Balanced Scheduling: Instruction Scheduling When Memory Latency is Uncertain

Daniel R. Kerns† and Susan J. Eggers
Department of Computer Science and Engineering
University of Washington, FR-35
Seattle, WA 98195
E-Mail: kerns@pure.com & eggers@cs.washington.edu

Abstract

Traditional list schedulers order instructions based on an optimistic estimate of the load delay imposed by the implementation. Therefore they cannot respond to variations in load latencies (due to cache hits or misses, congestion in the memory interconnect, etc.) and cannot easily be applied across different implementations. We have developed an alternative algorithm, known as balanced scheduling, that schedules instructions based on an estimate of the amount of instruction level parallelism in the program. Since scheduling decisions are program- rather than machine-based, balanced scheduling is unaffected by implementation changes. Since it is based on the amount of instruction level parallelism that a program can support, it can respond better to variations in load latencies. Performance improvements over a traditional list scheduler on a Fortran workload and simulating several different machine types (cache-based workstations, large parallel machines with a multipath interconnect and a combination, all with non-blocking processors) are quite good, averaging between 3% and 18%.

1 Introduction

Instruction schedulers for conventional machines generate code assuming a machine model in which load latencies are well-defined and fixed. Usually the latencies reflect the most optimistic execution situation, e.g., the time of a cache hit rather than a cache miss. Compiler optimizations intended to improve performance through instruction scheduling, such as reordering instructions to avoid pipeline stalls, insert independent instructions after loads to keep the CPU busy while memory references are in progress. The number of instructions inserted (in the best case) depends on this predefined latency value.

When a load reference must exceed the implementation-defined latency, the processor architecture generally stipulates that instruction execution be stalled. The advantage of this design (called *blocking* loads) is that it requires a simple and straightforward hardware implementation. The consequence for compiler technology is that the compiler does not have to consider multiple memory latencies during instruction scheduling.

Two architectural innovations make it worthwhile to reconsider how to schedule behind load instructions. The first is processor designs that do not stall on unsatisfied load references (called *nonblocking* loads) through the use of lockup free caches[17, 18, 15, 13], multiple hardware contexts[2, 1] or an instruction lookahead scheme[2]. Nonblocking loads allow a processor to continue executing other instructions while a load is in progress. Although the design requires more complex hardware, more instruction level parallelism can be exploited, and therefore programs execute faster. The second innovation is machines that have a large variance in memory response time. They may be due to congestion in a multipath interconnect or a hierarchy of memory, including both cache hierarchies and local and global memories.

Variable load instruction latencies, coupled with non-blocking loads, complicate scheduling, because the instruction scheduler does not know how many instructions to schedule after a load to maintain high processor utilization. If the memory reference is delayed beyond the scheduler's latency estimate, the processor will stall and processor utilization will drop. However, if the load latency is shorter than the estimate, the destination register of a load instruction will be tied up longer than necessary. This may increase register pressure enough to cause unnecessary spills to memory and a consequent increase in program execution time. In addition, an excessive number of instructions may migrate

This research was supported by NSF Grant No. CCR-9114167, NSF PYI Award No. MIP-9058-439, ONR Grant No. N00014-92-J-1395 and the Washington Technology Center.

†Author's current address: Pure Software, 1309 S. Mary Ave, Sunnyvale, CA 94087

to the top of the schedule, leaving an insufficient number to hide load latencies near the bottom. In this case the CPU will also be needlessly idled.

In this paper we present a code scheduling algorithm, called *balanced scheduling*, that has been specifically designed to tolerate a wide range of variance in load latency over the entire execution of a program. Balanced scheduling works within the context of a traditional list scheduler[9, 14, 21, 8, 6], but uses a new method for calculating load instruction weights. Rather than using weights that are determined by the implementation and therefore are fixed for all programs, the weight of each load is based on the amount of instruction level parallelism that is available to it. (We refer to this as *load level parallelism.*) This assignment is effective, since load instructions are scheduled for the maximum latency that can be sustained by the amount of load level parallelism in the code. In essence, our algorithm schedules for the code instead of scheduling for the machine. Looking at it another way, balanced scheduling amortizes the cost of incorrectly estimating actual load latencies over all load instructions in the program.

To validate the algorithm we compared the performance of several programs scheduled via balanced scheduling and a traditional list scheduler on a variety of processor and memory architectures. The processor models differed in their ability to exploit load level parallelism; each was coupled with three different memory systems, that exhibit dissimilar latency behavior. Both the balanced scheduler and the traditional scheduler were incorporated into the GCC[19] compiler and generated code for the Perfect Club benchmarks[4]. Performance improvements for balanced scheduling averaged 3% to 18% over the traditional list scheduler, for different processor and system model combinations.

The remainder of this paper is organized as follows. Section 2 introduces balanced scheduling, and section 3 describes the algorithm in more detail. Section 4 explains our experimental methodology; section 5 presents the experimental results. Section 6 discusses extensions and other applications of the balanced scheduling algorithm. A summary follows in section 7.

2 Balanced Scheduling

The traditional approach to instruction scheduling that considers machine resource constraints is *list scheduling*[9, 14, 21, 8, 6]. The primary data structure used by list schedulers is the *code DAG*, in which nodes represent instructions and edges represent dependences between them. Each node is labeled with a weight reflecting the latency of the instruction.[1] At each iteration of its algorithm a list scheduler creates a ready list of instructions that are eligible for scheduling, i.e., those whose predecessors in the code DAG have been scheduled or have had their latencies met. A set of heuristics is then applied to decide which instruction from the ready list should be scheduled next; the heuristics used depend on the particular list scheduler. For example, Gibbons and Muchnick[8] first schedule the instruction with the greatest operation latency. If more than one instruction qualifies, their scheduler breaks the tie by choosing the instruction(s) with the greatest number of successors. The final heuristic picks the instruction with the largest sum of the latencies along the longest path from the instruction node to a leaf node. Other styles of list schedulers include those that combine several levels of heuristics into a single weight and schedule in decreasing weight order[16, 22] and update scheduling weights dynamically[21]. Our heuristics are described in detail in Section 4.1.

