Scheduling Load Operations on VLIW Machines

Charles R. Hardnett, Krishna V. Palem, Rodric M. Rabbah
Georgia Institute of Technology
777 Atlantic Drive, NW
Atlanta, GA 30332
{hardnett, palem, rabbah}@ece.gatech.edu

Weng-Fai Wong
National University of Singapore
Singapore
wongwf@comp.nus.edu.sg

ABSTRACT

There continues to be a widening gap between processor speeds and memory access time. This gap is seen in systems ranging from embedded computing systems to high-performance supercomputing systems. In this paper, we present an instruction scheduling algorithm that can be targetted towards VLIW architectures commonly found in embedded systems and high-performance workstations i.e. Itanium. The goal of this paper is to present a simple instruction scheduling algorithm that does not require substantial hardware support to address the scheduling of load operations to mask the latency of delinquent loads; which are associated with high miss rates and very long average latencies. Our algorithm is named *Cache Sensitive Scheduling* (CSS). CSS is designed to be sensitive to the varying memory latencies of load operations, and compensate for those latencies within the instruction schedule by masking the typically long latencies of load operations with useful operations to reduce stall penalties. CSS can extend a rank-function based scheduler with two additional components to intelligently incorporate the profiled average latency of an operation, and the latencies of its predecessors. Our results show that these additional components are effective in generating schedules that are more sensitive to the latencies of load instructions. To support the selection and relative weight of our rank function components we use multivariate statistical analysis to determine the degree of correlation between our rank components and the execution time of the program. In our experiments with a VLIW parameterized compiler-simulator infrastructure using a variety of memory hierarchy configurations; we were able to achieve 20% speedups and 44% stall cycle reductions over a more conventional critical path scheduling algorithm.

Keywords

Instruction Scheduling, VLIW, EPIC, Instruction Level Parallelism, Cache, Rank Function, Multivariate Statistics

1. INTRODUCTION

There continues to be a widening gap between processor execution rates and memory access times. Researchers have proposed a number of strategies that address this problem via additional hardware logic, extensions to the instruction set, compiler optimizations, or some combination of these approaches. Our approach is a compiler-based instruction scheduling algorithm named *Cache Sensitive Scheduling* (CSS). CSS is targetted towards VLIW instruction set architectures found in embedded systems as well as high-performance workstations. These instruction set architectures (ISAs) support high degrees of instruction level parallelism (ILP). A single VLIW instruction contains a set of operations that are executed in parallel. These ISAs rely on the compiler to derive efficient instruction schedules; and as result provide the compiler with information regarding the number and type of functional units as well as the latencies of individual operations.

CSS is a localized instruction scheduler that benefits from the creation of large regions of code containing no branches. This provides CSS with more opportunities to shuffle operations to mask the latencies of load operations. Examples of these regions are hyperblocks [1] and superblocks [2] depending on how they are formed. Forming these regions requires the ability to convert branches resulting from if-statements to sequences of operations with no branches. Therefore, CSS requires ISAs to provide instruction predication, which is found in many embedded processors (ARM and SHARC) as well as high-performance explicitly parallel instruction computer (EPIC) workstations such as Itanium.

CSS is extends traditional compiler-based rank-function instruction scheduling techniques that focus on maintaining the critical path and improving ILP [3] [4] to mask the latency of delinquent load operations. CSS indirectly uses average latencies of load operations which are gathered from light-profiling. This makes CSS sensitive to the impact that a given load operation has on instruction schedule. CSS lessens or hides the impact these delinquent loads while still maintaining a high degree of ILP.

When an instruction scheduler becomes more aware of the latencies of load instructions and uses such information during scheduling, then we say it is a *Load Sensitive Scheduler* [5]. The algorithm we are presenting here, CSS, is a load-sensitive scheduler. The state-of-art instruction schedulers today will treat all load operations as equals. Either the load operations are assumed to always experience a cache hit latency (*Optimistic Scheduler*), or the load operations are expected to experience a cache miss latency (*Pessimistic Scheduler*). The example in figure 1 illustrates the problems with these approaches. As shown in part A, the pessimistic scheduler

*This work is supported in part by DARPA contract F33615-99-1499, Hewlett-Packard Laboratories and Yamacraw.

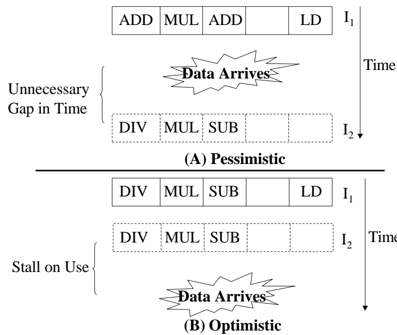

Figure 1: How conventional instruction schedulers handle load instructions

chooses to schedule the load instructions with longer latencies that resemble miss latencies. The result is that when data arrives early, the uses of that data are unnecessarily delayed in beginning execution. As shown in part B, an optimistic scheduler will choose to schedule the instruction earlier; however, if the data arrives in a later cycle the processor will be stalled until the data arrives. This approach will shorten the schedule, but increase the execution time. In addition, the optimistic scheduler is not concerned with hiding latencies; and therefore, there are no gaps to insert other useful non-dependent instructions.

```
for k = 1 to N step 5
    S1 += A[i][k]*B[k][j];
    S1 += A[i][k+1]*B[k+1][j];
    S1 += A[i][k+2]*B[k+2][j];
    S1 += A[i][k+3]*B[k+3][j];
    S1 += A[i][k+4]*B[k+4][j];
endfor
```

Figure 2: The above code is from the inner loop of a matrix-multiply implementation. The inner loop has been unrolled by 5.

