Center For Research in Embedded Systems and Technol gi

Technical Report

Cache Senstive Instruction Scheduling

Charles .R. Hardnett, Rodric M. Rabbah, and Krishna V. Palem
Georgia Institute of Technology
Atlanta, Georgia

Weng Fai Wong
National University of Singapore
Singapore

CREST-TR-01-003
GIT-CC-01-15
June 2001

Cache Sensitive Instruction Scheduling

Charles R. Hardnett, Rodric M. Rabbah, Krishna V. Palem
hardnett@cc.gatech.edu, rabbah@ece.gatech.edu, palem@ece.gatech.edu
Center for Research on Embedded Systems and Technology
Georgia Institute of Technology

Weng Fai Wong
wongwf@comp.nus.edu.sg
Department of Computer Science
School of Computing
National University of Singapore

Abstract

The processor speeds continue to improve at a faster rate than the memory access times. The issue of data locality is still unsolved, and continues to be a problem given the widening gap between processor speeds and memory access times. Compiler research has chosen to address this problem in many directions including source code transformations of loops, static data reorganization, dynamic data reorganization, and optimized instruction scheduling. This paper presents Cache Sensitive Scheduling(CSS). CSS is an instruction scheduling algorithm that relies on a rank function to choose operations in the proper order. The CSS rank function is built on the latency of the operation, the impact of this operation on other operations in the program, and the latency of operations that this operation is dependent on in some way. Our premise is based on the hypothesis that careful scheduling of load instructions can increase ILP and decrease execution times by overlaying the latency of load instructions with other useful instructions. This is particular useful on EPIC and VLIW types of machines, where increased ILP is always a benefit. Our rank function is designed to find these opportunities and exploit them. We will show that these techniques can be used to improve the performance of programs with a *range of memory access patterns spanning the regular to irregular.*

1. Introduction

The processor speeds continue to improve at a faster rate than the memory access times. The memory subsystem continues to be a major bottleneck in the performance of modern processors. Compiler research has chosen to address this problem in many directions including source code transformations of loops, static data reorganization, dynamic data reorganization, and optimized instruction scheduling. This paper addresses the problem via instruction scheduling. Traditional schedulers use the hit latency of load instructions when placing these loads in the schedule. Optimistic schedules are susceptible to stalled pipelines on load cache misses. If you decide to use the miss latency in the schedule, then the schedule is unnecessarily lengthened. Cache Sensitive Scheduling(CSS) is our solution, where we use average latencies of load instructions as an integral part of the rank function to properly control the scheduling of load instructions. Section 2(The Rank Function) provides details on the formulation of the rank function.

Our experimental framework is Trimaran[8]. Trimaran is a parameterized compilation system. The system includes a compiler for EPIC processors and a simulator for providing cycle-by-cycle accuracy of the program executions. Both the compiler and the simulator use a machine description file for targeting optimizations and providing accurate simulation. The machine description allows us to vary processor and cache parameters. Section 3(Experimental Framework) covers the implementation of CSS within Trimaran.

Our results show that it is possible to develop a rank function that can be used to hide the memory latency affects of memory operations during instruction scheduling. We show that ILP can be increased by over 150% while decreasing computational cycles by as much as 22%. We are targeting our algorithm towards EPIC type processors such as the Intel IA-64[ref]. We feel that these results are also applicable to the quasi-irregular and irregular applications. Section 4(Experimental Results) will demonstrate the impact of CSS using various performance metrics.

The final two sections of this paper will show where our work fits in the body of research that was previously presented, and where we feel this work can continue to have impact in this area of research.

2. The Rank Function

Cache Sensitive Scheduling(CSS) is our solution, where we use average latencies of load instructions as an integral part of the rank function to properly control the scheduling of load instructions. Our algorithm is based on greedy list scheduling with static priorities given to each operation via our rank function [1]. All operations are given a priority using this rank function; however, the rank function treats loads and other longer latency operations differently than other operations. Longer latency operations that affect the greatest part of the program will be given priority over other operations. In most cases, these are the load operations with a high incidence of data cache misses.