If a processor exposes the variations in actual memory reference latency to the compiler through non-blocking load instructions, instruction scheduling becomes more complicated. Traditional list schedulers use a single constant for the weight of all load instructions, usually an implementation-defined latency (e.g., cache hit time). They then schedule instructions independent of that load until the load latency has been consumed. As expected, traditional schedulers work best when the actual latency of each load matches the predefined (and optimistic) value. When it does not, a longer latency (e.g., the time of a cache miss) penalizes the program by stalling the CPU. This fixed estimate of memory latency prevents the scheduler from hiding latencies larger than the nominal value. Therefore, when the optimistic execution scenario does not occur, performance suffers. The worst scheduling situation exists when the actual latencies change over time, for example, as congestion in the interconnect varies.

In contrast, the balanced scheduler computes load instruction weights based on a measure of instruction level parallelism in the code rather than on an implementation-defined value. This measure, which we call *load level parallelism*, defines the number of instructions that may execute in parallel with each load instruction. The weight for each load is calculated separately, as a function of the number of instructions that may initiate execution during the load and the number of other loads that could also use them to hide latencies.

Both the balanced scheduling algorithm and the traditional scheduler operate on a basic block by basic block basis. The balanced scheduler simply incorporates the new method of computing weights for each load instruction into a traditional list scheduler.

Using the code DAG of Figure 1 as an example, Figure 2 illustrates the schedules generated by the traditional and the balanced schedulers. Nodes labeled Ln represent load instructions and nodes labeled Xn represent other non-load instructions of weight 1. The schedules in Figures 2a and 2b result from scheduling the graph of Figure 1 with a tra-

[1] Edges can also be labeled, allowing latencies to differ among successor nodes of a given node, as on the Intel i860.

$$\begin{tikzcd}
 & \text{Begin} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{L0} \arrow[d] & \text{X0} \arrow[d] & \text{X2} \arrow[d] \\
\text{L1} \arrow[dr] & \text{X1} \arrow[d] & \text{X3} \arrow[dl] \\
 & \text{X4} &
\end{tikzcd}$$

Figure 1: Code DAG of a hypothetical program.

Traditional Scheduling	Traditional Scheduling	Balanced Scheduling
W = 5	W = 1	
L0	L0	L0
X0	L1	X0
X1	X0	X1
X2	X1	L1
X3	X2	X2
L1	X3	X3
X4	X4	X4
(a)	(b)	(c)

Figure 2: Schedules generated from the code DAG in Figure 1, using the traditional and balanced schedulers. The traditional scheduler is illustrated with load instruction weights of 5 and 1, respectively.

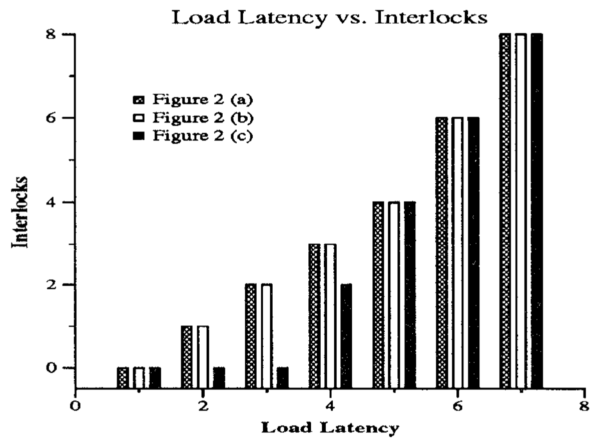

Figure 3: Interlocks generated from Figure 2 for various load latencies.

ditional scheduler, assuming load instruction weights of 5 and 1, respectively. These two schedules illustrate the effect of over- and under-estimating load instruction latency. In Figure 2a, if L1 incurs an actual latency greater than one, hardware interlocks will be inserted before X4. We say the scheduler is *greedy* in this case, because L0 captured all of the load level parallelism and left none for L1. The opposite situation occurs when load instruction weights are too small. Figure 2b illustrates the schedule produced when a weight of one is used. In this case we have not taken advantage of the load level parallelism with respect to L0. We say the scheduler was *lazy*, because it passed over opportunities for parallelism. Should the actual latency be greater than the scheduling assumption, the processor will needlessly stall. Figure 2c is the schedule that the balanced scheduler generates. The balanced scheduler has measured the load level parallelism in the DAG and determined that a weight of 3 assigned to each load instruction would generate an efficient schedule.

Figure 3 summarizes the number of interlocks that accrue when these schedules are executed with varying memory latencies. The chart shows that, for latencies in the range of 2–4, the balanced schedules are faster than both the greedy and lazy traditional schedules illustrated in Figure 2. Outside this range the balanced and traditional schedules perform equivalently.

In summary, balanced scheduling's strength is its ability to look beyond fixed latencies, thereby exposing additional instruction level parallelism. Whereas traditional schedulers plan for the optimal latency, balanced schedulers make scheduling decisions based on the amount of load level parallelism the code can support. It therefore produces fewer interlocks when the optimal case doesn't occur.

3 The Balanced Scheduling Algorithm

This section presents the balanced scheduling algorithm. The algorithm is first illustrated through two simple examples. The examples depict the two relationships load instructions can have with each other, i.e., occurring in series and in parallel, and, therefore, the two cases the algorithm must handle.

