Some Cache Optimization with Enhanced Pipeline Scheduling

Seok-Young Lee, Jaemok Lee, Soo-Mook Moon
{sylee,jaemok,smoon}@altair.snu.ac.kr
School of Electrical Engineering and Computer Science, Seoul National University, Korea

Abstract

Reducing the data cache stalls is getting more important as the gap between the CPU speed and the memory speed continues to grow. Many compiler-based techniques including prefetches have been proposed to mitigate the problem for numerical loops, but they are not very applicable to non-numerical integer loops. One simple idea applicable even to those loops is separating cache-missing loads and their uses by as many as the cache miss penalties by scheduling the loads earlier. Unfortunately, naïve code motion of those loads would not work since they are likely to be stuck at the loop boundary before being separated enough from their uses. Moreover, moving the loads requires other instructions defining their operands to move as well, which would also block separation. In order to overcome these limitations, we propose separating them even across loop backedges and complex control flows. This can be implemented by employing the code motion techniques of *enhanced pipeline scheduling*, which can allow such separation. Our experimental results on the Itanium and the Open-64 show that the proposed technique can reduce the stalls, increasing the performance tangibly, for some integer benchmarks that suffer from the data cache misses seriously.

1. Introduction

The CPU stalls caused by data cache misses take a significant portion of the running time for many programs. Since the gap between the CPU speed and the memory speed continues to grow, cache optimization has become an important issue in the compiler research. Many compiler-based cache optimization techniques have been proposed, such as increasing locality within loops [20,21], adding prefetches for array-accessing loops [10,11,12], and dynamic runtime optimizations [18,19]. Many of these techniques involve address determination using array index-based calculations, thus particularly effective for numerical, floating-point programs which process a large amount of data in the arrays.

Unfortunately, these techniques are not very applicable to reducing cache stalls for non-numerical integer programs. These programs often include loops where it is not easy to determine the address of loads with a low overhead (e.g., pointer-chasing loops). Moreover, integer loops often include complex control flows which make the address determination even harder. These characteristics make those array index-based optimizations difficult to employ.

For integer loops, it might be better to separate hot cache-missing loads and their uses by as many as the cache-miss cycles, instead of employing optimizations using index-based calculation. This can be simply achieved at the instruction scheduling phase of the compiler optimization, by setting the latency of those hot loads to their cache-miss cycles instead of the cache-hit cycles, which would delay the scheduling of those uses from hot loads.

Unfortunately, this would not work as we expect, because naïve code motion of those hot loads would make them stuck at the loop boundary before being separated enough from their uses if they

are located near the loop header. Moreover, moving the loads requires other instructions defining their operands to move as well, which would even limit enough separation. Finally, even if we delay the scheduling of uses, we cannot avoid a stall if we cannot fully schedule those newly-added slack cycles by other (useful) instructions.

In order to overcome these limitations for separating the hot loads and their uses, we propose employing code motions across loop iteration boundaries. That is, we move those hot loads (and the instructions defining their operands as well) across the loop backedges so that they can be separated enough from their uses. Actually, this means executing the hot loads an iteration earlier than their uses, lengthening their distances. We can even move the loads more than once across the backedge (even passing thru their uses with a different target register), if executing the hot loads multiple iterations earlier is needed to lengthen the distance enough. In fact, this is exactly what software pipelining does. Therefore, we propose implementing the idea in the context of software pipelining, for the purpose of separating hot loads from their uses as well as for extracting instruction-level parallelism (ILP). For this we employ *enhanced pipeline scheduling*, a software pipelining technique based on code motions across loop backedges. Our preliminary experiments with the Open-64 compiler for the Itanium show some promising results for a subset of integer programs that suffer from cache misses seriously.

The rest of the paper is organized as follows. Section 2 motivates the main idea of the proposed technique with some characteristic of hot cache-missing loads. Section 3 reviews briefly enhanced pipeline scheduling. Section 4 describes the proposed technique with its heuristics. The experimental results are described in Section 5. A summary follows in Section 6.

2. Separating Hot Cache-Missing Loads and their Uses across Loop Iterations

In this section, we motivate the main idea of the proposed technique with an example. We describe some observation of the hot cache-missing loads and explain why the proposed technique is not trivial to implement, although the idea is simple.

2.1 Hot Cache-Missing Loads

Most CPUs these days employ a *non-blocking* cache [3]. When there is a cache miss caused by a load, the CPU does not stall right away but continues to proceed until the use of the load is encountered; at that point the CPU stalls until the data comes to the CPU. One popular way to exploit the non-blocking cache is employing the *prefetch* instruction [4,9,8,10,11,12]. For a hot cache-missing load, the compiler adds a prefetch that accesses the same cache line, so that the prefetch executes and misses the cache earlier than the load. This can make the cache-line available, hopefully by the time when the load executes, obviating or reducing the CPU stalls. Prefetches are known to be highly effective in array-based, numerical loops since it is relatively easy to calculate the array address of prefetches to be located in earlier iterations, based on loop indexes.

Another way to exploit the non-blocking cache is separating the hot load and its uses enough, hopefully by as many as the cache miss cycles. This can make the cache miss latency be elapsed by the time when the use executes, obviating or reducing the CPU stalls [5,9,6,7,8]. We take this approach for handling integer loops where the calculation of load addresses is not straightforward. Unlike previous work, however, we extend the separation across the loop iteration boundaries and complex control flows in order to fully exploit the technique.

Before we explain the main idea of the technique, we first describe our observation of hot cache-missing loads in some of the integer programs. Figure 1 sketches one of the hot loops in the 164.gzip benchmark of SPEC CINT2000. It is a nested loop where the most frequent execution paths are composed of those in the outer loop body, which constitute a complex control flow as shown in Figure 1. There is a hot load in the basic block (BB) at the loop header, where the cycle boundary for parallel execution in the Itanium machine is depicted by a horizontal line (==).