Formally defining our rank function relies on several definitions:

Directed Graph G def { V, E} representing a region of code; basic block, hyperblock, etc.

V def {i | i is an instruction}

E def {(i,j) | i,j ∈ V}

preds(i) def {j | j ∈ V and (j,i) ∈ E}

succs(i) def {j | j ∈ V and (i,j) ∈ E}

lat(i,j) def non-negative attribute for an edge representing the latency of the instruction.

exec(i) def non-negative value for the number of times instruction i is executed during a profiling.

dmc(i) def non-negative value for data miss count for instruction i in regards to the L1 and L2 caches.

miss_ratio(i) def non-negative value for data miss ratio for instruction i. Computed as dmc(i)/exec(i).

hit_latency def expected latency for hits on the given architecture.

miss_latency def expected latency for the misses on the given architecture.

etime(i) def the earliest time an op can be scheduled based on its dependences, and scheduling on an infinitely wide machine.

Our rank function is based on the following quantities for operation *i*:

height(i) def $w^{+}(j,i)$ *where j is the sink node*

fanOut(i) def the number of outgoing edges, where $(i,j) \in$ *and* $j \in succs(i)$.

*avgLatency(i) def miss_lat(i)*miss_ratio(i) + hit_latency(i)*hit_ratio(i)*

We combine these quantities in the following fashion[1]:

$rank(i) = \alpha * height(i) + \beta * fanOut(i) + \gamma * avgLatency(i) - \delta * max\{avgLatency(j) \mid j \in preds(i)\}$

We will now discuss each component of the rank function, and how its presence affects the priorities of the operations.

1. The first term height(i) refers to the critical path. It will prefer critical path operations over non-critical path operations.

1 The α, β, δ, and γ factors were chosen based on several runs where these factors were changed to determine their importance. We plan to introduce a analytical method for computing these values in the future.

2. The fanOut(i) component will give preference to operations with a larger number of successors. Operations that directly affect the most operations will most likely affect more operations in the program and are given higher priority than operations with less influence.
3. The avgLatency(i) component will give preference to those instructions that have longer latency requirements i.e. load operations with high miss rates. This allows those instructions to be scheduled earlier than operations will lower latencies.
4. The max{...} term plays the role of demoting operations that are relying on a longer latency operation. These operations must be scheduled later or else they will require the data before it is available. The later scheduling will increase ILP opportunities.

Now lets look at point #4 in more detail.

compute_sign(δ): *if avgLatency (i)* $<$ *max{avgLatency(j)* $\mid j \in$ *preds(i)}*

then sign(δ) *is positive*

compute_sign(δ): *if avgLatency (i)* $\geq$ *max{avgLatency(j)* $\mid j \in$ *preds(i)}*

then sign(δ) *is negative*

The first condition shows that operation i will be pushed down the schedule by the latency of operation j, and it will need to be more than than amount it could be moved up due to its own latency. Therefore the rank of this operation is decreased. The second option is just the opposite, and it will cause the rank to increase.

The rank function is very flexible, addresses all longer latency operations where some ILP may be exploited. The latency characteristics of longer latency load operations will give them preference over all other instructions. If more than one load operation have similar miss rates, then their affect on the rest of the program will be measured by the fan out and height factors. This allows our rank function to seek out the highest priced load as the priority in scheduling.

In the next section we will discuss our experimental framework and results of various simulations using different architectural configurations.

3. Experimental Framework

Our experimental framework is Trimaran[8]. Trimaran is a parameterized compilation system. The system includes a compiler for EPIC processors and a simulator for providing cycle-by-cycle accuracy of the program executions. Both the compiler and the simulator use a machine description file for targeting optimizations and providing accurate simulation. The machine description allows us to vary processor and cache parameters.