When presented with load instructions in series, the balanced scheduling algorithm equally distributes among them all instructions with which they can execute in parallel. Referring again to the code DAG of Figure 1, the two load instructions, L0 and L1, may execute independently of X0, X1, X2 and X3. Since L1 is dependent on L0, the obvious partitioning would schedule two instructions after L0 and two after L1. The weight on each load instruction is simply one (for the issue slot of the load), plus the number of instruction issue slots that may be initiated independently of the load divided by the number of loads in series or, $1 + (4/2) = 3$. Issue slots are measured, because instruction weights repre-

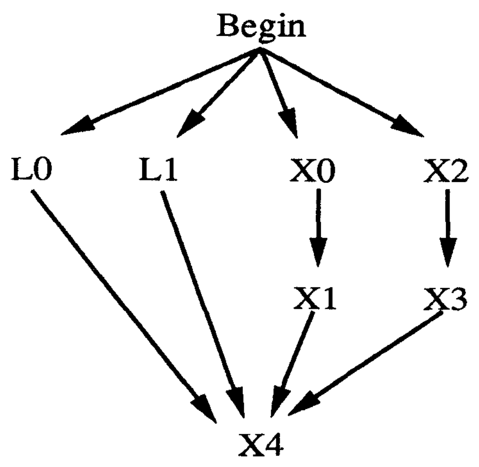

Figure 4: Code DAG in which L0 and L1 are independent and execute in parallel with all other instructions.

Balanced Scheduling
L0
L1
X0
X1
X2
X3
X4

Figure 5: Balanced schedule generated from the code DAG in Figure 4.

sent the number of machine cycles that should pass before an instruction that uses the result of the load is initiated.

When load instructions are not dependent on each other, i.e., they occur in parallel, their latencies can be hidden, using instructions drawn from the same set. Referring to the code DAG in Figure 4, the balanced scheduling algorithm takes advantage of the fact that L0 and L1 can, and should, share the same set of padding instructions. In Figure 4 each load instruction may execute in parallel with five other instructions, so they are each assigned a weight of six $(1+5/1)$. The final schedule is shown in Figure 5.

For a balanced scheduling algorithm to be successful, any combination of loads in series and loads in parallel must be accommodated.

A balanced scheduler operates by measuring load level parallelism and assigning weights accordingly. The algorithm, shown in Figure 6, examines each instruction i in the code DAG (G) and computes the set of instructions with which it may execute in parallel. It first eliminates from G those instructions that are predecessors or successors, recursively, producing G_{ind} (line 3). The resulting connected components of G_{ind} contain the sets of load instructions that may execute in parallel with i. Within each connected component, C, the path with the largest number of load instructions is located (lines 4–5). (We examine the longest load path, because loads on other paths can be overlapped with it.) Since the loads on this path execute in series, their sum (called $Chances$) represents the number of opportunities for scheduling i. Finally, the number of issue slots in the instruction execution pipeline that are required by i is divided by the number of loads in series (IssueSlots$(i)/Chances$), and is added to the accumulating weight of each load instruction in C (lines 6–7).

Figure 7a illustrates the balanced scheduling algorithm on a more challenging basic block. Using i=X1, step 4 generates the three connected components shown in Figure 7b. (L2 does not appear in a connected component because it is a predecessor of X1). The maximum path length in the component containing L1 is 1; therefore X1 contributes $1/1$ to L1's weight. The maximum path length in the second component is 3, and X1 contributes $1/3$ to the weights of each load instruction, L3, L4, L5 and L6. The third connected component has no load instructions. Table 1 shows the weight contributed by each instruction to each load at the completion of the algorithm. The latencies assigned to the five load instructions represent a distribution of load level parallelism that is representative of the load level parallelism in Figure 7.

If n is the number of nodes in the DAG, steps 4 and 5 together may be done in a worst case time of $O(n\ \alpha\ n)$[2], using the set union algorithm. First, each node in G_{ind} is labeled with its level from the farthest leaf. Next, it is combined with the nodes to which it is connected, using the set union function. Each time we perform set union, the set label is updated to reflect both the minimum and maximum level number that has been seen in that set. Therefore, the largest path length for each connected component is simply the maximum level number minus the minimum level number plus 1. Steps 6–7 are performed in $O(n)$ time and, therefore, do not impact the worst case time complexity. Connected component analysis is done for each instruction in the code DAG; therefore, the entire algorithm has a worst case time complexity of $O(n^2\ \alpha\ n)$. Since the worst case time complexity of list scheduling is $O(n^2)$, the balanced scheduling algorithm is nearly as efficient.

An alternate technique for assigning weights might compute a weight based on the *average* load level parallelism over all load instructions in a basic block. However, since load level parallelism typically varies within a basic block, this method does not consider those imbalances, often ignoring load level parallelism that is greater than the average for some loads, while unrealistically allocating nonexistent parallelism to others. Our early experiments indicated that this alternative produced schedules that executed no faster than schedules from the traditional scheduler.

[2] α is the inverse Ackerman function. As a function of n, it increases very slowly and may be considered constant[20].

Term	Definition
G	the code DAG.
Pred(i)	the transitive closure of the predecessor function on node i.
Succ(i)	the transitive closure of the successor function on node i.
Chances	the maximum number of loads on any path in a connected component.
IssueSlots(i)	the number of issue slots in the instruction execution pipeline required by instruction i.

```
1.  Initialize the latency of each load instruction to 1.
2.  for each instruction $i$ in $G$
3.      $G_{ind} = G - (\text{Pred}(i) \cup \text{Succ}(i))$
4.      for each connected component $C$ in $G_{ind}$
5.          Find the path with the maximum number of load instructions.
6.          for each load instruction $l \in C$
7.              add IssueSlots($i$)/Chances to the weight of $l$
```

Figure 6: Balanced scheduling algorithm

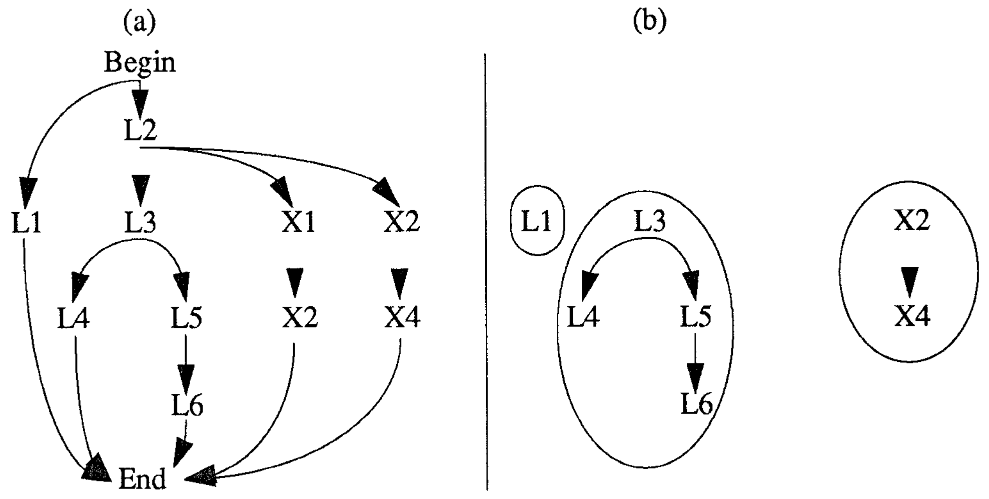

Figure 7: Balanced scheduling example, with the code DAG (a) and the connected components which determine the possible placements for X1 (b).

