miss. We only unroll those loops that contain arrays that exhibit reuse.

Memory System Component	Size	Associativity	Block Size (bytes)	Access Latency (hit)	Memory Update Policy
First level, on-chip data cache	8KB	1	32	2	write-through
First level, on-chip instruction cache	8KB	1	32	1	n.a.
Second level, on-chip unified cache	96KB	3	64	8	writeback
Third-level, off-chip unified cache	1MB	4	64	14	writeback
Main memory	--	n.a.	n.a.	50	n.a
TLB, instruction	48 entries	--	--	--	--
TLB, data	64 entries	--	--	--	--

Table 2: Memory hierarchy parameters.

Instruction type	Latency
integer op	1
integer multiply	8
load	2
store	1
FP op (excluding divide)	4
FP div (23 bit fraction)	17
FP div (53 bit fraction)	30
branch	2

Table 3: Processor latencies.

rics are calculated for each optimization, for both scheduling techniques.

5 Effect of the Optimizations

This section details the results of our experiments and provides explanations for our findings. We evaluate each optimization and compare its effects on balanced scheduling relative to the traditional scheduling approach.

5.1 Loop Unrolling

Loop unrolling with an unrolling factor of 4 improved the performance of all balanced scheduled programs, by an average speedup of 1.19 (Table 4). About half of the benefit was due to a 11% decrease in the number of executed instructions (integer and branch instructions related to branch overhead); a fifth was caused by a 19% reduction in interlock cycles due to fixed latency instructions; and the rest was a 23% savings in load interlock cycles. The best speedups were seen for programs whose frequently executed loops could be unrolled the full extent of the unrolling factor; when full unrolling did not occur, improvements were smaller. For example, frequently executed loops in doduc, mdljdp2, spice2g6 and swm256 contained multiple conditionals and consequently were not unrolled. For these benchmarks, the change in dynamic instruction count (column 6 in Table 4) was minimal when unrolling by 4, reflecting the paucity of unrollable loops. Unrolling also had limited effects on the ora benchmark, in which most of the execution time is spent in a large, loop-free subroutine. Finally, in the BDNA benchmark, many of the loops were large enough that the iteration instruction limit for unrolling disabled the optimization.[1] Considering only the other eleven programs for which loop unrolling was universally applied, balanced scheduling was able to eliminate on average 33% of interlock cycles, producing average speedups of 1.29.

Unrolling by 8 brought additional benefit (an average speedup of 1.28 relative to no unrolling), but not as great, and not for all programs. For some programs the more aggressive unrolling could not be completely applied[2]; in most cases, it increased register pressure, and the independent instructions, now relatively fewer in number, were less able to hide the latency of the additional spill loads. Consequently, for most programs, load interlock cycles and

[1] These blocks were large enough to exploit load-level parallelism without loop unrolling, corroborated by the comparison of traditional and balanced scheduling (later in this section), in which BDNA had one of the largest relative reductions in interlock cycles.