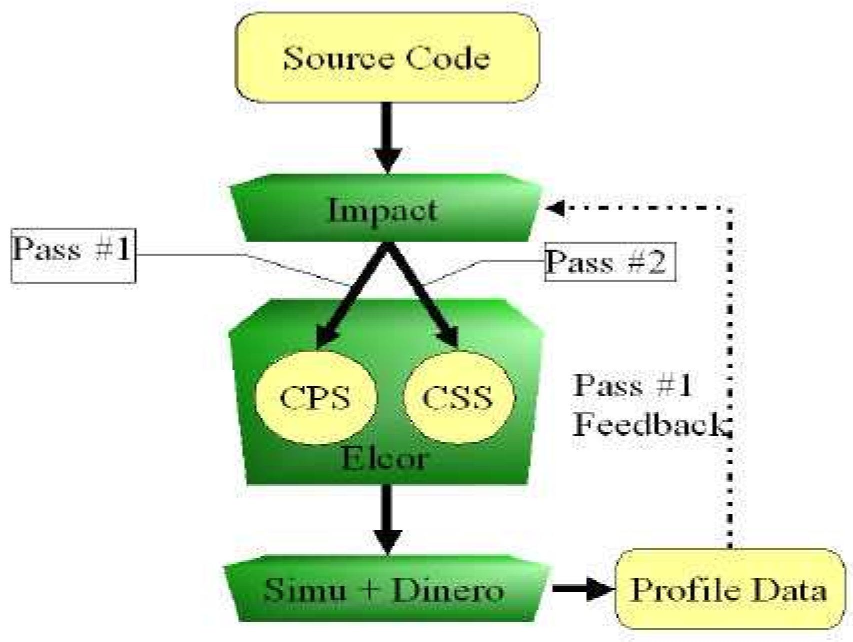

Figure 1Trimaran System for CSS(CPS = Critical Path Scheduler)

Trimaran is built on 2 compilers(see figure 1): Impact and Elcor. Impact is the front-end of the system, which performs basic front-end parsing and analysis in addition to basic block/hyperblock/superblock formation and profiling. The Elcor compiler is the back-end of the system and performs the optimizations, scheduling, and register allocation.

CSS requires 2 passes of compiler system. Pass one is where the standard scheduler is used(in this case is critical path scheduling(CPS)). The scheduled program is then simulated using SIMU(need ref) and Dinero(need ref). The simulation completes and returns cache statistics per memory operation. The statistics for each operation are based on the complete single execution of the program. These statistics include the average miss latency, the hit ratio, the miss ratio, and the number of hits and misses. This information is then fed into the second pass of the compilation.

The second pass of the compilation does not change for Impact. The changes all occur in Elcor. Elcor reads the profile data for each operation, and adds the data as an attribute for the respective operation. CSS is now the scheduler instead of CPS. The CSS algorithm is based on greedy list scheduling. During prescheduling, the scheduler assigns a static priority to all of the operations in the scheduling region(basic block, hyperblock. Etc.). The priorities are assigned based on our rank function, which rely on the profile data. As the operations become data ready, they are sorted in the data ready queue according to the preassigned priorities. The scheduler removes operations from this queue and places them in the schedule.

CPS is based on a rank function as well. Its function is simply choosing the instruction that is considered on the critical path over instructions that are not on the critical path. Given operations i and j, if i and j are on the critical path, then the their heights are compared:

if (height(i) > height(j)) then

```
        operation i is selected
    else
        operation j is selected
```

The critical path scheduler focuses on creating schedules that are closer to the length of the critical path. However, it does not consider exploiting ILP opportunities. In the next section we will show results of experiments that compare the critical path scheduler to the CSS scheduler.

4. The Experiments and Results

The experiments were designed to compare our scheduling algorithm to the critical path scheduling algorithm. We determined how susceptible these two algorithms are to changes in the memory subsystem of the architecture i.e. Quantity of memory units, latency of the memory hierarchy, etc.