Consider the example in figure 2. This example is based on code from an implementation of matrix-multiplication. The original inner loop requires two floating point load operations, one multiply operation, and one add operation. After loop-unrolling, it requires 5 times as many operations. This scenario provides an opportunity for uncovering ILP in the schedule. Due to the number of load operations, there are likely to be varying degrees of latencies and some delinquent loads. Ideally, the loads producing cache misses should be scheduled earlier in the iteration, while scheduling the operations that use the data later in the iteration. The cycles in between can be used to schedule the other operations that do not depend on the offending load operations. This reduces the effect of processor stalls related to dependent operations attempting to access data that has not arrived. At the same time, this increases the amount of ILP in the program. This is the essence of what CSS is designed to accomplish using a rank-based instruction scheduler framework. In other words, CSS uses realistic latencies of operations to determine how much of a gap should be placed between the load operation and the use of the data. The gap is tailored to the behavior of each load operation to provide the proper time for that particular load operation to load its data, while minimizing the effects found in optimistic and pessimistic scheduling. The result is a schedule that balances ILP against proper load-to-use distance, which results in better overall performance as shown in the table below.

Benchmark	% Compute Cycle Reduction	% Stall Cycle Reduction
DARPA DIS Benchmarks	18.18%	44.48%
Olden Benchmarks	6.35%	5.17%
Spec2000 Benchmarks	5.88%	6.46%

Table 1: Average percentage reductions in execution and stall cycles experienced by CSS in relation to the baseline scheduler over all selected benchmarks from a given suite.

We chose a rank-based instruction scheduling framework due to its power and simplicity. Rank functions can be easily integrated within a standard list-scheduling framework. Properly formulated, a rank function is an excellent heuristic for ordering operations in instruction scheduling. A conventional rank function could be:

$$CPFSrank(i) = \alpha * height(i) + \beta * fanOut(i) \quad (1)$$

The rank in equation 1 is used for the baseline scheduler, referred to as the Critical Path with Fanout Scheduler(CPFS). CPFS is based on two components, the $height(i)$ and the $fanOut(i)$. The first component, $height(i)$, gives priority to instructions which are on the critical path; and the second component, $fanOut(i)$ gives priority to instructions which enable the scheduling of the largest number of instructions. The α and β allow these factors to be weighted when computing the rank; for simplicity assume $\alpha = \beta = 0.5$. This rank function is not load-sensitive because it does not enable special handling of load operations that are present in the function. The two components of CPFS rank only characterize an operation by its data dependence relationships. A load-senstive rank function needs components that are directly affected by the varying latencies of load operations. Since CPFS is not load-sensitive, it will not be able to generate the appropriate gap between a load and its uses to alleviate the stall penalties of delinquent load operations.

Our solution uses profile-feedback compilation to enable the compiler ot take advantage of realistic memory usage measurements of each load operation including the average hit latency and miss, and the hit and miss ratios per instruction. Profile-feedback compilation is a well accepted methodology for compiling for embedded environments. It is now gaining acceptance for high-performance processors. We use linear regression and multivariate analysis techniques to support our algorithm is two ways:

1. Multivariate analysis is used to assess the degree to which each of the rank function components affects performance.
2. Linear regression is used to assess the stability of the profile data in context of varying benchmark input sets.

1.1 Summary of Main Results

In this paper, we make the following contributions:

1. We present a new scheduling algorithm with a load-sensitive rank function that has a running time $O(N * |V|)$ where V is

the set of all instructions in a region within the program, and N is the number of regions. Details are described in section 3.

2. Our scheduling algorithm obtains 20% improvements over the baseline CPFS scheduler. Details are found in section 4.

3. We also introduce a statistical analysis framework to evaluate the impact of the components of our rank function and to establish the stability of profile-based compiler optimization techniques. This methodology is detailed in section 5.

1.2 Experimental Setup

Our experimental infrastructure is based on *Trimaran* [6]. *Trimaran* provides a flexible compiler and simulator infrastructure for VLIW/EPIC-centered research. *Trimaran* allows the researcher to create any processor configuration within the HPL-PD [7] design space. In addition, a parameterized cache simulator is used to enable the creation of various cache structures. The flexibility of *Trimaran* enables us to validate our solution across various memory hardware parameters.

2. RELATED WORK

There have been other schedulers proposed to address the above stated issue in different ways. Kerns and Eggers [3] developed a scheduler that computes the available ILP for a given instruction. Their work targets RISC processors and shows a 3% to 18% improvement over a typical critical path scheduler. The Kerns scheduler treats all loads as having an optimistic latency, where CSS gives the operations a more realistic latency.

Sánchez and González [8] propose a way to integrate software prefetching with software pipelining in VLIW architectures. Their approach is to insert prefetch instructions into software-pipelined loops. The work extends the contributions of other similar algorithms [9] [10][11][12]. Prefetch insertion algorithms solve some of the problem, but require special hardware support. In addition, they are limited by the prefetch queue and buffer sizes. CSS does not require any special hardware support. In addition, CSS can be used with any prefetch insertion algorithm as well.

Johnson and Abraham [5] developed a load-sensitive scheduler which was available in internal releases of the Elcor compiler. This scheduler computed the slack in the schedule, which was the amount of time in the schedule that is found on non-critical paths through the DAG. They developed a framework for assigning this slack time to the load operations that were most likely going to miss. Their scheduler performed transformations on the DAG of operations in one of three ways: loads given longer latencies in the schedule, loads converted to prefetches and then moved within the control-flow, and loads converted to speculative loads and then moved up within the control-flow. Our CSS algorithm does not have to perform and control-flow transformations, and is applied to all loads in a very systematic way.

Ozawa, et al [13] developed an analytical algorithm for identifying loads that will cause misses during execution, and they complemented this algorithm with a scheduling strategy that targets these loads for optimization. Their work focused on scientific applications with regular access patterns via arrays. Our CSS algorithm is targeted applications with both regular and irregular access patterns.

3. THE CSS ALGORITHM

This section describes the details of CSS algorithm. The discussion is a top-down explanation of the algorithm, which includes the greedy list scheduler followed by the CSS rank function, and formal definitions of the rank function components.