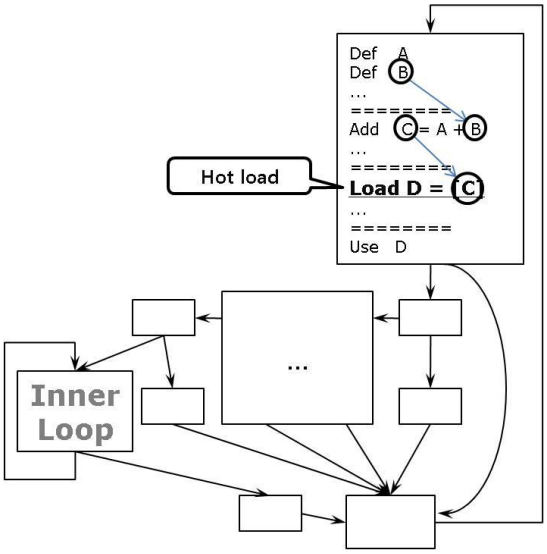

Figure 1. A hot cache-missing load in the 164.gzip benchmark

This load shows some typical characteristics of a hot load that can be found in integer loops. One characteristic is that there often is a tight data dependence chain from the loop entry to the source operands of the hot load. Each of those data dependent instructions constitutes one cycle in the scheduled loop, so moving the load in an earlier cycle requires moving the data dependent instructions as well, which is not easy since the chain often starts from the loop entry cycle.

Another characteristic is that there can be a non-trivial control flow composed of branches and joins within the loop such that earlier scheduling of the hot load requires code motions across it. In Figure 1, for example, even if we can move the load (and their data-dependent instructions) across the backedge, we might need to move it across the complex control flow at the bottom of the loop to the BBs near the loop entry for enough separation. Some path might have data dependences to the chain, thus

blocking the code motion, while others do not, thus allowing it, so it would be non-trivial to perform such a code motion efficiently with a minimal overhead of code duplication or recovery code for speculative loads (discussed shortly).

Finally, the hot loads are likely to be located close to the loop entry as in Figure 1, rather than in the middle of the loop. Since a load located at the entry is executed more often than the one located in some path of the loop body, it is likely to cause more cache misses than others. Moreover, the loads in the middle of the loop often access the data located close to the one accessed by the load at the loop entry, so cache misses, if any, are likely to occur at the entry rather than in the middle of the loop.

2. 2 Scheduling Hot Loads across Loop Iterations

Our proposed idea is scheduling the hot load (and its data-dependent instructions) across the loop backedge (and across the control flow). This would overcome the obstacle of the loop iteration boundary when we want to separate a hot load from its uses. Figure 2 illustrates moving a load across loop backedge to separate it from its use. In order to maintain the semantic correctness at the loop entry, a bookkeeping copy of the load is made at the incoming edge to the loop. In fact, this code motion achieves software pipelining since it allows the load of the next iteration (iteration n+1) to execute instructions of the current iteration (iteration n) while the load of the first iteration (iteration 1) is executed before entering the loop.

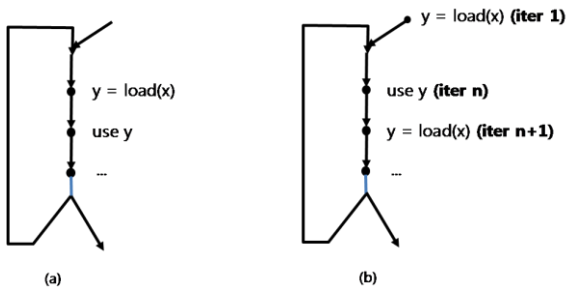

Figure 2. Scheduling a hot load across the loop backedge

There is a non-trivial issue in implementing this idea, though. As we saw in Figure 1, there can be a complex control flow in the loop body, so the load might need to move across the complex control flow. Global DAG scheduling techniques have been used for moving instructions across control flows, using speculative code motions across one target of a branch, join code motions above a join point, or code motions across both paths of a branch if data dependence allows, possibly combined with register renaming. So, if global scheduling can be applied across the loop backedge(s), we can move the hot the loads flexibly across backedges. One such a technique is a software pipelining technique called *enhanced pipeline scheduling* (EPS) [1, 2]. EPS achieves software pipelining via code motions across the loop backedge. So if we can employ EPS appropriately, we can use it for separating loads from their uses across iterations. In fact, such a separation is meaningful only when we can schedule

other instructions to fill the cycles between the load and the use, so our proposed cache optimization should be carried out in the context of scheduling, anyway. So we should adjust EPS for separation of loads and uses as well as for extracting ILP.

3. Overview of Enhanced Pipeline Scheduling

Previous section motivated the idea of separating hot loads and their uses across loop iterations and control flows, and proposed implementing the idea in the context of EPS, a software pipelining technique based on global code scheduling. We overview EPS in this section.

3.1 Code Motion-based Software Pipelining

EPS achieves software pipelining by moving operations across the backedge of the loop which has the effect of moving operations between loop iterations. This is quite different from the approach used by modulo scheduling (MS) [23] which places operation in a flat schedule with a pre-computed minimum initiation interval (MII). Due to its code motion pipelining, EPS can pipeline *any* type of loop including those with arbitrary control flows or outer loops. It should be noted that EPS is not explicitly guided by any MII as in MS. Rather, EPS blindly aims for a tight II by repeated greedy DAG scheduling (which will be described shortly).

Figure 3 shows the EPS process for an example loop, which performs an add, a load, and a branch. During the first stage of EPS in Figure 3 (a), the 1st cycle of the 1st iteration is scheduled only with x=x+4 as shown in Figure 3 (b). In the second stage, we schedule the second cycle of the 1st iteration. In order to generate increasingly compact groups as software pipelining proceeds, EPS is performed as follows: after a cycle is scheduled in a stage, all instructions scheduled in that cycle are available for code motion in the next stages. In fact, this is done by defining a DAG on the CFG of the loop, starting right after the previous cycle extending across the backedge and ending at the bottom of the previous cycle and by performing DAG scheduling to create a parallel group at the root of the DAG, which comprises the current cycle. Figure 3 depicts the DAG defined in each stage with unshaded edges. EPS repeats the DAG scheduling until the backedge is encountered. In Figure 3 (c), it is acceptable to move x = x+4 from the second iteration across the backedge to the second cycle of the 1st iteration as shown in Figure 3 (d). Moving the operation requires renaming since x is live at the exit, so we schedule it with a new target register x' after leaving a copy x=x' at the original location. In addition, since there is an edge that enters the loop from outside, a copy of the moved instruction is made at the entry so the instruction is still executed when the loop is entered.