Benchmark	Total Cycles (Millions) No Loop Unrolling	Total Cycles Speedup Unroll by 4	Total Cycles Speedup Unroll by 8	Dynamic Instruction Count (Millions) No Loop Unrolling	Dynamic Instruction Count Percentage Decrease Unroll by 4	Dynamic Instruction Count Percentage Decrease Unroll by 8	Load Interlock Cycles (Millions) No Loop Unrolling	Load Interlock Cycles Percentage Decrease Unroll by 4	Load Interlock Cycles Percentage Decrease Unroll by 8
ARC2D	17844.8	1.26	1.55	7704.9	11.4%	14.7%	32.1	34.3%	20.2%
BDNA	4316.3	1.03	1.05	3155.1	0.1%	1.3%	51.6	27.9%	21.1%
DYFESM	1049.5	1.32	1.36	776.4	12.4%	14.6%	19.4	76.8%	72.2%
MDG	24346.2	1.10	1.16	15344	7.9%	11.7%	184.8	11.0%	-12.1%
QCD2	1437.2	1.14	1.17	1169.4	16.8%	18.5%	32.1	34.3%	20.2%
TRFD	3633.1	1.34	1.31	2482.8	17.0%	13.6%	141.2	55.9%	56.1%
alvinn	8290.8	1.25	1.25	5283.8	36.6%	40.3%	707.8	11.3%	10.2%
dnasa7	22029.9	1.76	1.85	8532.4	22.5%	25.6%	1682.5	57.6%	61.1%
doduc	1944.1	1.02	0.99	1164.5	4.1%	4.0%	48.1	0.6%	6.4%
ear	22289.4	1.13	1.34	15776.9	17.1%	31.2%	142.1	-13.6%	-28.9%
hydro2d	12782.3	1.52	1.75	7404.7	20.2%	26.5%	511.8	66.6%	62.5%
mdljdp2	4966.6	1.01	1.01	3490.5	0.4%	0.6%	133	3.0%	2.8%
ora	8183.9	1.00	1.00	5034.3	0.0%	0.0%	0.0	----	----
spice2g6	43659.1	1.01	1.04	24157.2	3.2%	3.9%	12862.5	1.1%	0.3%
su2cor	8032.5	1.10	1.20	6024.7	4.9%	9.4%	294.3	6.2%	27.4%
swm256	20470.8	1.00	1.44	13611.1	0.4%	10.9%	3500.0	1.9%	83.0%
tomcatv	1896.1	1.27	1.26	1232.8	10.2%	11.3%	264.7	21.3%	3.9%
AVERAGE		**1.19**	**1.28**		**10.9%**	**14.0%**		**23.3%**	**26.1%**

Table 4: Balanced scheduling: Speedup in total cycles and percentage decrease in dynamic instruction count and load interlock cycles for unrolling factors of 4 and 8, relative to no unrolling.

Benchmark	BS speedup vs. TS			% reduction in load interlock cycles			% of total cycles due to load interlock cycles					
							No LU		LU 4		LU 8	
	No LU	LU 4	LU 8	No LU	LU 4	LU 8	BS	TS	BS	TS	BS	TS
ARC2D	1.33	1.52	1.78	53.3%	66.7%	77.4%	18.9%	30.3%	15.4%	30.4%	12.3%	30.6%
BDNA	1.01	1.01	1.09	75.8%	80.6%	79.1%	1.2%	4.9%	0.9%	4.5%	1.0%	4.4%
DYFESM	0.90	0.96	0.97	34.5%	83.7%	80.6%	1.8%	3.1%	0.6%	3.6%	0.7%	3.7%
MDG	0.98	0.99	0.99	64.6%	70.0%	65.9%	0.8%	2.2%	0.7%	2.5%	1.0%	2.9%
QCD2	0.98	1.01	0.99	6.4%	35.9%	34.5%	2.2%	2.4%	1.7%	2.6%	2.1%	3.2%
TRFD	0.93	0.96	0.98	51.8%	78.0%	80.8%	3.9%	8.7%	2.3%	10.8%	2.2%	11.8%
alvinn	0.93	0.99	0.99	14.9%	3.2%	3.7%	8.5%	10.8%	9.4%	9.8%	9.6%	10.0%
dnasa7	1.18	1.76	1.84	73.0%	87.2%	88.0%	7.6%	23.9%	5.7%	25.2%	5.5%	24.8%
doduc	1.00	1.00	1.01	44.5%	45.6%	47.4%	2.5%	4.5%	2.5%	4.6%	2.3%	4.3%
ear	0.93	0.95	0.95	44.2%	39.9%	28.2%	0.6%	1.2%	0.8%	1.4%	1.1%	1.6%
hydro2d	0.99	1.07	1.11	68.4%	88.7%	87.0%	4.0%	12.8%	2.0%	17.0%	2.6%	18.2%
mdljdp2	1.03	1.03	1.03	54.0%	55.4%	55.3%	2.7%	5.7%	2.6%	5.7%	2.6%	5.7%
ora	1.00	1.00	1.00	-----	-----	-----	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
spice2g6	1.02	1.01	1.01	7.5%	8.5%	8.0%	29.5%	31.4%	29.5%	32.1%	30.5%	32.7%
su2cor	1.18	1.22	1.26	87.8%	88.8%	90.4%	3.7%	25.3%	3.8%	27.5%	3.2%	26.5%
swm256	1.22	1.22	1.63	67.8%	68.4%	94.4%	17.1%	43.5%	16.8%	43.6%	4.2%	45.6%
tomcatv	1.22	1.36	1.37	72.4%	75.9%	72.7%	14.0%	41.3%	14.0%	42.9%	16.9%	45.3%
AVERAGE	**1.05**	**1.12**	**1.18**	**51.3%**	**61.0%**	**62.1%**	**7.0%**	**14.8%**	**6.4%**	**15.5%**	**5.8%**	**16.0%**