The first collection of experiments is based on the an architecture where there are 2 memory units to service memory request. The latency between the processor is C1, C2, and memory are considered typical for today's processors. The C1 data cache is 4-way set associative and 16KB. C2 is a unified 32KB cache that is 16-way set associative.

The second collection of experiments is based on the same architecture, but it only has one memory units.

The last set of experiments were designed to show what happens as the processor speeds increase, and memory access times decrease. We model this phenomenon by increasing the memory latency, which gives the same effect as increasing the processor speeds.

The experiments are based on comparing the effects of CPS and CSS algorithms. The CSS algorithm relies on the these values: $\alpha = 0.15$, $\beta = 0.15$, $\gamma = 0.35$, $\delta = 0.35$. The experiments were run using the Olden Benchmark Suite and the Data Intensive Systems Stressmarks and Benchmarks

Suite[9]. The target machine is an EPIC style processor with 4 integer units, 2 floating point units, 2 memory units, and 1 branch unit. The cache memory system composed 16K data C1 and instruction C1, and 32K C2 cache.

Results of ILP Experiments

We were first interested in the effect of CSS for finding and using as much ILP as possible.

We decided to use a metric that takes into account 2 major aspects of the scheduling problem:

1. Are the load operations scheduled as early as possible?
2. Does the new schedule create more opportunities for ILP?

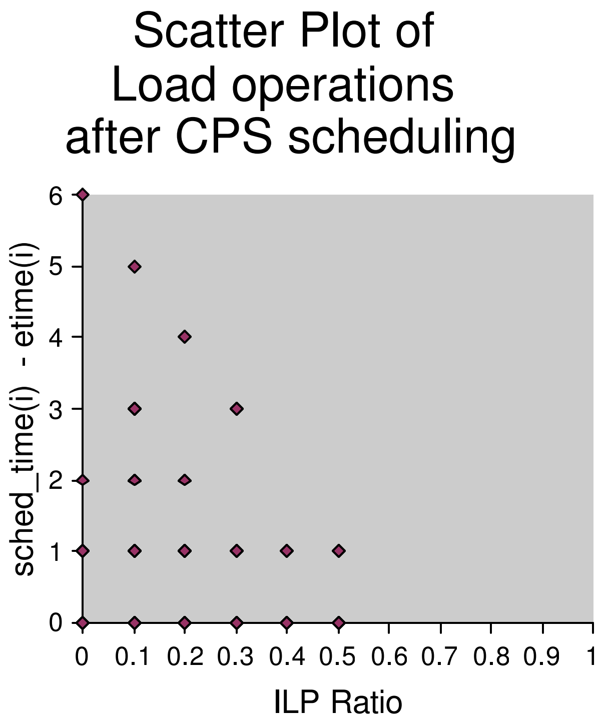

Figure 2Scatter plot shows that no load instruction is using more than 50% of the available ILP

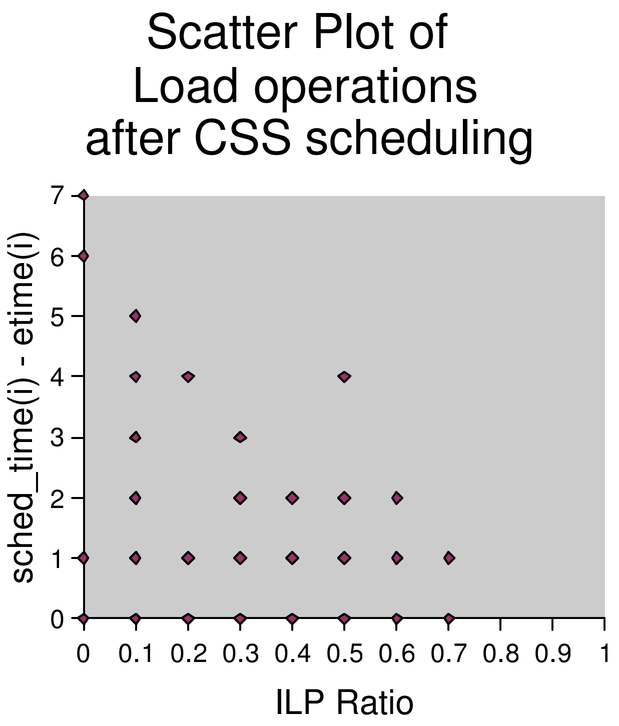

Figure3Scatter Plot shows that CSS is able to have load instructions using 60% to 70% ILP