3.1 Greedy List Scheduling with Ranks

Input: A Basic Block, Hyperblock, or Superblock $G \equiv \{V,E\}$
Output: A schedule where each $n \in V$ is bound to a cycle

1. Assign a *rank (priority)* to each instruction ($n \in V$).

2. *Sort* and build a priority list L of the instructions in non-increasing order of rank.

3. *Greedily list-schedule L* An instruction is ready provided that it has not been scheduled earlier, all of its predecessors have been scheduled, and the appropriate latencies have elapsed.

   Scan list L iteratively at each cycle, and choose the largest set of *ready* instructions that can be bound to resources given constraints on the number of available functional units (FUs).

Figure 3: Greedy list scheduling

Many instruction schedulers are based on the greedy list scheduling algorithm shown in figure 3. The greedy list scheduling algorithm schedules instructions found in a directed acyclic graph (DAG), which can represent a program region such as a basic block, hyperblock, or superblock. The nodes of the graph are the operations, the directed edges of the graph represent the data dependences between operations which are annotated with the expected latency of the source operation. The rank function prioritizes the instructions in the DAG [4]. This allows the scheduler to schedule instructions based on that priority ordering. The priority is a function of what the algorithm designer views as an important criteria in selecting operations. One traditional rank function is based on the height of the instruction within the DAG. This is typically referred to as the critical path scheduler, because the height in the DAG is a good estimator of the instructions on the critical path. The critical path is the longest path from the root node of the DAG to the farthest leaf node. Obviously, lengthening this path will likely lengthen the execution time of the program.

In the next subsection, we will define the CSS rank. After that, we will define the factors contributing to the rank function in the context of standard scheduling dependence DAG definitions.

3.2 The CSS Rank

The goal of the CSS rank is to create the appropriate distance between loads and uses of the values loaded. The appropriate distance as discussed in section 1 depends on the latency characteristics of each individual load instruction. This goal in itself can increase the length of the critical path; therefore, the CSS algorithm exploits ILP within the program to reduce the effect of the load latencies.

$$CSSrank(i) = \alpha * height(i) + \beta * fanOut(i) + \gamma * avgLatency(i) - \delta * predLatency(i) \quad (2)$$

The CSS rank function as shown in equation 2 contains some fa-

miliar rank function components, such as $height(i)$ and $fanOut(i)$ which are found in the CPFS rank (Equation 1). In addition, this function has other components, mainly $avgLatency(i)$ and $predLatency(i)$. Our unique contribution is these two components and the way in which they are incorporated. Our unique handling of these components gives us the ability to balance ILP gains with the ill-effects of poor memory utilization.

3.3 Formal Definition of Rank Factors

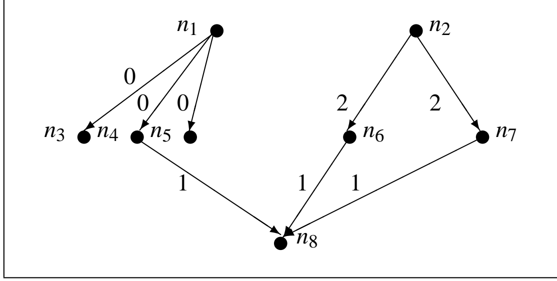

Figure 4: A Schedulable dependence DAG that represents basic block, hyperblock, or a superblock within a program.

Using the example DAG in figure 4, we show how each component of the rank function serves a purpose in either promoting or demoting the priority of a given operation. Promoting the priority of the operation will move the instruction earlier in the schedule, while demoting the instruction will move it later in the schedule. It is assumed that standard graph definitions for successors (succs) and predecessors (preds) are understood by the reader:

1. $height(i)$ refers to the relationship between i and the critical path of the DAG. If $height(i) > height(j)$ then node i is is higher in the DAG than node j, and will have a greater affect on the critical path.

   $$height(i) \equiv w^{+}(j,i) \text{, where j is the sink node of the DAG and } w^{+} \text{ defines a maximal weighted path} \quad (3)$$

   For example, consider nodes n_1 and n_2 in figure 4 and the definition of *height* in equation 3. In this case, $height(n_2) = 3 > height(n_1) = 1$, thus the $CSSrank(n_2)$ would be greater in the $height(i)$ component.

2. $fanOut(i)$ refers to the number of operations that are dependent on i. If $fanOut(i) > fanOut(j)$ then operation i will potentially enable the most operations to be scheduled.

   $$fanOut(i) \equiv \text{The number of outgoing edges of } i, \text{ where } (i,j) \in E \wedge j \in succs(i). \quad (4)$$

   Consider the nodes n_1 and n_2 in figure 4 and the definition of *FanOut* in equation 4. The $fanOut(n_1) = 3 > fanOut(n_2) = 2$, thus the $CSSrank(n_1)$ would be greater in the $fanOut(i)$ component.

3. $avgLatency(i)$ refers to the average latency experienced by operation i during profiling. Our profile framework is able to capture the latencies, hit/miss quantities, and hit/miss ratios for individual load operations. The average latency gives a more realistic latency experienced by the load than the optimistic or pessimistic latency.

   $$avgLatency(i) \equiv \begin{cases} alat(i) & \text{if } i \text{ is a memory operation} \\ default\ latency & \text{if } i \text{ is a non-memory operation} \end{cases}$$
   $$alat(i) = miss_lat(i) * miss_ratio(i) + hit_latency(i) * hit_ratio(i) \quad (5)$$

   where the $miss_lat(i)$ and the $hit_latency(i)$ are the latencies for individual load operations when a miss or hit occur respectively, which were found during our profile simulation. Likewise the $miss_ratio(i)$ and the $hit_ratio(i)$ depict the percentage of load that result in first-level cache misses and hits respectively. Again, consider the nodes n_1 and n_2 in figure 4 along with the definition in equation 5. In this case, the $avgLatency(n_2) = 2 > avgLatency(n_1) = 0$, thus the $CSSrank(n_2)$ would be greater in the $avgLatency(i)$ component. Note that the latency information is not used to alter the latencies on the edges of the nodes. Instead, the latency information is only used to compute the rank.

