Load Instruction Characterization and Acceleration of the BioPerf Programs

Paruj Ratanaworabhan and Martin Burtscher
Computer Systems Laboratory, Cornell University
{paruj, burtscher}@csl.cornell.edu

Abstract

The load instructions of some of the bioinformatics applications in the BioPerf suite possess interesting characteristics: only a few static loads cover almost the entire dynamic load execution and they almost always hit in the data cache. Nevertheless, these load instructions represent a major performance bottleneck. They often precede or follow branches that are hard to predict, which makes their L1 hit latency difficult to hide even in dynamically scheduled execution cores. This paper investigates this behavior and suggests simple source-code transformations to improve the performance of these benchmark programs by up to 92%.

1. Introduction

The bioinformatics field is primarily concerned with the development of advanced information and computation technology to tackle problems in biology. With the exponential growth of biological databases in recent years, bioinformatics applications are fast becoming an important computer workload. In response to this trend, the computer architecture community has released several representative benchmark suites, including BioInfoMark [13] from the University of Florida released in January 2005, BioBench [1] from the University of Maryland released in March 2005, and BioPerf [3] from Georgia Tech released in October 2005.

This paper investigates the bioinformatics applications from the BioPerf suite. In particular, we focus on characteristics of their load instructions. Loads represent a large percentage of the executed instructions in these applications and exhibit interesting properties. On the one hand, only a small fraction of the static loads are responsible for almost all executed loads, making these few loads an attractive target for performance optimizations. Even code size increasing optimizations can be applied without much concern over instruction cache performance degradation. On the other hand, these loads incur only few L1 data cache misses, which seems to leave no opportunity for optimization. Nevertheless, we show that the multi-cycle L1 hit latency can be problematic and poses a serious performance bottleneck in these bioinformatics applications. The L1 hit latency of modern superscalar microprocessors is typically greater than a single cycle. For example, for integer loads, the Power PC G5 [10] and the Alpha 21264 [12] have a three-cycle and the Pentium 4 [11] has a two-cycle L1 load-to-use latency.

It is generally true that most of the time this short latency can be fully hidden. On the software side, optimizing compilers perform local and possibly global code scheduling, i.e., they try to move independent instructions between loads and their first uses to mask the load latency. On the hardware side, today's high-end microprocessors contain large issue queues, ROBs, and load/store queues. Together with their dynamic schedulers, finding useful instructions to execute during the two or three cycles associated with the L1 hit latency is often not a problem. However, this paper shows that the code found in some bioinformatics applications can break the latency hiding mechanisms of current compilers and out-of-order execution cores and suggests some corrective measures.

The rest of this paper is organized as follows. The next section studies the characteristics of the load instructions in a number of bioinformatics codes and explains why their latency is hard to hide. Section 3 illustrates the source-code load scheduling necessary to avoid this problem. Sections 4 and 5 discuss the speedups due to hiding the L1 hit latency. Section 6 describes related work. Section 7 concludes the paper.

2. Load Instruction Characteristics

For this study, we selected nine applications from the BioPerf suite covering three key areas in bioinformatics, namely sequence analysis, molecular phylogeny analysis, and protein structure analysis. To profile and analyze the nine programs, we used the ATOM toolkit [17] and the inputs described in Table 2 of the BioPerf paper [3], i.e., the medium-sized Class-B input sets. The applications were compiled with the Alpha-DEC C compiler version 6.5 and the "-O3 -arch ev68" optimization flags for the profile runs.

Figure 1 shows the percentage of the executed instructions that are loads. Results for stores, conditional branches, and other instructions are also provided. On average, loads account for 30% of the executed instructions. The absolute numbers of executed loads range from 20 billions to 270 billions.

Table 1 shows the total number of instructions executed as well as the fraction thereof that is floating-point instructions. Only *hmmpfam*, *predator*, and *promlk* execute a significant number of floating-point operations. In these three applications, 1.7%, 6.5%, and 30.9%, respectively, of the executed instructions are floating-point loads. This indicates that, with the exception of *promlk*, integer loads are more significant than floating-point loads in our benchmark applications. Figure 2 plots the fraction of executed loads that stems from the most frequently executed load instructions. For clarity, only three representative bioinformatics programs are shown; the other six applications exhibit similar characteristics. For comparison purposes, three programs from the SPEC CPU2000 integer benchmark suite are also included. The figure shows that about 80 static loads cover over 90% of the executed loads in the three bioinformatics applications. For the SPEC CPU2000 programs *crafty*, *vortex*, and *gcc*, the same number of static loads provides a much lower coverage (about 10% to 58%).