Table 5: Balanced scheduling (BS) vs. traditional scheduling (TS) for loop unrolling: Total cycles speedup, percentage improvement in load interlock cycles, and load interlock cycles as a percentage of total cycles.

the dynamic instruction count continued to drop, but by less (on average, another 2.8% for load interlock cycles, for a total reduction of 26%, and another 3% for dynamic instruction count, for a total reduction of 14%). The three exceptions were TRFD, doduc and tomcatv, for which overall performance dropped relative to unrolling by 4. In TRFD the increase in spill instructions offset the reduction in branch overhead; in doduc, the more aggressive unrolling increased code size, contributing to a degradation in instruction cache performance; and in tomcatv, interlock cycles increased because of the increase in spill code.

The more important loop unrolling results, however, were found when comparing balanced and traditional scheduling. The combination of balanced scheduling and loop unrolling produced better average speedups than coupling loop unrolling with traditional list scheduling. Without loop unrolling, balanced scheduled codes executed on average 1.05 times faster than the traditional approach (Table 5); with it, balanced scheduling's advantage rose to 1.12 when unrolling by 4, and to 1.18 when by 8. (In other words, in contrast to balanced scheduling's 1.19 and 1.28 average improvements when unrolling by 4 and 8, traditional scheduling provided speedups of only 1.12 and 1.15.) In all cases (all programs and unrolling optimization levels) dynamic instruction counts between the two scheduling approaches were almost identical. Balanced scheduling's advantage stemmed entirely from its ability to hide load latencies: in particular, the balanced code induced an average of 51%, 61%, and 62% fewer load interlock cycles than traditional scheduling when unrolling 0, 4, and 8 times, respectively (again, Table 5). The result was that only 7%, 6% and 6% of total execution cycles were lost to load interlocks, while traditional scheduling's penalty was 15%, 16% and 16%.

Although balanced scheduling is, on average, a better match than traditional list scheduling with techniques that increase basic block size, such as loop unrolling, the average speedups mask a wide divergence in the programs' ability to benefit from it. For particular benchmarks, traditional scheduling even outperformed balanced scheduling. In these programs, balanced scheduling continued to be more successful at reducing load interlock cycles. However, it did this at the expense of hiding interlock cycles caused by multi-cycle instructions with **fixed** latencies, such as integer multiply and floating point instructions. The traditional scheduler, on the other hand, gave preference to the fixed latency operations.

The benchmarks in question exhibited two characteristics that limited the effectiveness of balanced scheduling. First, they had smaller basic blocks, and hence, less load-level parallelism. Second, as a percentage of total cycles, load interlock cycles were small and were outnumbered by non-load interlock cycles. In these cases, when non-load interlock cycles dominated total interlock cycles and insufficient parallelism existed, balanced scheduling sacrificed hiding non-load interlocks to hide the latency of load instructions. When more parallelism was exposed by unrolling loops, the frequency and magnitude of these performance degradations were reduced. Nonetheless, the results argue for either incorporating multi-cycle instructions with fixed latencies into the balanced scheduling algorithm or developing heuristics to statically choose between the two schedulers on a basic block basis.