4 Experimental Methodology

We designed a series of experiments to compare balanced scheduling with a traditional scheduling approach. These experiments modeled the execution of real programs running on several different architectures. This section describes the methodology of these experiments. The integration of the balanced scheduler into the GCC compiler, the workload and the simulator we used for our measurements are described, in turn, in sections 4.1 through 4.3.

For our experiments we classify the target machine characteristics into two groups. The processor characteristics are those that control how the processor exploits parallelism with respect to load instructions. The system characteristics are the attributes of the memory system in a particular implementation. We used several alternatives for each model, to demonstrate that balanced scheduling works well on architectures that contribute to latency uncertainty in different ways. The processor and system models we used are described in sections 4.4 and 4.5.

4.1 Compiler

We modified the GNU GCC version 2.2.2 compiler[19] to perform balanced instruction scheduling. The default instruction scheduler within GCC was replaced by a new module that can schedule using either the traditional or balanced approaches. In addition, several changes were made to GCC to increase scheduling effectiveness and improve instruction level parallelism. The changes include alleviating the effect of dependences in spill code introduced by register allocation, our heuristics for picking instructions from the ready list (one of which helps control register pressure) and modifications to GCC's RTL intermediate language. Both schedulers take advantage of these modifications.

Load	Contribution by										Total Weight
	L1	L2	L3	L4	L5	L6	X1	X2	X3	X4	
L1	0	1	1	1	1	1	1	1	1	1	10
L2	1/4	0	0	0	0	0	0	0	0	0	1 1/4
L3	1/4	0	0	0	0	0	1/3	1/3	1/3	1/3	2 5/12
L4	1/4	0	0	0	1	1	1/3	1/3	1/3	1/3	4 5/12
L5	1/4	0	0	1/2	0	0	1/3	1/3	1/3	1/3	2 11/12
L6	1/4	0	0	1/2	0	0	1/3	1/3	1/3	1/3	2 11/12

Table 1: How instruction weights are calculated for Figure 7. The total weight is one plus the sum of the weight contribution of each instruction to each load.

GCC performs instruction scheduling both before and after register allocation. Since register allocation may add spill code and/or copy instructions, the second scheduling pass serves to integrate these additional instructions into the final schedule. However, the effectiveness of the second scheduling pass is restricted because of dependences introduced by register allocation.

These false dependences negatively effect schedule performance in two ways. First, the final assignment of register numbers severely limits the code motion that a scheduler can perform. Second, when adding spill instructions, the GCC compiler always uses register numbers selected from a small pool of spill registers. The net effect is that spill code cannot be scheduled effectively with other instructions. We improve performance by increasing the size of GCC's spill register pool by two and implementing a FIFO queue-like ordering of the registers in the pool. An alternative approach would use software register renaming after register allocation to better integrate spill instructions.

As previously mentioned, both the balanced and traditional schedulers use the same list scheduler. Some list schedulers place instructions onto the ready list when all their predecessors in the code DAG have been scheduled. In contrast, our scheduler defers adding these instructions to the ready list until each predecessor has exhausted its expected latency. In the case of starvation the scheduler inserts virtual no-op's into the instruction stream. This delayed insertion of instructions into the ready list increases the accuracy of instruction placement within the schedule. Since our processors use the hardware interlock model of execution, the virtual no-ops are removed before actual code generation.

List schedulers select instructions from the ready list in priority order. In our case, the priority of an instruction is equal to its weight plus the maximum priority among its successors. In the event of ties we select instructions using alternate heuristics in the following order. The first selects the instruction that has the largest difference between consumed and defined registers; this heuristic helps control register pressure. The second ranks instructions based on the number of successors in the code DAG that would be exposed for scheduling if that instruction were to be selected; it gives the list scheduler more instructions from which to select. The final heuristic selects the instruction that was generated the earliest. Our list scheduler is a bottom-up scheduler, therefore we generate schedules in reverse order by scheduling from the leaves of the code DAG toward the roots.

The compiler has been configured for the MIPS RISC processor[12]. GCC's intermediate language, RTL, is not sufficiently RISC-like for an instruction scheduler to get maximum benefit, since some primitive operations in RTL are actually multi-cycle macros. In the context of this work, memory-to-memory copies are the most notable, since it is load instructions that we are concentrating on scheduling. Our implementation extracts GCC's intermediate language after optimization but before register allocation and modifies it to replace certain non-RISC patterns, such as memory-to-memory copy, with their RISC equivalents. The modified RTL is at a lower level and therefore more suitable for instruction scheduling.

Loop unrolling is an optimization that increases instruction level parallelism. Due to a conflict with the way we use profiling information (section 4.3), GCC's unrolling capability is not usable for these experiments. Therefore, unrolling was performed manually.