4. $predLatency(i)$ for operation i refers to the maximum $avgLatency(j)$, where $j \in preds(i)$. If i is dependent on longer latency operations then its priority should be decreased to allow the scheduling of operations that depend on operations with lower latencies.

   $$predLatency(i) = max\{avgLatency(j) \mid j \in preds(i)\} \quad (6)$$

   Finally, consider the nodes that depend on the nodes n_1 and n_2 in figure 4. These nodes are relatively equivalent in all components of the $CSSrank(i)$ with the exception of the $predLatency(i)$ component. In this case, since the nodes n_6 and n_7 depend on long latency operations, it would be better to schedule nodes n_3, n_4, and n_5 first since they will most likely be able to execute. Recall that n_2 is most likely a memory operation, and hence the $CSSrank(x)$, where $x \in \{n_6, n_7\}$ will be demoted by the $predLatency(i)$ component.

The combination of the *avgLatency* and the *predLatency* allow the rank function to move load instructions earlier in schedule, while moving instructions dependent on those loads to later in the schedule.

There is a delimma that can occur where components will conflict. For example, the $fanOut(n_1) > fanOut(n_2)$, while the $avgLatency(n_2) > avgLatency(n_1)$. The delimma is in deciding which component is more important; is the $avgLatency(i)$ more important than the $fanOut(i)$. The answer to this question, and other similar questions will determine the values for the α, β, γ, and δ. We developed an analysis to answer these questions and determine the values of those four variables by analyzing the relationships between a finite set of factors that we believe affect the program execution. Our analysis is based on Multivariate Statistics [14], and will be detailed in section 5.1.

4. EXPERIMENTAL RESULTS

The experimental results are simulation results using the *Trimaran* Infrastructure. *Trimaran* is explained in more detail in the following subsection. In the second subsection, we discuss the hardware parameters, and workload for the experiments. Finally, the last subsection discusses the results of the experiments.

Cache Size		Associativity	
L1	L2	L1	L2
16K/16K	32K(U)	4	16
16K/16K	96K(U)	4	6
64K/64K	1.75M(U)	2	7
32K/64K	8M(U)	4	8
16K/16K	512K(U)	4	8
64K/64K	512K(U)	2	8
32K/32K	8M(U)	4	2

Table 2: Cache parameters used in simulation experiments

4.1 Implementation & Infrastructure

The CSS algorithm was implemented as a part of the *Trimaran* Infrastructure [6], which is designed to investigate ILP in the context of VLIW-like machines found in the HPL-PD [7] architectural space. *Trimaran* uses the HMDES machine description language to enable parameterization of the number of registers, number of functional units, and operation latencies. The CSS algorithm is implemented within the compiler's back-end. In addition to the compiler, the *Trimaran* infrastructure provides cycle-accurate processor simulator. For this research the processor simulator was coupled with the Dinero cache simulator [15] to provide a full simulation of the processor and memory hierarchy. The processor simulator emulates the execution of each operation; load operations invoke the Dinero simulator. Dinero returns the number of cycles required to satisfy the load operation, which is based on whether it was a hit or a miss. The number of cycles returned by Dinero is used by the processor simulator to compute stall cycles using the stall-on-use model.

Our methodology utilized a profile-based framework [16]. We first collected profile information for a specific program running on base architecture; this is called the *training stage*. The training stage is done using a smaller data set. This enables the training to be done relatively quickly. Subsequently, The profile data is used by the CSS scheduler to optimize the program. The cost of this framework relies on the fact that the cost of training is amortized over the number of times the optimized program can be executed, and thus benefiting from the profile-driven compiler framework. This framework is becoming more common with the increasing demand for application specific embedded processors, which require application specific compiler optimizations. Although this may seem expensive in traditional compilation environments, this is more than justified within the embedded systems development environment.

4.2 Hardware Parameters and Workload

The processor model for the simulator was an EPIC processor model, which is similar to the Intel Itanium processor. Its VLIW nature resembles many state-of-the-art embedded microprocessors and microcontrollers, such as DSP-based microprocessors. The processor in our experiments contained 4 integer functional units, 2 floating point units, 2 memory units, and 1 branch unit. Our machine is equipped with various register files: general purpose, floating point, predicate, branch target, and control. All of the files except the branch target file have both static and rotating files. The predicate registers support CSS in that they enable if-conversion optimizations and hyperblock formation.

CSS is a scheduling algorithm that is designed to be sensitive to the memory hierarchy, and as a result we chose a variety of cache configurations for our experiments. The chosen parameters are summarized in table 2.

We selected benchmarks from the Spec2000, Olden [17][18], and DARPA-sponsored DIS Benchmarks and Stressmarks [19]. The workload was consistent with our goal of optimizing programs that have regular to irregular memory access patterns. The benchmarks in our workload also exhibit various degrees of resource usage; however, the vast majority suffer from performance degradation due to undesirable data locality.

The chosen platforms and workload provides a rich experimental environment for comparing the instruction scheduling strategies described in the next section.