1-4244-0509-2/06/$20.00 ©2006 IEEE

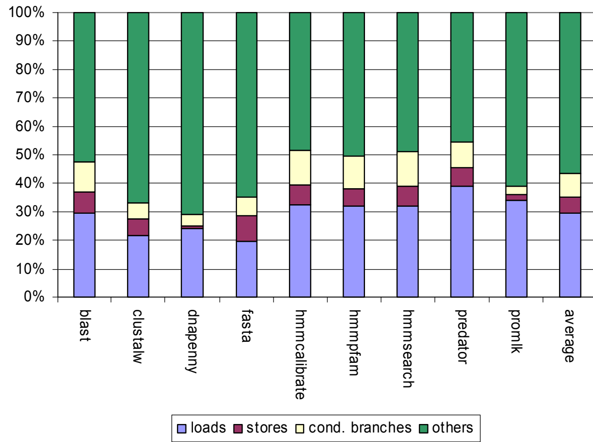


Figure 1: Instruction profile of the bioinformatics applications in the BioPerf suite

Program	Instructions (B)	Floating-Point
blast	77.3	0.04%
clustalw	789.4	0.04%
dnapenny	145.4	0.04%
fasta	542.1	0.63%
hmmcalibrate	67.9	0.15%
hmmpfam	277.4	5.07%
hmmsearch	894.2	0.02%
predator	837.6	13.85%
promlk	339.7	65.33%

Table 1: Number of executed instructions and percentage of executed floating-point instructions

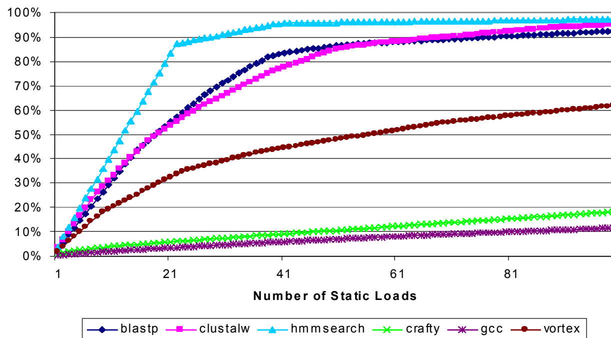


Figure 2: Cumulative frequency of executed loads versus number of static loads for representative programs from the BioPerf and the SPEC CPU2000 integer benchmark suites

Since loads represent a large fraction of the executed instructions in our nine applications and only a small number of static loads are responsible for the vast majority of the executed loads, these few loads contribute substantially to the total runtime of the applications.

2.1 Cache Performance

Table 2 shows the cache performance of our benchmark programs. The cache subsystem is modeled and simulated using ATOM. The parameters are given in Table 3 and reflect the cache configuration of our Alpha 21264 reference machine.

We find that the caches satisfy almost all load accesses. On average, merely 0.03% of the executed load instructions access the main memory. This number is likely to be associated solely with compulsory misses. 99.1% of the loads hit in the L1 cache. *Blast* has the highest overall miss rate. Given our system's L1, L2, and main memory latencies of 3, 5, and 72 cycles, the average memory access time (AMAT) for blast is 3 + 1.78% * (5 + 4.05% * 72) = 3.14 cycles. Clearly, the dominating term is the L1 hit latency. The L2 and main memory latencies only contribute 0.14 / 3.14 = 4.5% to the AMAT.