The first question is key because we do not want the CSS schedule to be longer than the CPS schedule(unless there is some gain there that compensates). The second question addresses the need to overlay the latency of load operations with useful work, and thus increasing the ILP used by the operation. The graphs in Figure 2 and Figure3show the results of scatter plot of load operations. The graph inFigure 2is based on CPS scheduling, while the graph in Figure3 is based on CSS scheduling.

Given load operation i, the x-axis is the ILP ratio(i) and the y-axis is the sched_time(i) – etime(i). It is clear that CSS is moving loads into the 60% and 70% ILP, where CPS is only reaching the 50% ILP usage. The graphs in Figure 4 is a histogram of the same data.

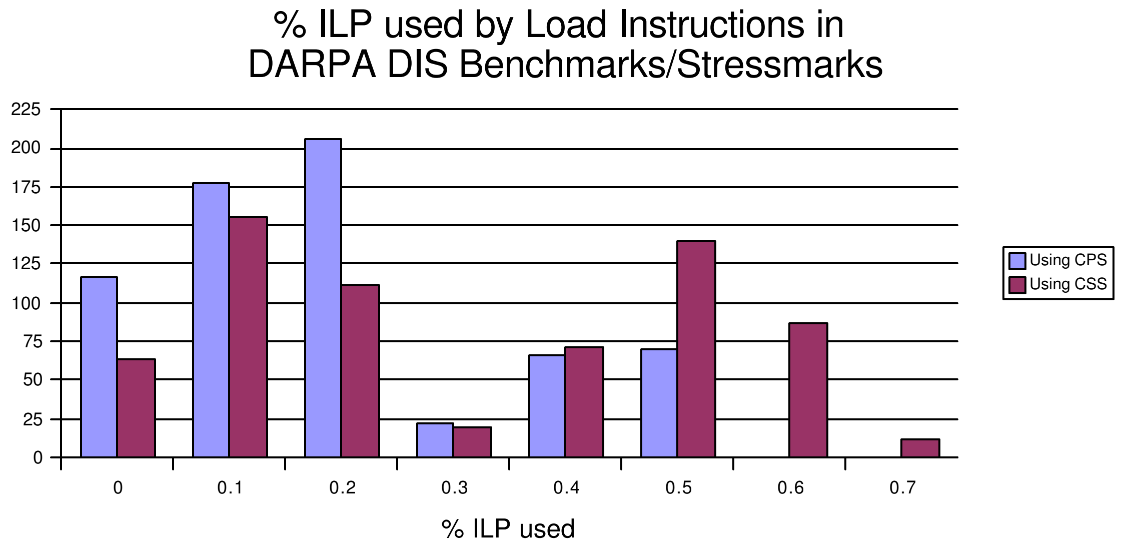

Figure 4Histogram to show how CPS and CSS perform across DIS benchmarks in terms of % ILP used.

The histogram demonstrates that the majority of the load operations in CPS are actually below the 50% line. In the case of CSS, it is clear that the majority of the loads have shifted to 50% and beyond. This shows that our CSS rank function is finding more opportunities than the CPS algorithm. Now the other question that we wanted to address is how well CSS would schedule operations in relation to the etime() of the operation. This is best seen in the table of Figure 5. W We computed the difference, sched_time(i) - etime(i), for each load in each benchmark over a range of simulations with varying machine configurations. These computations make up the column for using CPS versus the column for CSS. The results show that both CPS and CSS typically schedule instructions very close to their early time. In fact, both schedulers are performing at a difference of less than 1 cycle. The results of this table verify that CSS is truly scheduling load instructions as early as possible; and therefore, CSS is not lengthening the schedule unnecessarily to accommodate load latencies.