4.3 The Scheduling Algorithms

Name of Scheduler	Rank Function	Use of Profile Data
CPFS	$CPFSrank(i)$	No
CPFS-P	$CPFSrank(i)$	avg latencies for load instructions are used as edge latencies
CSS	$CSSrank(i)$	avg latencies for load instruction are used rank computation

Table 4: Summary of Schedulers

We compared CSS with two other schedulers as summarized in table 4. The CPFS scheduler was based on the CPFS rank function (Equation 1). As mentioned earlier, this rank function expressed the typical focus of many instruction schedulers. The CPFS scheduler was also the baseline for all of the experiments. The CPFS-P scheduler was a variation of CPFS where we attempted to use profile information within that scheduler, but not by making it part of the rank function. Instead, the average latencies of the operations were used as the latencies on the out-going edges of the load operations within the data dependence DAG. This forced the scheduler to insert delays for load instructions that were consistent with the profiled average latency experienced via the memory hierarchy. Finally, CSS is the algorithm we propose, that indirectly incorporates load latency information for the rank function computation. In this way, the longer latencies are balanced with the available ILP.All CSS experiments used the same values to control the rank function: $\alpha = 0.48, \beta = 0.11, \gamma = 0.23$, and $\delta = 0.18$. These values were determined from the multivariate statistical analysis discussed in section 5.1.

4.4 Results

The performance of the schedulers was based on 4 metrics: Computation Time, Stall Time, Closeness to $etime(i)$, and ILP Efficiency. These metrics are explained below:

Computation Time: Measures the number of cycles required to complete the computation.

Stall Time: Measures the number of cycles caused by stalls due to memory access delays.

Closeness to $etime(i)$: Captures how early operations are being scheduled relative to the earliest possible schedule time. The

Bench	Description	Pointer Data Structures
Bisort	Bitonic Sort	Binary tree
Health	Simulate Colombia Health System	Quadtree & lists
MST	Minimum Spanning Tree	Array of lists
Power	Power Pricing	Quadtree
Treeadd	Tree walking	Binary tree
Update	Pointer chasing w/ updates	Updates small blocks at random locations
Matrix	Iterative conjugate gradient	Dynamic indirect arrays
Neighborhood	Calculate image texture measures	Dynamic arrays of records
Transitive 800 nodes/5000 edges	All-pairs-shortest-path	Adjacency matrix
Data Management	DBMS Processing	Dynamic arrays of record
179.art	Adaptive Resonance	Dynamic array of records
164.gzip	Gzip compression	Dynamic array of records
181.mcf	Combinatorial Optimization	Dynamic arrays of records

Table 3: Benchmark Suites. Description/characteristics, and data input characteristics for the Olden, DIS, & SPEC benchmark suites

Benchmark	CPFS-Baseline Computation (millions of cycles)	CSS	CPFS-P
Data Man	1,659	-10.51%	35.17%
Matrix	1,731	-22.77%	10.32
Update	731	-18.23%	-0.98%
Neighborhood	1,256	-21.22%	56.34%
Bisort	2,970	-5.13%	50.59%
Health	109	-8.05%	166.78%
Mst	5,581	-3.82%	10.65%
Power	6,267	-10.59%	25.67%
Treeadd	23,383	-4.15%	98.57%
179.art	412,048	-5.49%	60.23%
164.gzip	296,710	-6.5%	87.34%
181.mcf	1,504	-5.65%	75.34%

Table 5: Changes to Computation Time

Benchmark	CPFS-Baseline Stalls (Millions of cycles)	CSS
Data Man	23,428	-25.63%
Matrix	25,331	-37.41%
Update	578,913	-81.62%
Neighborhood	80,074	-33.78%
Bisort	6,944	-3.32%
Health	97	-10.89%
Mst	4,469	-2.67%
Power	2,721	-2.02%
Treeadd	16,128	-2.68%
179.art	29,2450	-4.64%
164.gzip	144,178	-10.81%
181.mcf	743	-3.93%

Table 6: Changes to Stall Time

earliest possible schedule time is constrained by data dependences, as well as block and branch boundaries. This metric approaches 0 for schedulers that are producing compact schedules (i.e. optimistic schedulers).

ILP Efficiency: Measures how the amount of ILP being extracted by the scheduler during the execution of load operations. It is a ratio of actual ILP to the available ILP given an infinite resource machine.

Each of the following subsections focuses on one of the above metrics.

4.4.1 Computation & Stall Times

The data in the tables 5 and 6 was collected over several experiments using the various memory hierarchy configurations. These tables summarize the results of how the schedulers affect execution time of the program. The execution time is divided into two components: The computation time and the stall time. The data in table 5 shows that CSS reduced the computation time of programs in a significant way. The reduction in computation time is due to the increase in the amount of ILP exploited by the CSS rank function which will be supported by the ILP efficiency metric. The CPFS-P algorithm shows an increase in computation time in every instance. The CPFS-P algorithm will not uncover any more ILP than the CPFS algorithm; however, adding longer latencies to the edges of the DAG during scheduling will increase the length of the schedule. This increase in computation time is the result of an increased schedule length and no additional ILP to compensate for for the increase. Consider the following example:

```
0: ld r1, r3 ; Loading value
1:
2: add r5, r1, r2 ; CPFS scheduled time
3:
4:
5: add r5, r1, r2 ; CPFS-P scheduled time
```

Figure 5: Using latencies in CPFS scheduler

The code fragment in figure 5 shows that the time between the load and the use was increased to 5 cycles by CPFS-P from 2 cycles used by CPFS. Now in order to not increase the execution time, these 5 cycles must be filled with useful operations. Since the rank function is the same with CPFS and CPFS-P, the order in which operations are selected is not changed. Therefore, the ordering of the operations generated by the CPFS and CPFS-P schedulers will be identical with the exception of the increased latencies of load operations in the CPFS-P schedule. The result is an increase in the schedule length and execution time.

The data in table 6 summarizes the effect of the schedulers on stall cycles, which is the second component of the execution time. CSS heavily impacts the stall cycles, with reductions as high as 81% with an average of 40%. The reduction in stall cycles is a direct result of the enhanced placement of the load operations.

A subset of the olden benchmarks did not exhibit the performance gains found in the DIS and Spec benchmarks. CSS was designed to exploit ILP, and it properly moved load instructions to provide more slack between the load and its use. However, when CSS was not able to extract ILP from the benchmark, the slack could not be made available without lengthening the schedule. We found that many of Olden and SPEC benchmarks have less available ILP as seen with the ILP Efficiency metric.