	Local miss rate for loads			AMAT
	L1	L2	Overall	
blast	1.78%	4.05%	0.072%	3.14
clustalw	1.90%	0.00%	0.000%	3.10
dnapenny	0.46%	4.30%	0.020%	3.04
fasta	0.47%	0.05%	0.000%	3.02
hmmcalibrate	1.61%	4.24%	0.068%	3.13
hmmpfam	0.67%	10.64%	0.071%	3.08
hmmsearch	0.35%	7.69%	0.027%	3.04
predator	0.46%	0.15%	0.001%	3.02
promlk	0.52%	4.93%	0.026%	3.04
average	**0.91%**	**4.01%**	**0.03%**	**3.07**
gmean	**0.74%**	**0.75%**	**0.01%**	**3.07**

Table 2: Cache performance of each bioinformatics application

L1 data cache	size: 64 KB associativity: 2 ways block size: 64 bytes separate from instruction cache write policy: write back, write allocate
L2 cache	size: 4 MB associativity: direct-mapped block size: 64 bytes holds instructions and data

Table 3: Modeled parameters of the cache subsystem

Note that the L1 data cache miss rate is very low even though the input data that the applications operate on are too large to fit into the cache. The reason for the low miss rates is that these programs tend to operate on a chunk of data that fits into the L1 cache for a period of time before moving on to the next chunk. Hence, we only see a few compulsory cache misses that occur when the programs transition to the next chunk.

It should be noted that there are bioinformatics applications that are memory-bound and do not always hit in the L1 data cache. They are, however, not the focus of this paper. Examples of such applications include *diffseq*, *megamerger*, and *shuffleseq* from the European Molecular Biology Open Software Suite [16], which are part of the BioInfoMark suite.

2.2 Load to Branch and Branch to Load Sequences

2.2.1 Out-of-order Execution

Consider the code snippet in Figure 3, which is taken from *hmmsearch*. The snippet is part of the most frequently exe-

cuted code section. The arrows show the load dependency chains. Although these chains are relatively short, they turn out to be problematic. The ultimate consumers are instructions that lead to control-flow decisions, which can make the corresponding conditional branches hard to predict. The most critical point is in the first basic block (BB1). Until the conditional branch in this block is resolved, the instructions beyond it can only be executed speculatively and will have to be squashed if a branch misprediction occurs.

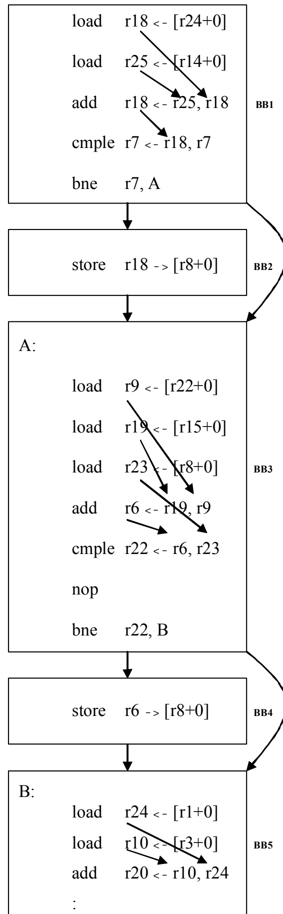


Figure 3: Data and control dependences in a section of *hmmsearch* machine code

To simplify the discussion, suppose that all loads hit in the L1 data cache and that all load operands are ready. Furthermore, assume that the L1 hit latency is three cycles, that the branch prediction logic incorrectly predicts the execution path to be BB1, BB3, and BB5, and that the issue queue can dispatch two instructions per cycle, matching the service rate of the L1 cache sub-system.

In cycle x, the first two loads from BB1 are issued. The dynamic scheduler then detects that the first two loads in BB3 and in BB5 are independent instructions. Thus, the scheduler issues these loads in cycles x+1 and x+2, respectively, to hide the latency of the loads issued in cycle x. These are speculative issues since the branch in BB1 has not yet been resolved. In cycle x+3, the load results from the first two loads in BB1 can be consumed by the add instruction from BB1. The scheduler then dispatches this non-speculative add together with the speculative third load in BB3. In cycle x+4, the compare instruction from BB1 can be issued, and the branch outcome is finally determined in cycle x+5 when the result of the compare is available. Figures 4(a) and 4(b) show the instructions in the pipeline at cycles x+4 and x+5.