4.2 Workload

The workload consisted of the Perfect Club suite of benchmarks[4]. Since these programs are written in FORTRAN, they were converted to C using *f2c*[7]. The Fortran-to-C converter produces C programs that correctly represent the semantics of the original FORTRAN programs. However, these C programs are conservative translations: after being compiled by a C compiler, they will most likely execute more slowly than if they were compiled by a FORTRAN compiler. For example, since almost all data is referenced through pointers in the C program, it is nearly impossible for a C compiler to do the memory reference disambiguation that might be obvious to a FORTRAN compiler. Instruction scheduling is affected, because load instructions are not free to move above stores. Since this problem severely restricts a scheduler's ability to exploit load level parallelism, we apply a transformation which more correctly models the dependences in the FORTRAN program and increases the available parallelism.

```
                                   float a[HUGE], b[HUGE];
float func(a, b)
   float *a, *b;                   float newfunc(a, b)
{                           ⇒         float *a, *b;
   a[1] = b[2];                    {
   a[2] = b[3];                       a[1] = b[2];
}                                     a[2] = b[3];
                                   }
```

Figure 8: An example *f2c* program showing the disambiguation problem and our transformation. In `func` the load of `b[3]` must be considered dependent on the store of `a[1]`. Our transformation results in `newfunc`. The resulting program produces incorrect results, but accurately models the code that would be generated by a FORTRAN compiler.

The FORTRAN standard[3] specifically disallows aliasing among dummy arguments (formal parameters) if there will be any stores to the dummy arguments. If the function `func` in Figure 8 were produced by *f2c*, the FORTRAN standard would assume that array `a` and array `b` were disjoint; therefore the load for `b[3]` could be scheduled before the store of `a[1]`. However, the C semantics for `func` insert a true data dependence between the store of `a[1]` and the load of `b[3]`. This dependence is an artifact of the Fortran-to-C translation and does not exist in the original program.

Our compiler takes advantage of the FORTRAN semantics by performing a parallelism-exposing transformation on the input C programs. The transformation would replace `func` with `newfunc`, as illustrated in Figure 8. New global variables are inserted with the same names as the original subroutine parameters. The formal parameters are replaced with names that are never referenced. The program is no longer semantically correct, but the compiler is now able to correctly model the FORTRAN independence between references to array `a` and array `b`. The net effect is the generation of code that is comparable to that generated by a FORTRAN compiler. This transformation is a conservative representation of the data dependences that a FORTRAN compiler could discover, since FORTRAN is quite specific about when aliasing may occur.

4.3 Simulator

After the second scheduling pass, the machine instructions are extracted and run through an instruction level simulator. Given a particular model for load instruction latencies (explained in section 4.5), the simulator simulates instruction issue and completion for each basic block and computes its execution time in cycles.

As the simulator encounters load instructions, it draws latency samples from a random distribution that represents the system-level characteristics being modeled (see section 4.5). The output of the simulator is one sample of the number of instruction and interlock cycles that comprise the execution time of the program on the modeled system. Because the results of the simulation are based on an independent and identically distributed random variable, we can take several steps to both reduce the execution time of the simulation and improve the quality of the results.

Our method executes the full instruction-by-instruction simulation 30 times with new random numbers on each iteration. The number 30 represents an arbitrary choice which is large enough to avoid statistical noise.

Second, we measure the accuracy of our results by generating confidence intervals. Confidence intervals are computed for percentage improvement using a bootstrapping[5] procedure. From the 30 sample runtimes, we randomly draw 30 samples, with replacement, in order to generate a second sample mean. This process is repeated until we have 100 sample means for the block. These 100 sample mean runtimes are scaled by the profiled execution frequency to compute the actual runtime of the block. The sample means for each block are summed giving 100 sample runtimes for the entire program. The mean runtime reported is the mean of the 100 sample mean runtimes.

In order to report a percentage improvement for balanced scheduling, the 100 sample means from the balanced scheduler are paired with an equal number from the traditional scheduler, and the calculation is performed. After sorting, a 95% confidence interval is directly extracted.

4.4 Processor-level model

Processor-level attributes model a processor's ability to exploit load level parallelism. We model three different configurations. The first is unrealistically aggressive and serves as a best case reference. The second two are restricted in ways that make them implementable. All of our processor models are assumed to maintain store/load consistency, i.e., if a load instruction follows a store, and they reference the same address, the load instruction receives the data that was written by the store instruction.

The first processor model (called UNLIMITED) can dispatch non-blocking load instructions with no limit on the number of loads outstanding. This model is similar to theoretical dataflow machines[10]. It is of interest because it exposes the maximum benefit that processor parallelism can achieve. The second (called MAX-8) allows a maximum of eight load instructions to be simultaneously executing. If a ninth load instruction is issued, the processor blocks until one of the eight outstanding loads completes. The third processor model (called LEN-8) restricts the maximum number of cycles a load instruction can take before blocking, as in the Tera Computer[2].[3] In this model, if a load instruction has been outstanding for eight cycles, the processor blocks until the data is returned.

The balanced scheduler has not been specifically configured for any of the processor models. In particular, it may schedule more than eight load instructions before using loaded data (as is prohibited in MAX 8), and it might assign load instructions weights greater than eight (not effective in LEN-8). If this information were available to the compiler, the results for MAX-8 and LEN-8 would improve. We used a processor-independent version of balanced scheduling to demonstrate that a code scheduling approach that was not associated with a particular implementation, but instead was based solely on program characteristics, such as the amount of load level parallelism, would generate efficient code.

4.5 System-level model

Three memory systems are modeled and simulated, representing different latency behavior in both current and future architectures. The first has a data cache. A load instruction's data is returned after 2 cycles on a cache hit and either 5 or 10 cycles on a cache miss. The model represents a typical workstation-class RISC processor that implements nonblocking load instructions, such as the Motorola 88000 series[15]. It is simulated with cache hit rates of 80% and 95%, modeling first level caches of 4K and 32K bytes, respectively[11]. Four configurations are modeled, and are referred to as L*hr*(*hl*,*ml*), where L*hr* stands for lockup-free caches with a hit rate of *hr*, and *hl* and *ml* are hit and miss latencies, respectively.