4.4.2 E-Time Closeness

Benchmark	E-time-Closeness (CPFS)	E-time-Closeness (CSS)	Difference (# of cycles)
Data Management	1.11	1.56	0.45
Matrix	0.62	0.88	0.26
Update	0.61	0.7	0.09
Neighborhood	1.52	1.75	0.23
Bisort	0.18	0.18	0
Health	0.09	0.14	0.05
Mst	0.09	0.14	0.05
Perimeter	0.09	0.14	0.05
Treeadd	0.09	0.14	0.05

Table 7: This table summarizes etime(i) closeness metric to determine if load operations are being scheduled as early as possible within data dependence, block/branch boundaries, and resource constraints

Table 7 contains the data for the E-Time Closeness metric. This metric was developed to determine whether or not load operations were being scheduled as early as possible. If load operations are being scheduled as early as possible then they cannot be scheduled earlier due to data dependences, block and branch boundaries, or resource constraints. The exact nature is beyond the scope of this paper. As values approach 0, then it can be concluded that operations are being scheduled as early as possible without violating any of the above constraints. In comparing CPFS and CSS, it is apparent that both schedulers schedule instructions very close to the e-time. If it were the case that CPFS was scheduling operations further from their e-time, then it could be argued that if CPFS simply scheduled operations earlier within the slack time, then it would compete with CSS. However, the data suggest that the operations need to be reordered in a way to balance the masking of load latencies and ILP.

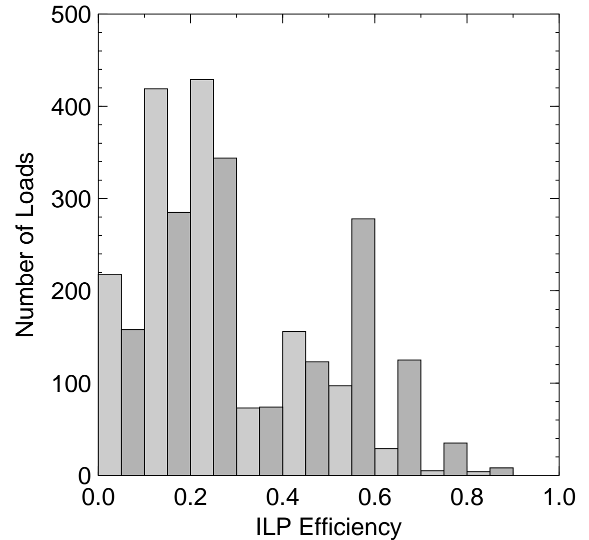

Figure 6: ILP Efficiency for SPEC. Compares the CPFS algorithm(light) with the CSS algorithm(dark).

4.4.3 ILP Efficiency

The graphs in figures6, 8 and 7 compare the ILP Efficiency of CPFS and CSS for each of the benchmark suites. The y-axis is the quantity of load operations, and the y-axis is the percentage of functional units used by the schedule while load operations are executing. The trend in all of the benchmark suites is that CSS is able to extract more ILP on average during the execution of load operations. Overall there is an average of 11.5% improvement.

5. CORRELATION STUDY AND PROFILE STABILITY

The rank function proposed here has 4 components, and each component is accompanied by a scaling factor. The scaling factors represent the relative weight the 4 components should have in influencing the priorities of operations during scheduling. We wanted to use a systematic methodology for determining the values of these scaling factors and settled on multivariate statistics as the basis. Upon using statistics for determining weights, we discovered its usefulness in determining the reliability of the profiled data. The next few subsections address the use multivariate analysis, and the use of linear regression techniques for profile stability.

5.1 Background: Multivariate Analysis

The unknowns in the $CSSrank(i)$ and $CPFSrank(i)$ are the α, β, γ, and δ factors. Originally these factors were chosen by hand optimization based on several experiments. However, this is an impractical method of determining the values of these factors. Therefore; it is necessary to use our profile stability methodology to improve the reliability of using these factors across a range of inputs and programs.

5.1.1 Multivariate Analysis Formulation

Multivariate statistics are an extension of univariate and bivariate statistics, where correlations are found between variables representing a complex data set. The set of variables may contain ndependent variables (IVs) and/or dependent variables (DVs). Inde-

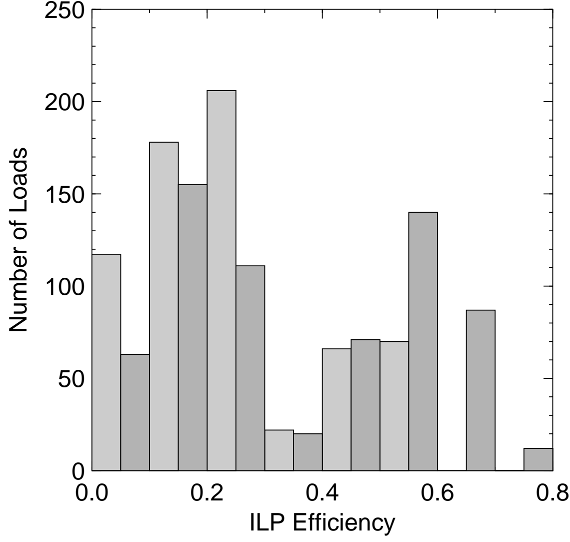

Figure 7: ILP Efficiency for DIS. Compares the CPFS algorithm(light) with the CSS algorithm(dark).