In the third stage in Figure 3 (e), after scheduling cc=(y==0), the load and add instructions are moved across the backedge to the third cycle of the 1st iteration from the second and third iterations, respectively, as shown in Figure 3 (f). The target of the add is renamed this time to x'' and its source operand is *forward substituted* when it passes through the copy x=x'. Since the backedge is

encountered, we can stop the EPS process now. The three data instructions that were originally constrained by strict data dependences can now be executed in parallel.

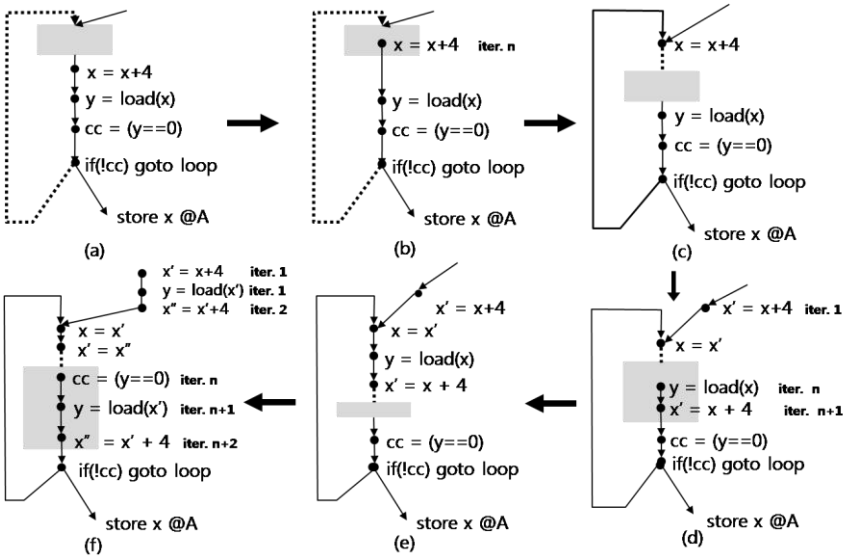

Figure 3. EPS Process for an example loop

3.2 Global DAG Scheduling

Each stage of EPS defines a DAG and an empty parallel group at the root of the DAG, which is then scheduled by a global DAG scheduling technique. Any global scheduling technique can be used, but we employ an aggressive one called *selective scheduling* [2]. It first computes available instructions that can move into the parallel group and remove inappropriate ones with a filtering heuristic. Then it chooses the best instruction and schedules it. Selective scheduling repeats this process until there are no more available instructions to move or the group is full.

Selective scheduling employs aggressive code motion techniques. *Renaming* is performed when needed in the context of code motion using copies. For example, when x=x+4 cannot be moved due to its target register x (e.g., x is live at the other target of a branch when we want to move the instruction speculatively above the branch), its target is renamed to x' and a copy instruction copy x=x' is left at the original place of the instruction as we saw in Figure 3. Although renaming leaves behind copies, they do not impose any true data dependences during subsequent code scheduling with the use of a technique called *forward substitution*. For example, if y=load(x) passes through x=x' during its code motion, it becomes y=load(x') (actually, both operations can also be grouped together for parallel execution after this transformation). It is known that renaming and forward substitution are successfully employed for software pipelining in EPS [2].

Speculative code motion can be performed across one target of a branch, and *join code motion* can be performed across a join point after making a bookkeeping copy. In Figure 4 (a), speculative code motion of y=w*w and join code motion of z=x+1 across the left path of the branch lead to a schedule in Figure 4 (b). If data dependences allow, however, it would be better to perform non-speculative

code motion across both targets of a branch, as u=u+1 in Figure 4 (c) (which we call *unification*), instead of speculative code motion or join code motion. Such a unified instruction is useful no matter which path is taken at execution time. Similarly, it would be desirable to perform "partially"-speculative code motions that speculate fewer branches since they have a better chance of being useful. So it is a good heuristic to schedule as many useful instructions as possible.

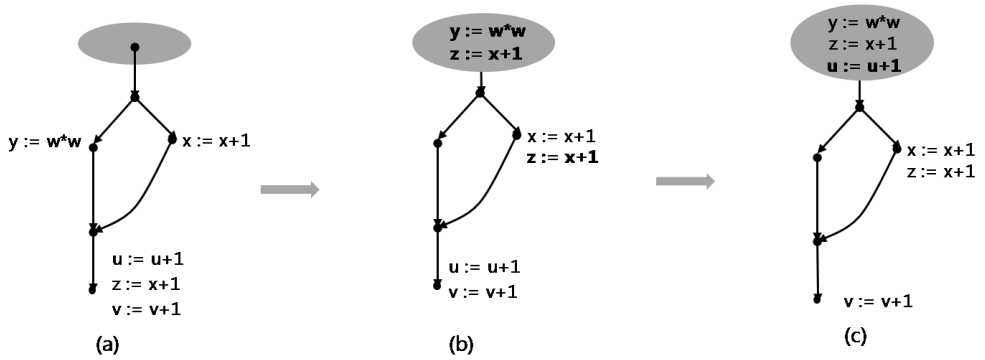

Figure 4. Speculative code motion, join code motion, unification

Selective scheduling estimates the degree of speculativeness for each available instruction and chooses the one whose degree is small in order to implement the usefulness heuristic. It then schedules in a way of minimizing speculative code motion or join code motion. In Figure 5 (a), for example, selective scheduling can move y=x+4 to the parallel group at the root of the DAG, while speculating only one branch and leaving only one bookkeeping copy, as shown in Figure 5 (b).