The second model has a memory interconnection network and no cache. The interconnection scheme uses a hashing function to assign addresses to memory modules, effectively randomizing memory access locations. In this architecture, memory latencies modeled by one of two zero-based probability mass functions, depicting normal distributions with standard deviations of 2 or 5. A standard deviation of 2 represents a machine in a relatively stable state (uniform network load, low to medium uncertainty). A standard deviation of 5 represents one with unpredictable memory latencies (changing network load, high uncertainty). The network machine is modeled in seven different configurations. Each distribution is combined with a mean of 2, 3 or 5, representing different base load levels. In a multithreaded processor such as the Tera, the different means are related to the number of active threads; the more threads, the lower the mean memory access time. We refer to these models as N(μ,σ) where μ is the mean of the distribution and σ is the standard deviation. All six configurations are reasonable design points for the machine. A seventh configuration models an unbalanced system, with a mean access time of 30 cycles and a standard deviation of 5 (N(30,5)). Although we recognize that a compiler would not likely generate code specifically for such an unbalanced configuration, we include it in order to gauge balanced scheduling's ability to handle a workload that has too little load level parallelism to hide the average latency.

The third machine has both a data cache and a Tera-style memory interconnection network. A cache hit occurs 80% of the time and takes two cycles. A cache miss is represented by a normal distribution with a mean of 30 and a standard deviation of 5. This configuration is referred to as L80-N(30,5) and has a mean latency of 7.6. In this case the 30 cycle latency is a reasonable design point, since the cache satisfies most requests. The model is intended to be representative of Alewife-like systems[1], where a commodity processor might be incorporated into a shared memory machine.

[3] The Tera restricts the number of instructions rather than cycles; since we assume that instructions other than loads execute in a single cycle, the two are equivalent.

5 Experimental Results

The first set of results is the percentage improvement in execution time of the balanced scheduler over the traditional scheduler. (The results for the UNLIMITED processor model appear in Table 2. Positive values indicate an improvement due to balanced scheduling.) For these experiments, the traditional scheduler uses load latencies equal to the cache hit time or effective access time for models with caches and the mean of the normal distribution for models without caches (labeled Optimistic Latency in the table). The percentage improvement of balanced scheduling over traditional scheduling is quite good. The average decrease in execution time for the UNLIMITED model varies from 3 to 18 percent for individual system models, with a mean improvement of 9.9%. The results for MAX-8 and LEN 8 are similar, with ranges of 7% to 16% and 3% to 16%, and means of 10.0% and 8.7%, respectively. These results demonstrate that balanced scheduling works well for several architectures, each of which contributes to latency uncertainty in a different way. It is important to emphasize that the balanced scheduler has not been customized for the restricted processors; these results represent the improvement from a machine-independent scheduler and would be better if the processor dependences were taken into account.

The balanced scheduler does relatively better (over the traditional scheduler) as the uncertainty of the load instruction latencies increases. This can be seen in three different situations: when the cache hit rate is low (L80 vs. L95); when the cache miss penalty is high (L80(2,10) vs. L80(2,5) and L95(2,10) vs. L95(2,5)); and when the standard deviation of the normal is high (N(2,5) vs. N(2,2), etc.).

To better understand the reasons for the performance im-

Processor model: UNLIMITED — Unlimited loads										
System	Optimistic Latency	Percentage improvement from balanced scheduling								
		ADM	ARC2D	BDNA	FLO52Q	MDG	MG3D	QCD2	TRACK	Mean
Data cache; bus-based interconnection										
L80(2,5)	2	5.8	6.7	6.0	4.9	9.8	7.0	19.3	7.2	8.3
	2.6	4.0	6.2	5.2	3.6	8.7	6.2	18.6	2.6	6.9
L80(2,10)	2	9.9	13.1	10.6	8.7	14.4	11.9	27.8	6.7	12.9
	3.6	7.5	11.7	8.1	6.7	11.6	10.7	25.8	2.2	10.5
L95(2,5)	2	3.4	3.9	4.4	2.8	6.9	3.7	16.9	6.1	6.0
	2.2	2.1	4.0	4.0	2.1	6.2	3.9	16.2	2.0	5.1
L95(2,10)	2	4.6	5.8	5.8	3.9	8.0	5.4	19.9	4.9	7.3
	2.4	3.2	6.1	5.6	3.8	7.2	5.8	19.0	1.7	6.6
No cache; network interconnection										
N(2,2)	2	8.0	9.3	8.0	6.5	11.3	9.2	21.3	9.4	10.4
N(3,2)	3	6.4	8.9	4.0	5.0	12.0	8.6	22.7	3.5	8.9
N(5,2)	5	4.8	5.5	3.4	3.6	13.5	6.5	20.0	3.9	7.7
N(2,5)	2	14.2	17.7	14.4	11.9	20.9	16.1	32.7	16.6	18.1
N(3,5)	3	11.5	18.2	10.8	10.9	20.0	14.9	35.9	3.9	15.8
N(5,5)	5	9.2	12.0	9.3	7.6	18.3	10.3	27.8	4.9	12.4
N(30,5)	30	-3.5	-5.0	1.9	4.1	19.3	-0.9	7.1	0.6	3.0
Mixed										
L80-N(30,5)	2	12.4	20.4	15.8	12.7	20.0	13.3	39.6	11.3	18.2
	7.6	7.0	9.3	18.4	6.3	14.3	4.5	19.4	-2.5	9.6

Table 2: Percent improvement in execution time from simulations using processor model UNLIMITED