[2] There were exceptions, however. For example, the 64 instruction limit on unrolling by 4 prevented swm256 from being fully unrolled; the higher limit with an unrolling factor of 8 allowed more unrolling and therefore provided more benefit.

Benchmark	Speedup over BS alone: TrS	LU 4, TrS	LU 8, TrS	LA	LU 4, LA	LU 8, LA	LU 4, TrS, LA	LU 8, TrS, LA
ARC2D	1.02	1.35	1.72	1.17	1.38	1.54	1.42	1.67
BDNA	1.02	1.03	1.07	---	---	---	---	---
DYFESM	0.98	0.44	0.47	1.09	1.31	1.32	1.21	1.21
MDG	1.04	1.11	1.13	1.15	1.16	1.17	1.27	1.27
QCD2	1.01	1.18	1.20	1.04	1.16	1.17	1.23	1.19
TRFD	0.99	1.55	1.49	1.34	1.36	1.36	1.44	1.45
alvinn	0.97	1.24	1.25	1.01	1.25	1.25	1.24	1.25
dnasa7	1.11	1.76	2.19	1.20	1.75	1.83	1.77	2.11
doduc	1.01	1.10	0.56	0.97	0.99	0.99	0.67	---
ear	0.96	1.20	1.41	1.03	1.12	1.31	1.24	1.41
hydro2d	1.02	1.55	1.76	1.44	1.79	1.82	1.79	1.95
mdljdp2	0.88	0.89	0.89	1.00	1.00	1.00	0.89	0.88
ora	1.06	1.06	1.06	1.00	1.00	1.00	1.06	1.06
spice2g6	1.05	1.18	1.14	1.02	1.02	1.04	1.12	1.12
su2cor	0.95	1.09	1.25	1.13	1.17	1.14	---	---
swm256	1.01	1.14	1.48	1.23	1.60	1.60	1.65	1.65
tomcatv	1.01	1.29	1.34	1.53	1.39	1.39	1.40	1.40
AVERAGE	**1.01**	**1.19**	**1.26**	**1.15**	**1.28**	**1.31**	**1.29**	**1.40**

Table 6: Speedups over balanced scheduling alone for combinations of loop unrolling by 4 and 8 (LU 4 and LU 8), trace scheduling (TrS) and locality analysis (LA).

	Speedup of Balanced Scheduling over Traditional Scheduling				
	No trace scheduling			Trace scheduling	
Benchmark	No LU	LU 4	LU 8	LU 4	LU 8
ARC2D	1.33	1.52	1.78	1.62	1.99
BDNA	1.01	1.01	1.09	1.09	1.13
DYFESM	0.90	0.96	0.97	0.85	0.85
MDG	0.98	0.99	0.99	0.97	0.97
QCD2	0.98	1.01	0.99	0.96	0.99
TRFD	0.93	0.96	0.98	1.01	1.01
alvinn	0.93	0.99	0.99	0.97	0.98
dnasa7	1.18	1.76	1.84	1.83	2.17
doduc	1.00	1.00	1.01	1.06	1.02
ear	0.93	0.95	0.95	0.91	0.92
hydro2d	0.99	1.07	1.11	1.09	1.06
mdljdp2	1.03	1.03	1.03	0.91	0.91
ora	1.00	1.00	1.00	1.00	1.00
spice2g6	1.02	1.01	1.01	1.14	1.06
su2cor	1.18	1.22	1.26	---	---
swm256	1.22	1.22	1.63	1.37	1.64
tomcatv	1.22	1.36	1.37	1.38	1.43
AVERAGE	**1.05**	**1.12**	**1.18**	**1.14**	**1.16**

Table 7: Balanced scheduling (BS) vs. traditional scheduling (TS): Total cycles speedup for loop unrolling alone and trace scheduling and loop unrolling