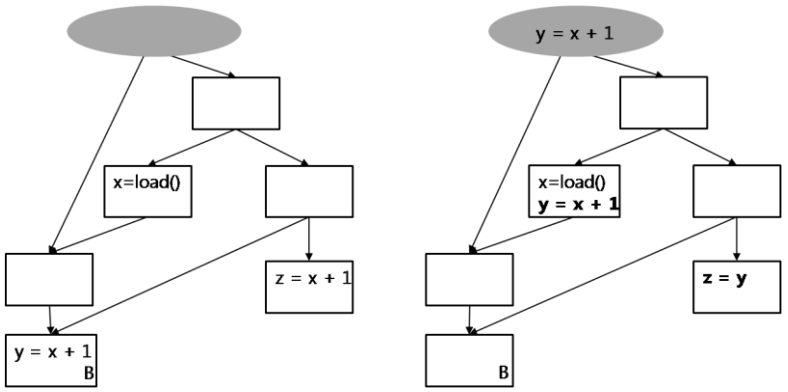

Figure 5. Performing partially-speculative code motion with a minimal booking copy

4. Adjusting Enhanced Pipeline Scheduling for Cache Optimization

Previous section overviewed the EPS with its aggressive DAG code motion techniques. This section describes how we adjusted EPS for cache optimization as well as for increasing ILP.

4.1 Increasing the Load Latency

The first update needed is increasing the latency of the hot loads, assuming that they will miss the cache. In Itanium, for example, the L1 cache hit latency is one cycle while the L1 cache miss latency is five cycles. So we set the latency of the hot loads to five cycles instead of one cycle, so that EPS schedules their uses five cycles later after the hot loads are scheduled. We hope that the newly added four slack cycles are scheduled by other *useful* instructions; otherwise we will still have a stall.

Increasing the load latency may affect the ILP, though. If the hot load is on the critical path, delaying the scheduling of its uses will increase the length of the critical path, lowering the ILP. For software pipelining of loops, we can compensate the loss of ILP by overlapping more iterations which can achieve an II similar to the one when the load latency is one cycle, unless there is a cyclic data dependence including the load and it constitutes the II of the loop. For the case of EPS, this means additional code motions across the loop backedges. In order to overlap the hot load enough across the backedges, the hot load and its dependent instructions should be scheduled aggressively, which can lead to more complex and worse code, in theory. So code motions should be controlled somehow.

On the other hand, the cache benefit caused by longer load latencies might be higher than its ILP penalty. That is, there is a better chance of scheduling useful instructions for the newly added cycles with EPS, while no useful work is possible if the CPU were stalled during those cycles. Therefore, it might be a useful strategy to pursue scheduling the increased cycles even if there is some ILP loss.

4.2 Speculative Loads

Speculative code motion of loads is important in increasing ILP since loads are often on the critical paths so their earlier execution affects the performance. There is one problem for the speculative load, though, which is when the speculative load causes an exception by accessing an illegal address or causing a page-fault. If the load came from a taken path, it is a legitimate exception. However, if the other path is taken at execution time, the exception should be ignored since it would not occur in the original, non-scheduled code.

For speculative exception handling, Itanium supports a speculative load instruction and a check instruction, so that compiler generates a speculative load instruction when a load moves above a branch speculatively, while leaving a check instruction in the original location of the load, as shown in Figure 6. The compiler also generates a recovery block, composed of the load instruction and any chain of instructions data-dependent on the load which also moved above the check instruction. If an exception condition is detected at the speculative load, it does not raise an exception right away. If the original path is not taken, then there is no problem. When the original path is taken and the check instruction is executed, a jump is made to the recovery block where the load is executed again, a genuine exception is raised, and possibly, the chain of data-dependent instructions are executed as well, which handles the exception precisely.

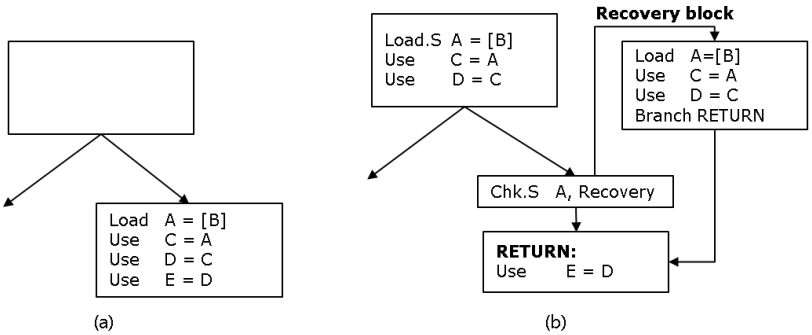

Figure 6. An example of speculative load

Although speculative loads allow earlier execution of loads, they also affect the ILP negatively. Since the check instruction works as a branch instruction, it affects code motion. Also, the recovery block is not scheduled, so if the recovery block is really executed, it is executed sequentially without any parallel execution. This is the reason why the original EPS on top of Open-64 generates speculative loads conservatively, such that only when the number of data dependent instructions on the load is small, the speculative load is generated.

For our separation of hot loads across backedges, however, we must generate a speculative load since the load must move above the loop exit branch. So, we allow speculation for the hot loads aggressively. Figure 7 shows the example of speculative loads for the 164.gzip loop we discussed in Section 2. This loop is the hottest loop in the benchmark, where the hot loads in the loop cause 80% of the data cache misses and 70% of the stalls for the benchmark. It cannot be handled by modulo scheduling due to control flows. Among the hot loads in the loop, the one at the loop entry BB is the hottest. This load is scheduled across the backedge of the loop. Since the loop exit branch is speculated during the code motion, a check instruction is left at the original location of the load, while the instruction is converted into a speculative load (we do not show the recover code for simplicity). Since the use of the load is delayed to be scheduled, it actually remain in the entry BB, so there is a control flow and other instructions between the load and the use, which avoids some stalls (if we allows more aggressive code motion of the load, it can move to the entry BB where the use is located, being unified across the control flow, except for the edge coming from the inner loop where a bookkeeping copy of the load is made. However, we do not allow such a code motion in the current heuristic due to the side effect, as will be discussed in Section 4.3).

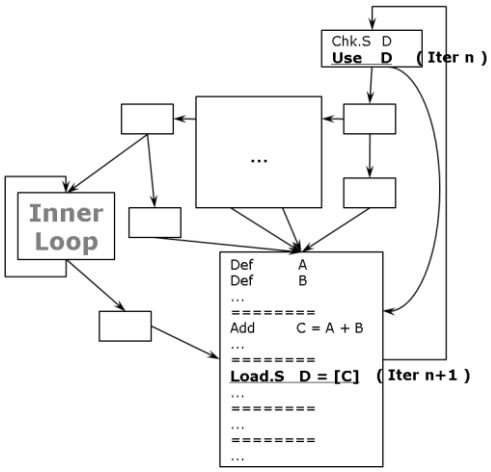

Figure 7. Speculative load in the 164.gzip example loop.