Program: MDG (BIns = 5,144 million)											
System	Optimistic Latency	TIns	UNLIMITED			MAX 8			LEN 8		
			Imp%	TI%	BI%	Imp%	TI%	BI%	Imp%	TI%	BI%
Data cache; bus-based interconnection											
L80(2,5)	2	5,358	9.8	10.4	5.6	7.8	13.9	10.9	9.6	10.4	5.7
	2.6	5,351	8.7	9.6		7.4	13.7		8.2	9.3	
L80(2,10)	2	5,358	14.4	21.6	13.6	10.8	25.2	20.6	14.6	22.5	14.7
	3.6	5,299	11.6	20.2		8.9	24.7		11.6	21.2	
L95(2,5)	2	5,358	6.9	5.9	3.4	6.0	8.8	7.1	6.8	5.8	3.4
	2.15	5,351	6.2	5.5		6.0	8.9		6.2	5.3	
L95(2,10)	2	5,358	8.0	9.4	6.1	6.4	12.1	10.2	8.0	9.9	6.6
	2.4	5,351	7.2	8.9		6.8	12.5		7.2	9.4	
No cache; network interconnection											
N(2,2)	2	5,358	11.3	12.8	6.9	10.2	16.6	11.8	11.2	13.1	7.2
N(3,2)	3	5,351	12.0	16.1	9.7	10.0	21.1	16.5	12.0	16.0	9.5
N(5,2)	5	5,297	13.5	24.4	16.8	10.8	32.1	27.0	12.9	24.3	17.0
N(2,5)	2	5,358	20.9	30.0	18.7	15.9	35.6	28.3	19.2	30.4	20.4
N(3,5)	3	5,351	20.0	31.8	21.3	14.5	37.2	30.9	17.4	31.9	23.0
N(5,5)	5	5,297	18.3	35.5	25.9	13.5	42.3	36.4	16.4	36.3	28.0
N(30,5)	30	5,393	19.3	71.8	67.9	14.5	79.4	77.5	18.7	73.1	69.6
Mixed											
L80-N(30,5)	2	5,358	20.0	49.9	42.3	13.6	52.3	47.9	13.9	49.5	44.7
	7.6	5,405	14.3	46.9		11.5	50.9		10.4	47.4	

BIns =	The number of instructions executed, in millions, for the balanced scheduler.
TIns =	The number of instructions executed, in millions, for the traditional scheduler.
Imp% =	The percentage improvement of the balanced scheduler over the traditional scheduler.
TI% =	The percentage of cycles which were interlock cycles for the traditional scheduler.
BI% =	The percentage of cycles which were interlock cycles for the balanced scheduler.

Table 3: Detailed analysis of performance in MDG

provements, we did a component analysis of the execution times. All of our instructions execute in a single cycle; therefore the runtime of a program is the sum of the number of instructions executed and the number of interlocks incurred. Table 3 presents interlock information on the performance of one of the benchmarks, MDG. In this table, the percentage of the total number of cycles that were interlock cycles is reported for both the traditional and balanced schedulers. MDG's performance gain with balanced scheduling (and also that of the other programs) is a result of both executing fewer instructions (BIns < TIns) and incurring fewer interlocks (BI% < TI%).

Balanced schedules execute fewer instructions because their schedules often contain less spill code. Table 4 presents data on the percentage of total instructions executed that was classified as spill code. (A spill instruction is defined to be any instruction that is inserted by the register allocator.) Balanced scheduling incurred fewer spills than the traditional scheduler for virtually all implementation-defined latencies on all programs. (The sole exceptions were ARC2D with an optimistic latency of 30 cycles and FLO52Q with 3.6 cycles.)

We hypothesize that the reduction in interlocks and spill code when using the balanced scheduler is a direct consequence of its always considering load level parallelism when calculating latency weights. It measures the parallelism, and, whether it is high or low, tries to use it to hide all load latencies in a basic block.

When there is significant load level parallelism, code DAGs tend to be bushy, causing all list schedulers to schedule independent instructions in parallel. The balanced scheduler manages this by assigning load instruction weights in such a way that load latencies are hidden by the other instructions. Traditional schedulers lack the guidance for efficient load placement. Therefore they incur similar register pressure, but also more interlocks.

When there is little load level parallelism, traditional schedulers greedily let independent instructions float to one end of the basic block. Therefore they incur spills at that end, and interlocks at the other. In contrast, the balanced scheduler spreads out the few independent instructions behind all loads. In all cases uses quickly follow definitions, and little or no spill code is generated. If the load level parallelism is less than the latency assumed by the traditional scheduler, balanced scheduling generates fewer spill instructions than the traditional technique.

In both situations (high and low load level parallelism) balanced scheduling contributes either little additional or less register pressure. When actual latencies differ from the optimistic latency, balanced scheduling incurs fewer interlocks; when both latencies are equal, the number of interlocks produced by the two schedulers is similar.

When load latencies are much larger than the amount of load level parallelism and therefore cannot be hidden via instruction scheduling, there is no guarantee the balanced scheduler will do better. In this case, register pressure can be a problem, and balanced scheduling can insert more spill code than the traditional scheduler. The situation is illustrated in Table 5, which summarizes the results for the N(30,5) model. This model assumes a mean latency much larger than the amount of load level parallelism of the programs in our workload. Two interrelated factors contribute to balanced scheduling's poor performance with this model. First, as latencies get long, interlocks account for an increasingly large proportion of execution time. Both schedulers do poorly, and often equally poorly (for example, see TRACK). Second, a consequence of long load latencies is that each load instruction consumes more cycles relative to other instructions, and its contribution to execution time is greater. Therefore whichever scheduler generates more spill loads will have the poorer performance. Occasionally balanced scheduling chooses load instruction weights that cause higher than necessary register pressure and consequently issues more spill instructions (for example, see ARC2D).

In summary, these results indicate that balanced scheduling reduces execution time relative to traditional list scheduling in most cases. Because its schedules are based on the amount of load level parallelism that a program can support, they cause fewer interlocks during program execution and contain less spill code. The benefits are most apparent when memory latency uncertainty is high, as evidenced by greater miss rates and penalties, and larger standard deviations from mean latencies.

6 Extensions

Balanced scheduling has been presented in a specific form (weights calculated based on load level parallelism) to solve a specific problem (scheduling with uncertain load instruction latencies). The technique should be applicable to a wider set of problems, such as other multi-cycle instructions (e.g., floating point operations coupled with asynchronous floating point units), disabling balanced scheduling when the latency is known (e.g., for the second access to a cache line), techniques that enlarge basic blocks (trace scheduling and software pipelining) and superscalar architectures.