5.2 Trace Scheduling

Trace scheduling alone brought little benefit for this workload (Table 6, column 2). Three factors limited its effectiveness. First, traces do not cross back edges of loops, and therefore trace scheduling does not schedule beyond loop boundaries. Since the bulk of execution time in our applications was spent in loops, there was little opportunity to apply it effectively. Second, because it optimizes the most frequently executed traces at the expense of less frequently executed paths, trace scheduling is most effective when there are dominant paths of execution. DYFESM and doduc, in particular, have few dominant paths, and consequently trace scheduling had either limited effectiveness or even degraded performance. Third, programs such as BDNA already had extremely large basic blocks, and therefore trace scheduling was not needed to increase available parallelism. For these reasons, when evaluating the interaction between trace scheduling and the two code scheduling techniques, we first unrolled the loops.

Averaged across the entire workload, the speedups of trace scheduling and loop unrolling were 1.19 with an unrolling factor of 4, and 1.26 when unrolling by 8 (again, Table 6).[1] For programs where trace scheduling could accurately pick traces and string together several nonloop basic blocks, the wins were larger, on average 1.29 and 1.43 for unrolling by 4 and 8, respectively. Included in this group were programs for which loop unrolling alone had been less beneficial; by effectively increasing basic block size in both the loops with conditionals and the sequential code, trace scheduling improved their performance. Consequently, the combination of loop unrolling and trace scheduling sometimes brought benefits to balanced scheduling that were greater than the sum of the individual effects.

As with the loop unrolling experiments, the comparison to traditional scheduling came out in balanced scheduling's favor (Table 7). When combined with trace scheduling and an unrolling factor of 4, balanced scheduled programs executed 1.14 times faster than traditional scheduling with the same optimizations. The comparable speedup for trace scheduling and an unrolling factor of 8 was 1.16. As before, the improvements stemmed from balanced scheduling's ability to hide load interlock cycles more effectively: their decrease rose to 65% and 56%, resulting in only 5% of total execution cycles wasted on load interlocks for the two degrees of unrolling. Traditional scheduling with trace scheduling and unrolling by 4 and 8, on the other hand, was still left with 15% of total cycles lost to load interlocks. On an individual basis, however, traditional scheduling was sometimes superior to balanced scheduling, as in some of the loop unrolling experiments, because non-load interlock cycles were hidden more effectively.

As a final note, we observe that the overall performance gains due to trace scheduling are smaller than those obtained in the original trace scheduling studies. This discrepancy can be attributed to the differing machine models. Trace scheduling was first applied to VLIW machines, where instruction issue bandwidth was not a bottleneck. However, in our single-issue model, where instruction issue bandwidth can be a scarce resource, speculative execution, with its increased instruction count, can stress it.

[1] Because of speculative execution and poor trace selection, the dynamic instruction count for DYFESM more than doubled for both balanced and traditional scheduling on a single-issue machine. On a wider-issue machine (superscalar or VLIW), this effect would be smaller, because of the increased instruction issue bandwidth. If we had excluded DYFESM from the results for balanced scheduling plus trace scheduling and loop unrolling, speedups would have been 1.23 (4) and 1.31 (8). (See also the discussion at the end of this section.)

5.3 Locality Analysis

In contrast to the first two optimizations, which increase the amount of instruction-level parallelism in the code, locality analysis allows the balanced scheduler to exploit the existing parallelism more effectively. Locality analysis should improve performance for two inter-related reasons. First, by relying on more accurate information about memory latency (cache hits and misses), less uncertainty should exist, and the scheduler will presumably generate a better schedule. Specifically, since we do not apply balanced scheduling to load instructions that are cache hits, independent instructions that would normally be used to hide their latency can be applied to loads that will miss. Second, since the spatial locality analysis relies on (limited) loop unrolling, basic block sizes should increase, consequently giving the balanced scheduler more flexibility in using independent instructions to hide load latencies. Taken together, these effects should decrease interlock cycles over applying balanced scheduling alone.