Benchmark	**E-time-Closeness (CPS)**	**E-time-Closeness (CSS)**	**Difference (# of cycles)**
DataManagement	1.11	1.56	0.45
Matrix	0.62	0.88	0.26
Transitive	0.47	0.65	0.18
Update	0.61	0.7	0.09
Neighborhood	1.52	1.75	0.23
All DIS	**0.87**	**1.11**	**0.24**
Bisort	0.18	0.18	0
Em3d	0	0	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
Tsp	0.09	0.12	0.03
All Olden	**0.09**	**0.12**	**0.03**

Figure 5Shows that there is less than a cycle difference between how early CPS is scheduling the operations, and how early CSS is scheduling the operations

Combining these results we can see how CSS is preferred. CSS is scheduling operations as early as possible, while intelligently placing the load instructions to maximize ILP.

We also looked at the compute cycles of the programs as a measure of how compact the schedules would be. The EPIC machines bundle several operations into one instruction. These operations are expected to execute in parallel within the bundle. Therefore, a more compact schedule implies more ILP. The data in the table of Figure 6shows that CSS shows considerable performance improvements over CPS. CSS is a much better performer on the DIS benchmarks Vs the Olden benchmarks. We believe that the Olden benchmarks did not have as much available ILP as the DIS benchmarks, and this is why there is very little difference. However, CSS is still able to find the little available ILP, and exploit it.

Benchmark	Compute Cycles (CPS)	Compute Cycles(CSS)	% improvement
DataManagement	5276549.13	4722144.13	-10.51%
Matrix	5094812.88	3934834.38	-22.77%
Transitive	2829763.5	2377003.5	-16.00%
Update	519946.38	425174.63	-18.23%
Neighborhood	27661181	21791584.25	-21.22%
All DIS	**8276450.58**	**6650148.18**	**-19.65%**
Bisort	2696996.88	2558675.13	-5.13%
Em3d	4706215	3963937.13	-15.77%
Health	10910665.13	10032766.75	-8.05%
Mst	1423045.13	1368741.25	-3.82%
Perimeter	9515278.88	9179192.88	-3.53%
Treeadd	2515005	2410573.25	-4.15%
Tsp	2056352.63	1974931.38	-3.96%
All Olden	**4831936.95**	**4498402.54**	**-6.90%**

Figure 6Shows CSS with significant improvement on DIS suite, while showing moderate improvements on the Olden suite

Results of the Memory Unit Experiments

We executed a few experiments where we changed the number of memory units from 2 to 1. The purpose of these experiments was to see if the CSS algorithm with 1 memory unit could outperform the CPS algorithm with 2 memory units. This kind of experiment is very important to and processor architects they may trade-off a memory unit for a smarter scheduler in the compiler. The embedded market is very interested in these types of tradeoff, since a reduction in hardware means a reduction in power. Running the CPS algorithm on the Olden benchmarks with the 2 memory unit architecture results in an average of 4238422.43 cycles. Running the same benchmarks using the CSS algorithm on an architecture of 1 memory unit results in 4157105.71 cycles. This shows that the CSS algorithm can outperform the CPS algorithm even using less memory units. We did the same for the DIS benchmarks to find that CPS produced an average of 7157411.8 cycles on 2 memory units, and CSS produced an average of 5974806.8 cycles on 1 memory unit.

The CSS algorithm shows that it is feasible to consider smarter compiler scheduling algorithms, and other locality enhancing algorithms to enable the reduction and/or simplification of memory subsystem hardware. This is an area of research that is particularly promising for the embedded systems community.