4.3 Adjusting Scheduling Heuristics

Although many of the global code motions in EPS are powerful and aggressive in extracting parallel instructions for a given cycle, they are involved with some side effect that can negatively affect the quality of other cycles. So if we perform such code motions excessively, it may reduce the overall quality of the generated code and the final ILP, unless there are abundant resources.

For example, renaming can contribute more to increasing the register pressure by lengthening the live range, in addition to generating additional copies, which would take resources unless being coalesced away. Moreover, speculative loads can be problematic as we discussed in Section 4.2.

Join code motion introduces bookkeeping copies at the incoming edges to the join point, which can affect the code quality of those edges negatively. Figure 8 illustrates the negative impact of a join code motion. When z=x+1 moves across the left edge of the join point to fill otherwise idle resource at the top BB, a bookkeeping copy of it should be added at the right edge of the join point. If there is no reduction of cycles for the original BB of z=x+1 (located at the bottom), this will increase one cycle for the right path. This can cause a serious penalty if there are many such code motions and if the right path is taken frequently at runtime. In fact, we could observe the problem in an 181.mcf loop where there is a complex control flow which requires many code motions to pass above multiple join points, generating excessive bookkeeping code.

Speculative code motion from cold paths to hot paths in a loop can also cause a problem by increasing the II of the loop. In 164.gzip, for example, excessive speculative code motions often place useless instructions to the hot paths, creating extra cycles and thus leading to poor performance.

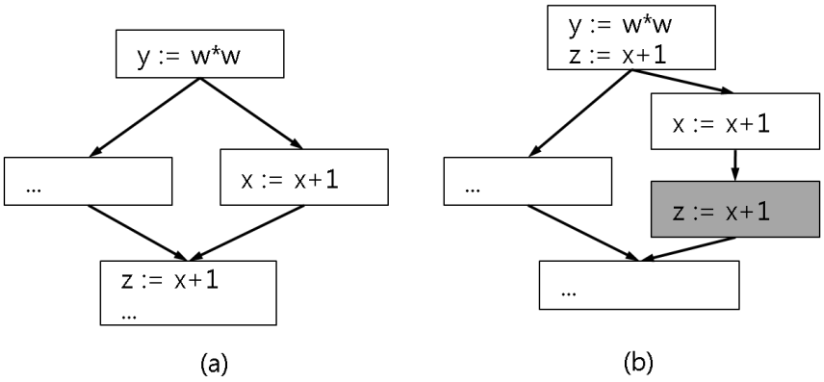

Figure 8. Negative impact of join code motion

In order to avoid excessive code motions, the original EPS uses a strict filtering heuristic when a parallel group is scheduled in each stage of EPS. That is, an available instruction is filtered out if the number of join points that it should pass to move from its original location to the group exceeds some predefined number. Similarly, an available instruction which speculates more than a predefined number of branches is filtered out.

From the point of separating a hot load and its uses, however, it would be desirable to loosen the filtering heuristic, so that the hot load and its data-dependent chain of definitions can move more aggressively, possibly across the loop backedges, and other instructions can also move aggressively to fill the added cycles between the load and its uses.

In order to compromise between the two points, we use the following heuristic. We classify instructions in a loop into two groups. The first group consists of the hot loads and their data-dependent chain of definitions, which we can hot-load-related instructions. The second group includes the rest of the instructions, which we call non-related instructions. We apply loose filtering heuristics for the first group so as to encourage aggressive code motions, while we apply the original, strict heuristic for the second group in order to avoid excessive code motions. For the example code fragment in Figure 9, the first group and the second group are depicted for a given hot load instruction. Only the first group will be scheduled aggressively.

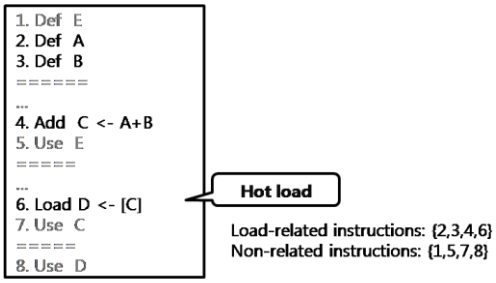

Figure 9. Load-related instructions and non-related instructions

5. Experimental Results

Previous section described the enhanced version of EPS for the cache optimization as well as the ILP optimization. In this section we report our preliminary evaluation results of the enhanced EPS to understand its impact on the cache performance and the overall performance.

5.1 Experimental Environment

We performed the experiment on a real machine with a product-level compiler. Our target machine is 900Mhz Intel Itanium2 processor and the compiler is the Open-64 version 3.0 [16]. We implemented EPS on top of the Open-64 compiler, and the EPS phase is located between the global instruction scheduling (GIS) phase and the register allocation phase of the Open-64 compiler.

Itanium2 has a 256KB L1 data cache. The latency of integer loads is one cycle for an L1 cache hit, 5 cycles for an L1 cache miss, and 11 or more cycles for an L2 cache miss. Itanium2 can issue up to 6 instructions in each cycle, where two loads and one floating-point instruction can execute in parallel. Our benchmarks include some of the integer programs in SPEC CINT2000 and CINT2006 [17].

We identified hot cache-missing loads via profiling using the *performance monitoring unit* (PMU) of Itanium2 [13]. The PMU can count and monitor the L1 data cache misses during the execution of an application program. It can also provide useful profiling information.

Using the PMU we counted the cache stall cycles generated by each memory instruction and identified those load instructions whose stall cycle overhead takes more than 1% of the running time. A load instruction not located in a loop is excluded, so is a load which is part of all cyclic data dependences in a loop since such a load is not movable and there is no opportunity for stall reduction. The final hot load instructions are marked so that EPS can identify them.

Table 1 shows the number of hot loads in each benchmark and the number of loops that include those hot loads, thus the target of our enhanced EPS (no other loops are touched by EPS).