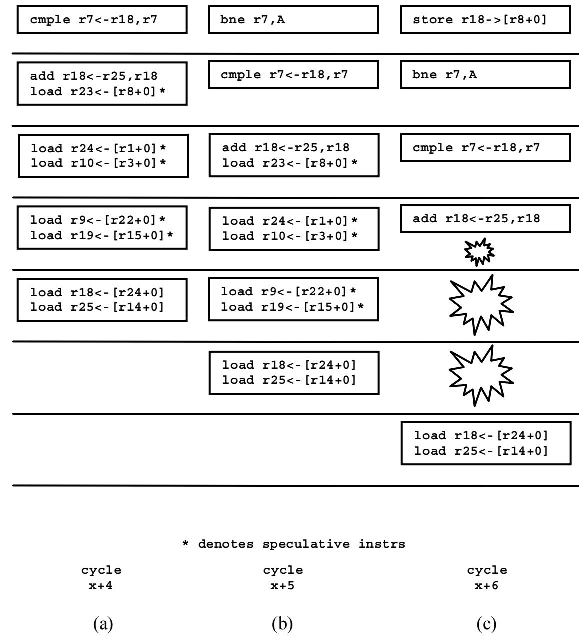


Figure 4: *hmmsearch* instructions in cycle x+4 (a), x+5 (b), and x+6 when branch misprediction occurs (c)

Now, consider what happens in cycle x+6 when the branch misprediction is detected (Figure 4(c)). The speculatively issued instructions are squashed and the fetch path is corrected to BB2, BB3, and BB5. At this point, the L1 hit latency of the first two loads in BB1, which belong to a load to branch sequence, is effectively added to the branch misprediction penalty because their latency delayed the resolution of the branch and no useful instructions were executed during this delay. In addition, during the next few cycles, the CPU fetches the loads in BB3, which come after the mispredicted branch, along with their dependent instructions. Because the processor pipeline is empty after recovering from the branch misprediction, there are no independent instructions to hide the latency of the loads in BB3 at this point. In other words, their L1 hit latency is fully exposed. The loads in BB5 suffer from the same problem if the conditional branch in BB3 is mispredicted.

Code sections involving a load to branch or a branch to load sequence that is followed by load-dependent instructions occur frequently in these bioinformatics applications and often involve hard-to-predict branches. The L1 hit latency of these load instructions, thus, becomes problematic. It either increases the branch misprediction penalty or is exposed after a

branch misprediction. This can significantly increase the overall runtime, as we will show.

Table 4(a) shows how often load to branch sequences occur as a percentage of the total executed loads in the nine bioinformatics applications. It also gives the average branch misprediction rate for the branches in such sequences, which are very high. Table 4(b) indicates the percentage of the executed loads that have tight dependence chains and appear right after a branch that has a misprediction rate of 5% or higher. Note that the two load sequences are not mutually exclusive. A load dependent chain that comes after a hard-to-predict branch can also lead up to a branch instruction. We use a hybrid branch predictor [15] with an entry for each static branch (i.e., there is no aliasing) to measure the branch misprediction rate.

	Load to branch	Average branch misprediction rate
blast	75.7%	19.9%
clustalw	56.2%	5.9%
dnapenny	33.6%	12.1%
fasta	31.6%	17.2%
hmmcalibrate	91.6%	11.2%
hmmpfam	92.4%	10.4%
hmmsearch	93.5%	9.9%
predator	51.1%	10.5%
promlk	15.2%	6.3%

(a)

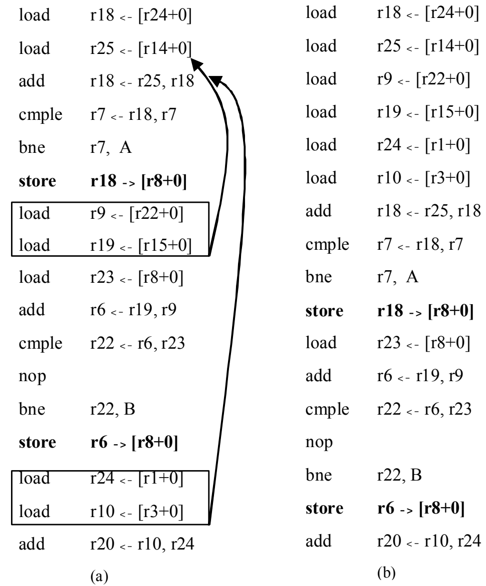
	Load dependent chain after hard-to-predict branch
blast	32.7%
clustalw	19.6%
dnapenny	6.7%
fasta	23.2%
hmmcalibrate	56.5%
hmmpfam	57.8%
hmmsearch	60.4%
predator	21.1%
promlk	2.3%