Program speedups from locality analysis averaged 1.15 (Table 6), with the percentage gain divided almost evenly between decreases in dynamic instruction count (branch overhead and spill code) and non-load interlock cycles[2]. The gains were less than those achieved by the optimizations that increased basic block size (loop unrolling and loop unrolling coupled with trace scheduling), because there were fewer opportunities where locality analysis could be applied. The most striking exception was tomcatv, which has very sequential accesses to large, read-only arrays: speedup for this program was 1.5. In general, however, four factors limited the effectiveness of the locality analysis algorithm. First, in order to exploit locality, the compiler must be able to determine how an array is aligned relative to cache lines. When subsections of arrays are passed as parameters to a subroutine, this is not possible. The second factor is the presence of index expressions that introduce irregularity into the memory reference pattern. When array references contain index expressions that are not exclusively composed of affine functions of loop induction variables, reuse cannot be identified statically. Third, some benchmarks pass multi-dimensional arrays as parameters to subroutines, but the array dimensions are dependent on other subroutine parameters. The cache alignment characteristics of these arrays are therefore unknown at compile time. Fourth, even though a load may be identified as a cache hit, it may actually turn out to be a cache miss because of interference with another load reference.

5.4 "Putting It All Together"

Considering these results across optimizations (rather than across programs for a particular optimization, as we have done previously), balanced scheduled code consistently produced speedups over that generated by traditional scheduling. Its performance improvements stemmed from its ability to hide load interlocks more effectively. With only two exceptions, balanced scheduled code produced fewer load interlocks than that of the traditional scheduler for all programs, on all levels of optimization[3]. The differences ranged from two to three times as many interlocks (averaged across all programs for a particular optimization) for the traditional scheduler (see summary results in Table 8).

[2] The bulk of load interlock cycles had been reduced by balanced scheduling.

[3] except locality analysis. Since traditional scheduling relies on a single load latency, it has no counterpart in locality analysis.

Optimizations in addition to balanced scheduling	Relative to traditional scheduling with the same optimization: Program speedup	Relative to traditional scheduling with the same optimization: Percentage decrease in load interlock cycles	Relative to balanced scheduling with no other optimizations: Program speedup	Relative to balanced scheduling with no other optimizations: Percentage decrease in load interlock cycles	Load interlock cycles remaining after applying the optimization: (% of total cycles) Balanced scheduling	Load interlock cycles remaining after applying the optimization: (% of total cycles) Traditional scheduling
No optimizations	1.05	51	n.a.	n.a.	7	15
Loop unrolling by 4	1.12	61	1.19	23	6	16
Loop unrolling by 8	1.18	62	1.28	26	6	16
Trace scheduling with loop unrolling by 4	1.14	65	1.19	42	5	15
Trace scheduling with loop unrolling by 8	1.16	56	1.26	34	5	15

Table 8: Summary comparison of balanced scheduling and traditional scheduling.

Optimizations	Speedup relative to locality analysis alone	Speedup relative to balanced scheduling with no unrolling and no trace scheduling
Locality analysis	n.a.	1.15
Locality analysis with loop unrolling by 4	1.11	1.28
Locality analysis with loop unrolling by 8	1.14	1.31
Locality analysis with trace scheduling and loop unrolling by 4	1.12	1.29
Locality analysis with trace scheduling and loop unrolling by 8	1.21	1.40

Table 9: Summary comparison of locality analysis results.

As we applied optimizations that increased instruction-level parallelism more aggressively (i.e., no optimization through trace scheduling and loop unrolling with an unrolling factor of 8), balanced scheduling was able to extend its advantage over traditional scheduling by exploiting the additional instruction-level parallelism. The percentage of load interlock cycles in balanced scheduled code dropped, while in traditionally scheduled code it remained constant or rose slightly. The consequence was a consistent rise in speedup of balanced scheduling over the traditional approach.

Locality analysis contributed additional speedup when applied along with the other optimizations. When used with loop unrolling, speedups of 1.28 and 1.31 were obtained over balanced scheduling alone, for unrolling factors of 4 and 8, respectively (Tables 6 and 9). When trace scheduling was also applied, these speedups reached 1.29 and 1.40.