Figure 10 depicts the ratio of the total stall cycles and the stall cycles of hot loads, compared to the total running time as 100% for each benchmark. The stall cycles of hot loads constitute 12% and 55% of the running time for 164.gzip and 181.mcf, respectively, while it takes 2~3% for other benchmarks.

Table 1. Number of hot loads and number of loops including them.

Benchmark	# of Hot loads	# of loops	**Benchmark**	# of Hot loads	# of loops
164. gzip	1	1	254. gap	1	1
175. vpr	2	1	256. bzip	1	1
176. gcc	2	1	300. twolf	2	1
181. mcf	6	3	445. gobmk	1	1
197. parser	2	1	456. hmmer	4	1

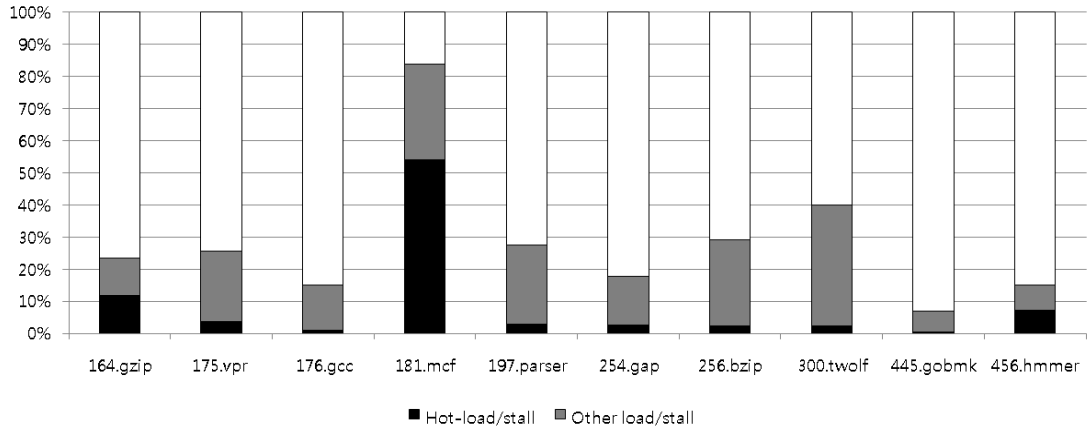

Figure 10. Ratio of the stall cycles to total execution time

5.2 EPS Configurations

We experimented with four configurations of EPS, compared to the base case where EPS is not turned on in Open-64, as follows:

① Base

Open-64 –O3 with EPS disabled. Only modulo scheduling and GIS [22] of Open-64 is turned on.

② EPS-no-cache: EPS without cache optimization

The original EPS is turned on for the target loops without any cache optimization. Actually, EPS is applied only to those loops that include hot loads in order to make a fair comparison with the cache-optimizing EPS. We also apply EPS to those loops previously handled by modulo scheduling if they include hot loads, after disabling modulo scheduling; if they are if-converted, we disable if-conversion as well since EPS does not currently handle if-converted loops appropriately. This can make ② perform worse than ①. Actually, there is no reason for EPS to perform worse than modulo scheduling for single-path loops, if EPS were guided by a pre-computed II as in modulo scheduling.

③ EPS-strict: EPS with increased load latencies and strict filtering heuristics

This is the same as ②, except that the load latency of hot loads is increased from one cycle to five cycles. We use the same scheduling heuristics as in the original EPS, so the same strict filtering heuristics are applied to load-related instructions as well as others. We can evaluate if the existing heuristics are enough to separate hot loads from their uses and to fill the added cycles.

④ EPS-selective: EPS with selective filtering heuristics

This is the same as ③, except that loose heuristics are applied to load-related instructions while strict heuristics are applied to others. This is the heuristics that we developed in Section 4.3 and it is expected to separate the loads from their uses enough, while filling the added slack cycles with a reduced side effect of aggressive code motions.

⑤ EPS-loose: EPS with loose filtering heuristics

This is the same as ③, except that loose heuristics are applied to both load-related instructions and others. This is expected to lead to excessive code motions, which can separate the loads from their uses well, but can affect the overall schedule negatively.

5.3 Results

Figure 11 shows the ratio of the total stall cycles for each configuration compared to Base as 100%. EPS-no-cache would affect the cache stalls little since it focuses only on ILP without any consideration of cache stalls. It may even increase the cache stalls as in 175.vpr and 300.twolf, by scheduling hot loads and its uses closer, by cross-iteration code motions or aggressive code motions across control flows that were not present in Base.

The remaining three configurations are with the increased load latency. As we move from ③ to ⑤, we loosen the filtering heuristics, which tends to reduce the stall cycles. This is so since more aggressive code motions that would otherwise be prohibited due to strict filtering heuristics allow better separation of loads and their uses and better filling of the slack cycles. Generally, EPS with cache optimizations could reduce the stalls of Base and EPS-no-cache tangibly in most benchmarks.

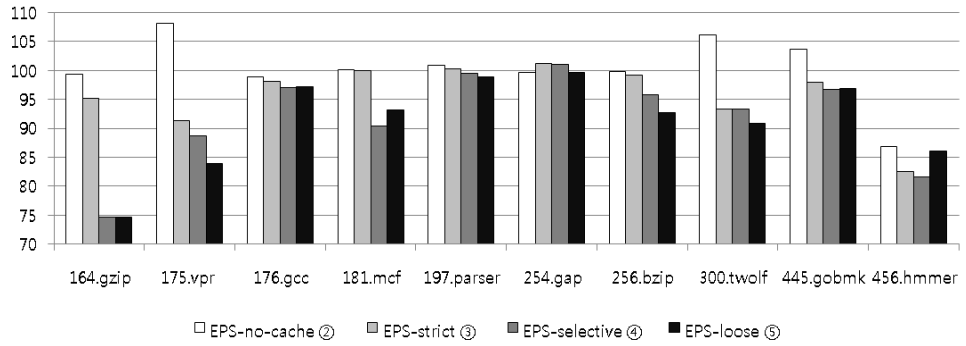

Figure 11. Ratio of total stall cycles (lower is better)