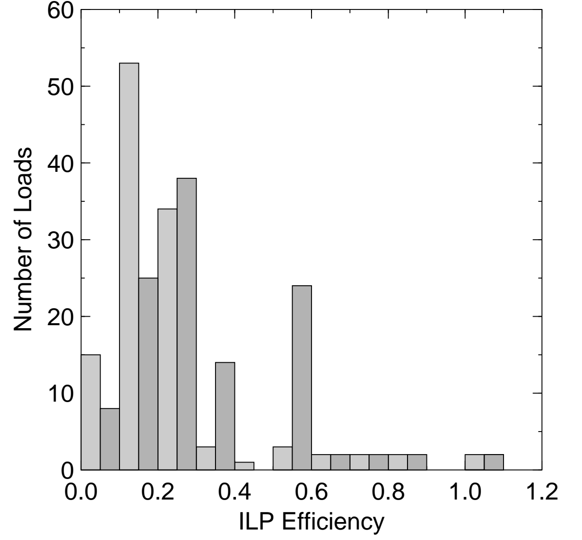

Figure 8: ILP Efficiency for Olden. Compares the CPFS algorithm(light) with the CSS algorithm(dark).

pendent variables (IVs) are variables that are used to predict dependent variables (DVs); therefore, IVs are oftentimes considered predictors and DVs are oftentimes considered predicted variables.

The multivariate analysis we have chosen is based on the following formulation. We are given a set of IVs, $X_1, X_2, X_3, ..., X_k$ that are linearly combined to predict the DV, Y':

$$Y' = A + B_1\ X_1 + B_2\ X_2 + ... + B_k\ X_k \tag{7}$$

The B_i are the regression coefficients. These are a measure of the strength of the association between X_i and Y'. We can represent this entire linear equation as a vector $\vec{z}$. Now instead of points x and y, we will consider vectors $\vec{z}$ and $\vec{w}$. The covariance relationship is now:

$$S_{zw} = \frac{\sum(\vec{z}_i - \bar{\vec{z}})(\vec{w}_i - \bar{\vec{w}})}{n-1} \tag{8}$$

Similarly, the correlation can be computed as follows:

$$r_{zw} = \frac{S_{zw}}{\sqrt{S_z{}^2 S_w{}^2}} \tag{9}$$

The r_{zw} correlation is a matrix of correlations between all x and y pairs. The column of the matrix corresponding to the DV dictates the strength of the relationship of IVs to the DV.

5.2 Using Multivariate Analysis to Support CSS

We build on this formulation and introduce matrices to find the relationship between the individual IVs. The relationship between the individual IVs is particularly important because it will unveil the alues that should be assigned to the α, β, γ and δ CSS scaling factors.

For CSS we decided to use the following variables:

- *Latency*: This is a continuous IV that represents the average latency of all load instructions within the program
- *Fan − Out*: This is a continuous IV that represents the average fan-out for instructions within the program.
- *Height*: This is a continuous IV that represents the average length or height across all regions in the program.
- *Predecessor_Latency*: This is a continuous IV that represents the average latency of predecessor nodes in the program.
- *Time* : this is the DV, and it represents the actual execution time.

These variables directly relate to the factors needed by the CSS rank function. We collected data on over 2000 simulations spanning the collection of benchmarks and varying architectural parameters. We then used SAS to analyze the data, and generate the correlation matrix in table 8. The correlation matrix reveals the relative strength of the effect of each of the factors.

The actual scaling factors for CSS can be computed directly from the ast column of the correlation matrix. The last column of the matrix eveals how each of the independent variables affects the dependendent variable, *Time*. By normalizing the values in this column to fall between 0 and 1, and to sum to 1, we can obtain the relative importance of each independent variable. In this case, the result shows that $\alpha = .48, \beta = .11, \gamma = .23$, and $\delta = .18$.

5.3 Profile Stability

Stability of profile data is a concern for compiler designers that use program execution profile data to drive compiler optimizations.

none	latency	fanout	height	PredLat	log(Time)
latency	1.0	0.54583	0.24990	0.93812	0.19096
fanout	–	1.0	0.33410	0.50981	0.09290
height	–	–	1.0	0.13475	0.40680
predLat	–	–	–	1.0	0.14960
log(Time)	–	–	–	–	1.0

Table 8: correlation matrix for CSS

Profile data stability is the ability to rely on data collected from a set of profiled program executions to correlate with other non-profiled program executions. This is needed because it is impossible to collect profile data for all possible data inputs of a given program, and it is impossible to execute all programs. Therefore, the data collected during a set of profiled executions needs to be applicable to some superset of data input and program combinations.

For this paper, a linear regression analysis was performed on a data collected from experiments using the Transitive benchmark from the DIS suite. The Transitive benchmark performs the transitive closure on a graph. We chose this benchmark because there were 6 supported input data sets that vary the data input size. The following are the details to this experiment:

inputs	number of vertices	number of edges
input1	800	5000
input2	800	500,000
input3	1000	5000
input4	1250	5000
input5	1250	500,000
input6	1250	900,000

Table 9: Input sets used for DIS Transitive benchmark

1. The benchmark was executed using 6 different input sets. The input sets are detailed in table 9. The table shows that the total size of data being processed is significantly diverse.

2. Two cache configurations were chosen from table 2. The cache configuration of the first and third rows were chosen because of the increase in L1 and L2.

3. The CPFS and CSS schedulers were used.

4. The detailed profile information includes the miss and hit ratios in respect to L1, the number of misses and hits in respect to L1, and the average latency for each unique load operation. This data created a table of over 6000 entries. A snippet of this table is shown in table 10. As you can see, a load operation executing on the same memory heirarchy is only slightly affected by the size of the input data.

An examination of the distribution of the miss ratios for the load operations, found that the distribution was sparse containing spikes at the very high miss rates and at the very low miss rates. Very few load operations fell in between. Due to this data skewing, we partitioned the space and performed the regression using the recomputed means as least square means for each partition. Performing the linear regression in these partitions shows that statistically, load

inputs	least squares means
input1	0.7166
input2	0.7166
input3	0.6626
input4	0.6539
input5	0.6540
input6	0.6540

Table 11: Least squares means for load operations considered to have high miss rates over the various input sets.

inputs	least squares means
input1	0.0004
input2	0.0004
input3	0.0000
input4	0.0004
input5	0.0004
input6	0.0004

Table 12: Least squares means for load operations considered to have low miss rates over the various input sets.

operations are affeted very little by the data inputs when the cache remains constant. These results are shown in tables 11 12.