(b)

Table 4: Load to branch sequences as percent of total executed loads and average misprediction rate for the following branches (a), loads after hard-to-predict branches (greater than 5% misprediction rate) as percent of total executed loads (b)

2.2.2 Load Hoisting with an Optimizing Compiler

The instruction sequence in Figure 3 can be optimized. As BB1 dominates BB3 and BB5, the first two loads in BB3 and BB5 will always be executed if BB1 is executed. Because these two load pairs comprise independent instructions, they can be hoisted into BB1 (Figure 5(a)). This hoisting allows the out-of-order engine to schedule the loads from BB3 and BB5 non-speculatively to hide the latency of the original loads in BB1 as well as the latency of the hoisted loads. Thus, performing this code transformation will reveal how much the original code is slowed down due to the L1 hit latency. Note that the transformation does not change the predictability of the branch. It only changes the type of the instructions that precede and follow the branch. They are now single-cycle and/or independent instructions whose execution does not cause pipeline stalls, even after a branch misprediction recovery action.




```
if ((sc = ip[k-1]  + tpim[k-1]) > mc[k])  mc[k] = sc;
if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k])  mc[k] = sc;
if ((sc = xmb  + bp[k]) > mc[k])  mc[k] = sc;
```

(c)

Figure 5: Load hoisting: original code (a), after load hoisting (b), corresponding source code (c)

Unfortunately, hoisting these loads is beyond the capabilities of most optimizing compilers (see below). The culprits are the intervening store instructions in BB2 and BB4 (highlighted in bold in Figures 5(a) and 5(b)). Hosting any load across a store requires precise static memory disambiguation information, which compilers often cannot derive. However, if we look at the source code that corresponds to this sequence of machine instructions (Figure 5(c)) and the type of the variables, it quickly becomes obvious that this load hoisting is safe. The hoisted loads correspond to `xmb` and the `dpp`, `tpdm`, and `bp` arrays that can never alias with a store to the `mc` array. The only load that cannot be hoisted is the third load in BB3 as it reads from the `mc` array.

3. Source-Code Load Scheduling

Having identified the type of instruction sequence whose L1 hit latency is difficult to hide, we performed some load scheduling at the source-code level to remove the problem. Because of the availability of semantic and context information, manually scheduling loads at this level allows us to hide the L1 hit

latency in cases where an optimizing compiler and an out-of-order engine fail to do so. In addition, this optimization is easy to do in these applications because they only contain a few performance critical static loads (see previous section).

To identify which loads to optimize for a given application, we use ATOM to detect the two load sequences described in Section 2.2, and map the loads back to source code lines. A profile run then determines, for each sequence, the frequency of execution, the branch misprediction rate, the L1 miss rate, and information about the corresponding lines of source code. The optimization candidates are the frequently executed loads that lead to or follow branches with high misprediction rates. It should be noted, however, that we cannot always find opportunities to schedule loads in the source code. Although candidate loads may exist at the machine instruction level, there may not be enough opportunity in the source code to schedule the loads (e.g., in a tight loop). Of the nine bioinformatics applications we have studied, six are amenable to such scheduling (Section 3.3) and we will restrict our evaluation to these six applications. Although the primary purpose for this scheduling is to hide the L1 hit latency, our transformations sometimes introduce additional opportunities for optimization. The next two subsections demonstrate the load scheduling on the programs *hmmsearch* and *predator*.

load index	5175	5177	5179	5182
frequency	3.97%	3.97%	3.97%	3.97%
L1 miss rate	0.05%	0.02%	0.07%	0.03%
branch misprediction	11.20%	28.41%	38.24%	0.50%
in function	P7Viterbi	P7Viterbi	P7Viterbi	P7Viterbi
line number	132	133	134	136
in file	fast_algorithms.c	fast_algorithms.c	fast_algorithms.c	fast_algorithms.c