5.5 Simulating Real Architectures

Examining results from the original comparison of balanced and traditional scheduling (without optimizations) [KE93] illustrates both the limitations of simple architecture models and the benefits of a very optimizing compiler for execution cycle analysis. There were several methodological differences between the two studies. First, the original work relied on a stochastic model, rather than execution-driven simulation, to simulate cache behavior and network congestion. Second, it included first-level cache load latencies, but assumed single-cycle execution for all other multi-cycle instructions, modeling an instruction cache that always hit and an ideal, pipelined floating point unit, and excluding a cache hierarchy and TLB. This work incorporates all factors; fixed, multi-cycle instruction latencies, in particular, proved to have a marked effect on code scheduling performance. Finally, the previous study used code generated by gcc, which omits several optimizations, such as predicated execution and array dependence analysis to disambiguate between loads and stores, that benefit code scheduling by exposing more load-level parallelism. These optimizations are included in the Multiflow compiler. For the four programs that both studies have in common, we estimate that balanced scheduling had a 10% advantage over traditional scheduling with the simple model, but only 4% when modeling the 21164. Recall that the 21164 has a longer memory latency, ranging from 2 to 50 cycles, depending on the memory hierarchy level, and multi-cycle floating point and integer multiply instructions. Therefore there are more latencies of several types to hide, some of which (the fixed latency arithmetic operations) balanced scheduling does not yet address. Despite these differences, this study validates the earlier conclusion that balanced scheduling is on average superior to traditional scheduling.

6 Conclusion

Modern processors with lockup-free caches provide new opportunities for better instruction scheduling by exposing the variations in load latency due to the memory hierarchy. Unfortunately, traditional scheduling techniques cannot take advantage of this opportunity, because they assume that load instruction weights are fixed, best-case latency estimates. Balanced scheduling, on the other hand, is able to hide variable latencies by basing load instruction weights on a measure of load-level parallelism in the program, rather than using the fixed and optimistic latency estimates.

Our studies have shown that the balanced scheduler can increase its advantage over a traditional scheduler when more instruction-level parallelism is available. By applying loop unrolling, trace scheduling, and locality analysis in conjunction with the balanced scheduler, load interlock cycles were reduced to as low as 5% of total cycles when averaged across an entire workload. In contrast, traditionally scheduled code wasted no less than 15% of its execution cycles waiting for load instructions, when the same optimizations were applied. The ability of the balanced scheduler to hide load latency translated into significant performance gains. Average speedups of balanced scheduled code over traditionally scheduled code increased from 1.05 without the ILP-enhancing optimizations to as much as 1.18 when they were used.

Of all three optimizations, loop unrolling produced the most significant benefits for balanced scheduled code, achieving performance improvements of 1.19 and 1.28 over no unrolling. Relative to traditional scheduling, balanced scheduling increased its edge from from 1.05 (no unrolling) to 1.12 (unrolling by 4) to 1.18 (unrolling by 8).

While loop unrolling increases parallelism within loop bodies, trace scheduling exposes it in unrolled loops that contain conditionals and in the sequential portions of code. However, since loops dominate execution in our workload, trace scheduling was only beneficial when used in conjunction with loop unrolling. The combination produced balanced schedules that executed 1.19 times faster when unrolling by 4, and 1.26 times faster when unrolling by 8, over balanced scheduling alone. Furthermore, they increased balanced scheduling's lead relative to traditional scheduling to 1.14 (4) and 1.16 (8).

Locality analysis provided a performance improvement of 1.15 over balanced scheduling alone. The figure reflects both the relatively small amount of array reuse in our workload and the benefits of the limited loop unrolling that is required to implement the algorithm. Combining locality analysis with the other optimizations improved the speedups. The best case, of course, occurred when unrolling by 8, trace scheduling, and locality analysis were all used: here average speedups reached 1.40.