The results of this experiment is representative of the other DIS benchmarks used in this research. We anticipate the same effect from the other suites based on a coarse grain profile analysis presented in [20]. The furher breakdown and discussion of this topic is for future work.

6. CONCLUSIONS

We have presented a new scheduling algorithm called Cache Sensitive Scheduling (CSS). CSS is a scheduling algorithm that is based on the rank function framework. This enables CSS to be easily extended and integrated with generic compilers. The need for CSS is due to the increased responsibities given to the compiler by emerging EPIC COTS technologies. These processors are accompanied by ISAs that allow the compiler to build instruction schedules as well as move data within the memory hierarchy. The CSS algorithm takes advantage of the latency and predication features of VLIW/EPIC processors to generate schedules that are sensitive to the latency requirements of the load instructions. Contrary to conventional schedulers, CSS is able to find the best load-to-use gap in the schedule, and avoid the problems associated with data arriving either earlier or later than it is needed.

The CSS algorithm can be used as an enhancing optimization in conjunction with high-level loop optimizations and data reorgani-

input	cache config	op	hits	hit rate	misses	miss rate	avg. latency
1	0	2	1248084	0.020016	61105017	0.979984	48.570461
2	0	2	1211738	0.020023	59305630	0.979977	31.326433
3	0	2	1957596	0.01598	120545616	0.98402	48.404846
4	0	2	3070749	0.012767	237443609	0.987233	48.72382
5	0	2	3003985	0.012829	231147464	0.987171	32.710495
6	0	2	2986222	0.01276	231043018	0.98724	48.735279

Table 10: fragment of profile data collected on the Transitive benchmark for a single load operation on the same cache, but varying input.

zation techniques. In this framework, CSS would be able to address locality issues that are not addressed by high-level with loop and data optimizations.

To accompany the CSS algorithm, we have presented a multivariate statistics framework that is used as a stability technique to ensure that the rank function accounts for all of the factors that can affect the execution time of the program at the scheduling level. In this context, the statistical analysis is used to determine the values of the scaling factors, α, β, γ, and δ. In addition, we used regression analysis to determine the stability of the profile information in respect to input data. Our experiments found that the behavior of individual load operations is largely unaffected by the input data set size or complexity. Further investigation will be performed to further encourage and support profile-based compilation in the context of memory focused compiler optimizations.

Acknowledgements

We thank Felicia P. Hardnett, Biostatistician with Centers for Disease Control for assisting us in formulating and implementing the multivariate analysis.

7. REFERENCES

[1] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *25th Annual International Symposium on Microarchitecture*, 1992. [Online]. Available: citeseer.nj.nec.com/mahlke92effective.html

[2] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. mei W. Hwu, "Superblock formation using static program analysis," in *26th International Symposium on Microarchitecture*, 1993, pp. 247–255.

[3] D. R. Kerns and S. J. Eggers, "Balanced scheduling: instruction scheduling when memory latency is uncertain," *ACM SIGPLAN Notices*, vol. 28, no. 6, pp. 278–289, 1993.

[4] K. V. Palem and B. B. Simons, "Scheduling time-critical instructions on RISC machines," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 4, pp. 632–658, September 1993.

[5] T. Johnson and S. Abraham, "Load sensitive scheduling," hP Labs.

[6] T. Consortium, "Trimaran project homepage," www.trimaran.org.

[7] V. Kathail, M. Schlansker, and B. Rau, "Hpl-pd architecture specification: Version 1.1," Hewlett Packard, Palo Alto,CA, Tech. Rep., 2000.

[8] F. J. Sánchez and A. González, "Cache sensitive modulo scheduling," in *International Conference on Parallel Architectures and Compilation Techniques*, 1997, pp. 338–348. [Online]. Available: citeseer.nj.nec.com/191954.html

[9] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, "Compiler-directed data prefetching in multiprocessors with memory hierarchies," in *Proceedings 1990 International Conference on Supercomputing, ACM SIGARCH Computer Architecture News*, vol. 18, 1990, pp. 354–368. [Online]. Available: citeseer.nj.nec.com/42264.html

[10] A. Klaiber and H. Levy, "An architecture for software-controlled data prefetching," in *Proceedings of 18th ISCA*, 1991, pp. 43–55.

[11] 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*, 1992, pp. 62–73.

[12] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 40–52.

[13] T. Ozawa and et. al, "Cache miss heuristics an preloading techniques for general-purpose programs," in *Proceedings of 28th Annual International Symposium on Microarchitecture*, 1995, pp. 243–248.

[14] *Using Multivariate Statistics*. Harper Collins College Publishers, 1996.

[15] J. Edler and M. Hill, "Dinero iv trace-driven uniprocessor cache simulator," 1998.

[16] W. Y. Chen, S. A. Mahike, N. J. Warter, S. Anik, and W. W. Hwu, "Profile-assisted instruction scheduling," *International Journal for Parallel Programming*, vol. 22, no. 2, pp. 151–181, April 1994.

[17] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, "Supporting dynamic data structures on distributed-memory machines," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 2, pp. 233–263, March 1995. [Online]. Available: citeseer.nj.nec.com/rogers95supporting.html

[18] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," *ACM SIGPLAN Notices*, vol. 33, no. 11, pp. 115–126, 1998. [Online]. Available: citeseer.nj.nec.com/roth98dependence.html

[19] DARPA, “Data intensive systems benchmark suite,” www.aaec.com/projectweb/dis/.

[20] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe, “Convergent scheduling.” [Online]. Available: citeseer.ist.psu.edu/lee02convergent.html