Table 5: Profile of the frequently executed loads in *hmmsearch*

3.1 Hmmsearch

A sample profile of the most frequently executed loads in *hmmsearch* is given in Table 5. The profile points us to a section of code in the P7Viterbi function, which is replicated in Figure 6(a). This loop contains several short IF statements whose THEN paths contain a single store statement. The IF conditions are rather involved, requiring loads from at least two arrays. Each statement in boxes 1, 2, and 3 contains a chained dependence on `mc[k]`, `dc[k]`, and `ic[k]`, respectively. The code in the three boxes is otherwise independent. When compiling the statements in this loop into machine code, it will contain tight dependence chains involving loads to control transfer instructions that are followed by other loads as shown in Figure 3. The profile in Table 5 points to the loads in the first four IF conditions in box1. Each load rarely misses in the L1 cache, which is expected for regular incremental array accesses. Except for the last IF statement, which is a bounds check, these IF statements have a high branch misprediction rate. Therefore, we can expect the L1 hit latency of the corresponding loads to degrade the performance of this loop as described in Section 2.2.

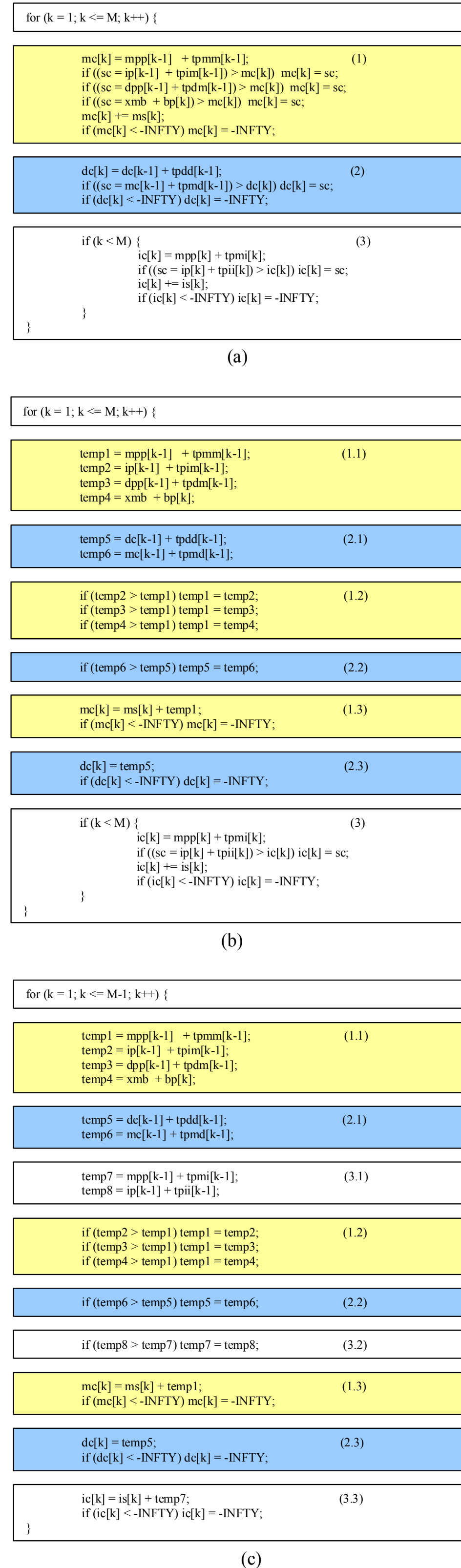


Figure 6: A loop in *hmmsearch*: original code (a), code after load scheduling at source level (b) and (c)