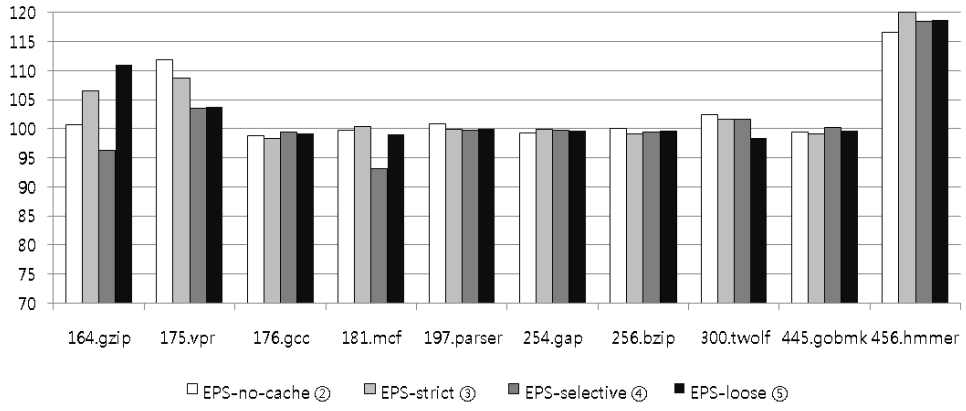

Figure 12. Ratio of total execution cycles (lower is better)

Figure 12 shows the ratio of the total execution cycles for each configuration compared to Base as 100%. EPS-no-cache which performs EPS only for those loops with hot loads does not show a better performance than Base or is even tangibly worse (i.e., 175.vpr). When we perform EPS with cache optimizations, there is a tangible performance improvement compared to Base in 164.gzip and 181.mcf, where the cache stall cycles take a significant portion of the running time (see Figure 10). The improvement is actually made only in EPS-selective, though, and both EPS-strict and EPS-loose show worse results. This indicates that loose filtering for load-related instructions and strict filtering for others is effective since it compromises between cache stall reduction and ILP.

For other benchmarks, there is little performance benefit over Base, which seems to be due to the tiny portion of stall cycles compared to the total execution cycles, hence little opportunity for performance improvement using cache optimization. Compared to EPS-no-cache, however, 175.vpr and 300.twolf show some tangible improvement, and their performance results are somewhat consistent with the stall cycle results. In fact, many benchmarks show a similar pattern between the stall cycles graph in Figure 11 and the execution cycles graph in Figure 12 for the four configurations, indicating that a correlation exists between cache stall reduction and performance improvement.

One thing to note from the graphs is that there are some cases where the cache stall cycles reduce tangibly when we move from strict filtering (③) to loose filtering (⑤), yet the execution cycles does not reduce or even increase tangibly. We found that some of these are due to the overhead caused by more aggressive code motions such as increased register pressure, more duplication at the join points, and negative effects of speculative loads, as we discussed in Section 4.1.

For EPS-selective, we counted the number of instruction groups (cycles) remaining between the hot load and its uses to check if EPS-selective can fill the slack cycles effectively so as to make enough distance between them to avoid stalls. Because there can be multiple paths from a hot load to its uses, we counted instruction groups for each path and calculated the weighted average using path profile, which is depicted in Table 2 as the average distance between the hot load and its uses.

Table2. Estimation of Average Distance (cycles) between hot load and its uses.

Benchmark	Distance	Benchmark	Distance
164. gzip	2.0	197. parser	2.4
175. vpr	2.0	254. gap	1.0
176. gcc	2.0	256. bzip2	3.0
181. mcf - loop1	10.9	300. twolf	1.4
181. mcf - loop2	4.4	445. gobmk	5.0
181. mcf - loop3	3.0	456. hmmer	1.5

Table 2 shows that EPS-selective could successfully fill the instruction groups for 181.mcf,

256.bzip2, and 445.gobmk since the distances are more than three, which would be the reason for the stall reduction depicted in the Figure 11. For the 164.gzip, the distance is relatively small (as we can see from Figure 7), but the loop iterates heavily, which leads to its high stall reduction. On the other hand, the distance is relatively big in 197.parser, yet the original ratio of stall cycles caused by the hot loads is trivial, so the stall reduction is also small. It should be noted that load-use distance would be one cycle if we apply EPS-no-cache, so they would lead to five-cycle stalls for every L1 cache miss.

We analyzed 164.gzip further to understand the performance impact of the balance between ILP and stall reduction. In order to see the impact of the scheduling heuristics on ILP, we calculate the average, weighted II for the hot paths in its hot loop for each heuristic, as shown in Table 3. Although there are several paths in the loop, hot paths of the loop take over 90% of overall execution time.

Table3. Estimations of II of hot paths in 164.gzip

Base	EPS-no-cache	EPS-strict	EPS-selective	EPS-loose
6	6	7	6	11

In case of EPS-strict, the hot-load is not pipelined (i.e., not scheduled across the backedge) as in Base because of its strict filtering heuristic. So, we cannot separate the load and its uses effectively, and the slack cycles are not hided across iterations; they are scheduled by some useless instructions from infrequent paths, which increases the II only without a big stall reduction.

EPS-selective and EPS-loose could pipeline the hot load, thus reducing stalls significantly while hiding the slack cycles across iterations. For EPS-loose, however, the II increases seriously, offsetting the benefit of stall reduction and lengthening the execution time. Since EPS-loose schedules instructions too aggressively, instructions from the infrequent paths are scheduled to the frequent paths. Actually, increasing the II by one cycle increases the execution time by 3.5% in 164.gzip, so the increased II affects performance. Considering the performance gain from stall reduction in 164.gzip is a maximum of 5%, scheduling the loop without increasing II is as important as making enough distance between the load and its uses.

6. Related Work

Exploiting non-blocking caches by increasing load latencies for compiler scheduling have been researched previously.

Abraham et al. found that a fraction of memory load instructions are responsible for the majority of cache misses from a profile of SPEC89 benchmarks [5]. So they set the latency of such loads to the cache miss latency while setting the latency of others to the cache hit latency, and perform instruction scheduling as we did. Unfortunately, they could not achieve any performance benefit because the added cycles cannot be filled by useful instructions. In our case, software pipelining and aggressive

scheduling across control flows allows more effective filling of those added latencies.