5. Related Work

There have been other schedulers designed to address this issue in various ways. Kerns and Eggers[7] developed a scheduler that computes the available ILP for an instruction. This is computed by recursively eliminating the predecessors and successors of the instruction. The result is a set of connected components of instructions that can be executed in parallel with the given instruction. For each of the load instructions in these connected components increase the weight/priority of that load instruction. Their work targets RISC processors and shows a 3% to 18% improvement over a typical critical path scheduler. Sánchez and González[6] propose a way to integrate software prefetching with software pipelining in VLIW architectures. Their approach is to insert prefetch instructions into modulo scheduled loops. Others have also studied prefetch insertion[5][4][3][2]. Our CSS algorithm is designed with VLIW machines in mind, and produces code that maximizes ILP, and at the same time we do not treat all loads alike. A load that hits most of the time will not be scheduled in any special way, even if there is some ILP available. Since, there is no long latency to overlay this there is not much gain in scheduling this operation earlier.

Johnson and Abraham [10] 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[11] developed an analytical algorithm for identifying loads that will cause misses during execution, and they complemented this algorithm with a scheduling algorithm that targets these loads for optimization. Their work focused on scientific applications with regular access patterns via arrays. Our CSS algorithm is targeted towards the quasi-regular and quasi-irregular applications.

6. Conclusions & Future Directions

We have presented a new scheduling algorithm called Cache Sensitive Scheduling(CSS). We have shown how CSS relies completely on a rank function that systematically selects loads to be scheduled in an optimal way to increase the performance of the program. We have shown that CSS increases the ILP used by the program, and at the same time compresses the schedule. The result is a program that executes with higher utilization of the functional units, and hides the latency of load misses by overlaying this latency with the execution of useful instructions. Finally, we have argued that CSS can be used to eliminate memory units in systems where multiple memory units is not feasible. The performance of CSS with one memory unit rivaled that of CPS using two memory units.

We recognize that one of the weaknesses of the current method is that it requires a profiled execution of the program to enable the CSS algorithm to use the latencies of the load operations. We will be investigating analytical methods for determine the appropriate multiplicative factors for controlling the parts of the CSS rank function. We also recognize the need to take advantage of the speculative load operations and prefetch operations found in EPIC instruction sets. In addition, we would like to do a more thorough investigation of how CSS and other advanced compiler algorithms can help reduce the complexity of hardware and/or eliminate parts of hardware. Finally, we plan to combine a collection of locality analysis methods and locality enhancing algorithms into a unified systematic approach to handling locality issues during compilation.

References

[1] Daniel R. Kerns and Susan J. Eggers, *Balanced Scheduling: Instruction Scheduling When Memory Latency is Uncertain*, in , 1993

[2] E.H. Gornish, E.D. Granston, and A.V. Veidenbaum, *Compiled-Directed Data Prefetching in Multiprocessors with Memory Hierarchy*, in *17th ISCA*, 1990

[3] A.C. Klaiber and H.M. Levy, *An Architecture for Software Controlled Data Prefetching*, in *18th ISCA*, 1990

[4] T.C. Mowry, M.S. Lam, and A. Gupta, *Design and Evaluation of a Compiler Algorithm for Prefetching*, in , 1992

[5] David Callahan, Ken Kennedy, and Allen Porterfield, *Software Prefetching*, in *IV-ASPLOS*, 1991

[6] F. Jesús Sánchez and Antonio González, *Cache Sensitive Modulo Scheduling*, in *IEEE*, 1997

[7] Krishna V. Palem and Barbara B. Simons, *Scheduling Time-Critical Instructions on RISC machines*, in *ACM Symposium on POPP*, 1990

[8] The Trimaran System, www.trimaran.org
[9] Data Intensive Systems Benchmark Suite, www.aaec.com/projectweb/dis/
[10] Teresa Johnson and Santosh Abraham, *Load Sensitive Scheduling*, HP Labs Technical Report, 1994
[11] T. Ozawa, Y. Kimura, and S. Nishizaki, "Cache miss heuristics and Preloading Techniques for general-purpose programs", MICRO-28, 1995.