7 Summary

This paper describes an instruction scheduling algorithm, called balanced scheduling, that is appropriate for computers that expose uncertain memory latencies. Balanced scheduling is fundamentally different from previous list schedulers in two respects. First, it ignores the optimistic, implementation-determined memory latency when assigning scheduling priorities, basing them instead on the amount of parallel execution that is achievable in the program. Second, it computes individual scheduling weights for each load instruction separately, rather than using a single value for all

		Percentage of Spill Instructions									
		Balanced	Traditional Scheduler with Optimistic Latency of								
Program	BIns	Scheduler	2	2.15	2.4	2.6	3	3.6	5	7.6	30
ADM	2,494	7.43	9.59	9.15	9.15	9.15	9.22	9.42	9.50	8.70	7.49
ARC2D	11,149	10.47	13.52	13.74	13.74	13.68	13.27	13.46	13.89	12.25	10.11
BDNA	2,391	22.84	26.50	26.32	26.32	26.32	24.17	24.94	24.68	24.73	25.54
FLO52Q	3,323	4.61	7.14	6.82	6.82	6.82	6.97	3.91	6.55	5.89	4.90
MDG	5,144	7.49	7.86	8.04	8.04	8.04	8.04	8.13	8.00	8.86	9.21
MG3D	60,784	7.38	9.73	10.36	10.36	10.36	10.36	10.86	10.36	8.85	7.88
QCD2	1,176	19.91	29.30	28.92	28.92	28.92	28.92	28.78	28.02	26.89	28.02
TRACK	398	15.78	20.41	17.85	17.85	17.85	17.85	17.85	17.84	17.45	17.46

Table 4: Spill Instructions Executed

			UNLIMITED			MAX 8			LEN 8		
Program	TIns	BIns	Imp%	TI%	BI%	Imp%	TI%	BI%	Imp%	TI%	BI%
ADM	2,496	2,494	-3.5	67.8	69.0	-1.7	76.1	76.5	-2.7	69.9	70.7
ARC2D	11,108	11,149	-5.0	67.3	68.9	-3.9	78.4	79.1	-4.7	70.9	72.1
BDNA	2,478	2,391	1.9	65.0	65.6	12.5	85.3	84.0	2.5	71.1	71.4
FLO52Q	3,332	3,323	4.1	67.0	65.7	1.4	76.6	76.4	3.7	69.0	67.9
MDG	5,393	5,144	19.3	71.8	67.9	14.5	79.4	77.5	18.7	73.1	69.6
MG3D	61,116	60,784	-0.9	63.1	63.7	1.5	86.6	86.5	-3.8	67.4	68.8
QCD2	1,270	1,176	7.1	69.0	69.2	23.4	86.4	84.5	6.3	72.2	72.6
TRACK	406	398	0.6	81.6	81.9	4.7	85.6	85.2	2.3	82.3	82.3

Table 5: Analysis of N(30,5) results — the effect of spill code.

loads in a basic block. Balanced scheduling thus insulates program execution from machine uncertainties by generating schedules that are optimized for the program rather than the machine.

To validate the algorithm balanced scheduling was incorporated into the GCC compiler and the performance of the Perfect Club benchmarks scheduled with both balanced scheduling and a traditional list scheduler was compared. Three processors were modeled, representing machines with varying abilities to exploit instruction level parallelism. Each of the processor models was coupled with several memory systems that exhibit dissimilar latency behavior. Execution time reductions of balanced scheduling over the traditional list scheduler averaged between 3% and 18%, depending on the processor model, system model and program. The results demonstrate that, if the capability to exploit uncertain memory latency is architected in future machines, balanced schedulers can effectively take advantage of the additional flexibility to generate faster schedules.

8 Acknowledgements

We would like to thank David Bradlee, Craig Chambers and Robert Henry for their valuable comments on an earlier draft and the compiler group at Tera Computers for their design review.

References

[1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114. IEEE, May 1990.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *1990 International Conference on Supercomputing*, pages 1–6. SIGARCH, June 1990.

[3] ANS X3.9-1978. *American National Standard Programming language FORTRAN*. American National Standards Institute, New York, 1978.

[4] M. Berry, D. Chen, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Samah, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The perfect club: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3), Fall 1989.

[5] Bradley Efron. The jackknife, the bootstrap, and other resampling plans. SIAM/CBMS-NSF Regional conference series in applied mathematics, volume 38, 1982.

[6] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* ACM doctoral dissertation award; 1985. The MIT Press, 1986.

[7] S. I. Feldman, David M. Gay, Mark W. Maimone, and N. L. Schryer. A Fortran-to-C converter. Computer Science Technical Report 149, AT&T Bell Laboratories, Murray Hill, NJ 07974, April 1991.

[8] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), July 1986.

[9] John L. Hennessy and Thomas R. Gross. Code generation and reorganization in the presence of pipeline constraints. In *Symposium on Principles of Programming Languages*, pages 120–127, January 1982.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[11] Mark Donald Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* PhD thesis, University of California, Berkeley, November 1987.

[12] Gerry Kane. *mips RISC Architecture.* Prentice-Hall, 1988.

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

[14] E. Lawler, J. K. Lenstra, C. Martel, B. Simons, and L. Stockmeyer. Pipeline scheduling: A survey. Research Report RJ-5738, IBM, July 1987.

[15] Motorola. *MC88100 RISC Microprocessor User's Manual.* Prentice Hall, 1990.

[16] Krishna V. Palem and Barbara B. Simons. Scheduling time-critical instructions on RISC machines. In *ACM Symposium on Principles of Programming Languages*, January 1990.

[17] C. Scheurich and M. Dubois. Lockup-free caches in high-performance multiprocessors. *Journal of Parallel and Distributed Processing*, 11(1):25–36, January 1991.

[18] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processor. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 53–62, April 1991.

[19] Richard Stallman. *The GNU project optimizing C compiler.* Free Software Foundation, Inc.

[20] Robert Endre Tarjan. *Data Structures and Network Algorithms*, volume 44 of *Regional Conference Series in Applied Mathematics.* Society for Industrial and Applied Mathematics, 1983.

[21] H. S. Warren, Jr. Instruction scheduling for the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1), January 1990.

[22] Michael J. Woodard. Personal communication. Scheduling techniques used in Sun SPARC compilers, September 1992.