Kerns and Eggers introduced variable latencies for load instructions based on their load-level parallelism and data dependences, and proposed *balanced scheduling*, an updated list scheduling considering variable latencies [6]. Although balanced scheduling outperforms other list scheduling approaches such as greedy scheduling or lazy scheduling, it is a basic block scheduling.

Winkel researched a cache-miss-latency-aware modulo scheduling on Itanim2 with a production compiler [7]. Their modulo scheduler could achieve enough load-use distances without increasing II, thus improving cache stalls without affecting ILP. However, it targets only loops with a (if-converted) single basic block, thus not applicable to loops with control flows such as those hot loops in 164.gzip and 181.mcf. Sánchez and González performed a similar work on modulo scheduling [8].

Choi introduced a PMU-profile-based cache optimizations using prefetches and separation of load and its uses [13]. Although they did not describe the details of the experiments, the separation did not show any tangible performance improvement while showing good results with prefetches. We believe they did not consider aggressive code motion techniques for the separation, as in our paper.

7. Summary

We proposed a cache-miss-latency-aware EPS, which attempts to separate hot loads and their uses in integer loops using aggressive code motion techniques of EPS. It balances between cache stall reduction and ILP so as not to affect the II due to too-aggressive scheduling heuristics for separation. We could observe some promising, preliminary results for a few integer benchmarks which suffer from data cache misses seriously. And our analysis shows that the proposed technique works as intended for reducing the stalls without affecting the II. In fact, if the other benchmarks were also suffering from cache misses more seriously, we could observe more cases of performance improvement since their cache stalls reduce with the proposed EPS. We need to work on further to develop it as a more generally-applicable technique for integer loops.

Acknowledgement

This work was supported by a grant from the Gelato Federation. We thank Shin-Ming Liu and Teresa L. Johnson for their insightful comments on our earlier report.

References

[1] K. Ebcioğlu, A compilation technique for software pipelining of loops with conditional jumps. In Proceedings of workshop on Microprogramming, pages 69-79, 1987, Colorado Springs, Colorado, United States
[2] S.-M. Moon and K. Ebcioğlu. Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining. ACM Transactions on Programming Languages and Systems(TOPLAS), 19(6):853-898, Nov. 1997.
[3] David Kroft, Lookup-free instruction fetch/prefetch cache organization, Proceedings of the 8th annual symposium on Computer Architecture, p.81-87, May 1981, Minneapolis, Minnesota.

[4] David Callahan, Ken Kennedy, Allan Porterfield, Software prefetching, Proceedings of the fourth international conference on Architectural support for programming languages and operating systems(ASPLOS), p.40-52, Apr. 1991, Santa Clara, California, United States
[5] Abraham et. al, Predictability of load/store instruction latencies, Proceedings of the 26th annual international symposium on Microarchitecture, p.139-152, Dec. 1993, Austin, Texas, United States
[6] Daniel R. Kerns, Susan J. Eggers, Balanced scheduling: instruction scheduling when memory latency is uncertain, Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation(PLDI), p.278-289, June 1993, Albuquerque, New Mexico, United States
[7] Sebastian Winkel, Rakesh Krishnaiyer, Robyn Sampson, Latency-tolerant software pipelining in a production compiler, Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization, p104-113, Apr. 2008, Boston, United States
[8] F. Jesús Sánchez, Antonio González, Cache sensitive modulo scheduling, Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture, p.338-348, Dec. 1997, Research Triangle Park, North Carolina, United States
[9] Tien-Fu Chen , Jean-Loup Baer, Reducing memory latency via non-blocking and prefetching caches, Proceedings of the fifth international conference on Architectural support for programming languages and operating systems(ASPLOS), p.51-61, Oct. 1992, Boston, Massachusetts, United States
[10] Todd C. Mowry, Monica S. Lam, Anoop Gupta, Design and evaluation of a compiler algorithm for prefetching, Proceedings of the fifth international conference on Architectural support for programming languages and operating systems(ASPLOS), p.62-73, Oct. 1992, Boston, Massachusetts, United States
[11] C. K. Luk, T. C. Mowry "Compiler-based Prefetching for Recursive Data Structures", Proceedings of the seventh international conference on Architectural support for programming languages and operating systems(ASPLOS), p.222-233, Oct. 1996, Cambridge, Massachusetts, United States
[12] David Bernstein , Doron Cohen , Ari Freund, Compiler techniques for data prefetching on the PowerPC, Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques, p.19-26, June 27-29, 1995, Limassol, Cyprus.
[13] Y. Choi, A. Knies, G. Vedaraman, and J. Williamson. A Design and Experience: Using the Intel Itanium-2 Processor Performance Monitoring Unit to Implement Feedback Optimizations, In EPIC2 Workshop, Nov 2002.
[14] Jean-Francois Collard , Daniel Lavery, Optimizations to prevent cache penalties for the Intel® Itanium® 2 Processor, Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, Mar. 2003, San Francisco, California, United States
[15] Intel Itanium 2 Processor Reference Manual For Software Development and Optimization, May 2004
[16] Open-64 Home, http://www.open64.net/
[17] SPEC (Standard Performance Evaluation Corporation) http://www.spec.org/
[18] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, Dynamo: a transparent dynamic optimization system, Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation(PLDI), p.1-12, June 2000, Vancouver, British Columbia, Canada
[19] Trishul M. Chilimbi, Martin Hirzel, Dynamic hot data stream prefetching for general-purpose programs, Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation(PLDI), June 2002, Berlin, Germany
[20] Michael E. Wolf ,Monica S. Lam, A data locality optimizing algorithm, Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation (PLDI), p.30-44, June 1991, Toronto, Ontario, Canada
[21] David Callahan, Steve Carr, Ken Kennedy, Improving register allocation for subscripted variables, Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation(PLDI), p.53-65, June 1990, White Plains, New York, United States
[22] D. Berstein and M. Rodeh. Global Instruction Scheduling for Superscalar Machines. Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation (PLDI), p241-255, June 1991, Toronto, Ontario, Canada
[23] M. Lam, Software pipelining: an effective scheduling technique for VLIW machines, Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation(PLDI), p.318-328, June 1988, Atlanta, Georgia, United States