Although our results demonstrate that the balanced scheduler is more effective than traditional scheduling in taking advantage of the additional instruction-level parallelism generated by the three optimizations for most programs, there are cases where traditional scheduling does better. Because it uses optimistic estimates for load latency, it saves instruction-level parallelism for multi-cycle operations. When (fixed latency) multi-cycle operations dominate loads and there is insufficient instruction-level parallelism to hide instruction latencies, traditionally scheduled code can execute faster.

Having shown that balanced scheduling is a good match with optimizations that increase instruction-level parallelism, we intend to examine its effects on wider-issue (superscalar) processors that require considerable instruction-level parallelism to perform well. We also plan to investigate new techniques to better handle fixed, non-load interlock cycles within the framework of the balanced scheduling algorithm.

Acknowledgments

We would like to thank John O'Donnell from Equator Technologies, Inc., Michael Adler, Tryggve Fossum and Geoff Lowney from Digital Equipment Corp., Stefan Freudenberger from HP Labs, and Robert Nix from Silicon Graphics Inc. for access to the Multiflow compiler source and technical advice in using and altering it; both were invaluable. We also thank Dean Tullsen for the use of his Alpha simulator. This research was sponsored by NSF PYI Award #MIP-9058-439, ARPA contract #N00014-94-1136, and Digital Equipment Corporation.

References

[Asp93] T. Asprey. Performance features of the PA7100 microprocessor. *IEEE Micro*, 13(3):22–35, June 1993.

[BCK+89] M. Berry, D. Chen, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, O. Johnson F. Seidl and, R. Goodrum, and J. Martin. The Perfect Club: Effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, December 1989.

[CS95] R.P. Colwell and R.L. Steck. A 0.6um BiCMOS Processor with Dynamic Execution. In *IEEE International Solid-State Circuits Conference '95*, page 176-177, February 1995.

[DA92] K. Diefendorf and M. Allen. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 12(2):51–61, April 1992.

[Dix92] K.M. Dixit. New CPU benchmark suites from SPEC. In *COMPCON*, pages 305–310, February 1992.

[ER94] J. Edmondson and P. Rubinfeld. An overview of the 21164 Alpha AXP microprocessor. In *Hot Chips VI*, pages 1–8, August 1994.

[FERN84] J.A. Fisher, J.R. Ellis, J. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. In *ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 36–46, June 1984.

[FGL93] S.M. Freudenberger, T.R. Gross, and P.F. Lowney. Avoidance and suppression of compensation code in a trace scheduling compiler. Technical Report HPL-93-35, Hewlett Packard, 1993.

[FJ94] K.I. Farkas and N.P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *21th Annual International Symposium on Computer Architecture*, pages 211–222, April 1994.

[GM86] P.B. Gibbons and S.S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 11–16, July 1986.

[Gwe94a] L. Gwennap. UltraSparc Unleashes SPARC Performance. *Microprocessor Report,* pages 1–8, October 3, 1994.

[Gwe94b] L. Gwennap. 620 Fills Out PowerPC Product Line. *Microprocessor Report,* pages 12–16, October 24, 1994.

[HG83] J.L. Hennessy and T.R. Gross. Postpass code optimization of pipeline constraints. ACM Transactions on Programming Languages and Systems, 5(3):422-448, July 1983.

[KE93] D.R. Kerns and S.J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 278-289, June 1993.

[Kro81] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *8th Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[LFK+93] P.F. Lowney, S.M. Freudenberger, R.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnell, and J.C. Ruttenberg. The Multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7:51–143, 1993.

[McL93] E. McLellan. The Alpha AXP architecture and 21064 processor. *IEEE Micro*, 13(3):36–47, June 1993.

[Mip94] MIPS Technologies, Inc. Mips Open RISC Technology R10000 Microprocessor Technical Brief. October 1994.

[MLG92] T.C. Mowry, M.S. Lam and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[Sit78] R.L. Sites. Instruction Ordering for the Cray-1 Computer. Technical Report 78-CS-023, Univ. of California, San Diego, July 1978.

[Sta] R. Stallman. *The GNU Project Optimizing C Compiler*. Free Software Foundation.