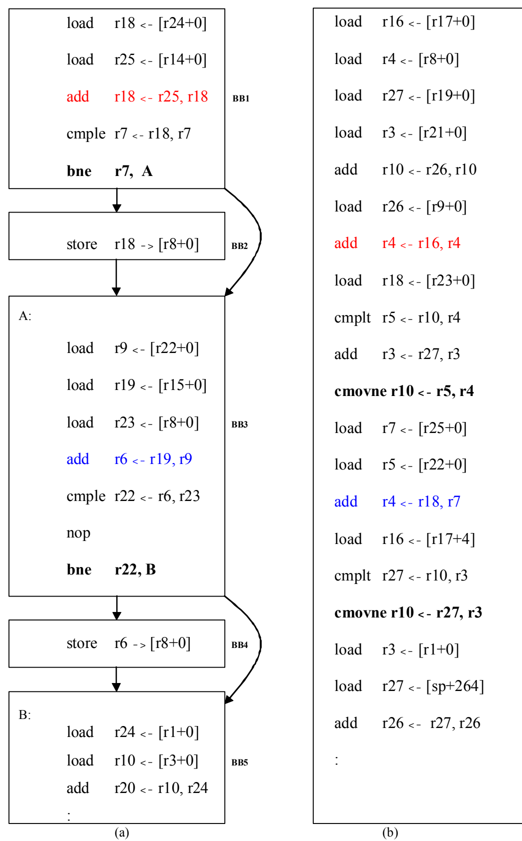


Figure 7: Generated machine code: original code (a) and after load scheduling in the source code (b)

We can hide these load latencies by manually scheduling the source code. For box 1, we observe that loading an element from arrays `mpp`, `tpmm`, `ip`, `tpim`, `dpp`, `tpdm`, and `bp` in each iteration can be done independently. The same is true for the elements of arrays `dc`, `tpdd`, `mc`, and `tpmd` in box 2. Therefore, we create six temporary variables to hold the results of the computations that need to be performed in the condition checks of the IF statements in boxes 1 and 2. Then, we substitute the variables for the original computations. After that, we schedule the code as shown in Figure 6(b) so that the bodies of each box are used to hide the latency of each other. Notice how the new code remedies the tight load dependence chains from the original code and enables the compiler to convert the original control flow into much faster conditional moves in boxes 1.2 and 2.2. This kind of load scheduling is hard to do for a compiler because of the store instruction in the THEN path of each IF statement. Static compiler analysis may not be able to perform the necessary memory disambiguation, thus impeding the scheduling of loads.

This loop can be further optimized. We can break the guarding IF condition in box 3 and schedule the code more, creating two additional temporary variables and using the body of box 3 for further latency hiding (Figure 6(c)). Again, the compiler will be unable to move any loads out of the guarding conditions if it cannot guarantee safety. Based on the source code, we know that box 3 will be executed as many times as boxes 1 and 2 except for the last iteration. We can, therefore, shorten the loop count by one, duplicate the load-transformed code of boxes 1 and 2, and place the duplicate right after the loop exit.

Figure 7 compares the machine instructions of the original code with the manually transformed code. The original store instructions that appear in Figure 7(a) are not visible in Figure 7(b) as they are pushed down by the hoisted load instructions. Now, the dynamic scheduler has enough non-speculative instructions available to hide the latency of the loads. In addition, notice that the original branch instructions are transformed into faster conditional move operations (highlighted in bold in Figure 7). So, for *hmmsearch*, our source-code transformation allows the compiler to optimize the original code in two ways: it can always hide the L1 hit latency of the loads and it can eliminate the conditional branches.

```
c = k * m;                                                  - (1)
for (tt = 1, z = row[i]; z != PAIRNULL; z = z->NEXT)       - (2)
    if (z->COL == j)                                        - (3)
      { tt = 0; break; }                                    - (4)
if (tt != 0)                                                - (5)
    c = va[j];                                              - (6)
if (c <= 0)                                                 - (7)
    { c = 0; ci = i; cj = j; }                              - (8)
else                                                        - (9)
    { ci = pi; cj = pj; }                                   - (10)
                              (a)
```

```
temp1 = k * m;                                              - (1)
c = va[j];                                                  - (2)
for (tt = 1, z = row[i]; z != PAIRNULL; z = z->NEXT)       - (3)
    if (z->COL == j)                                        - (4)
      { tt = 0; break; }                                    - (5)
if (tt == 0)                                                - (6)
    c = temp1;                                              - (7)
if (c <= 0)                                                 - (8)
    { c = 0; ci = i; cj = j; }                              - (9)
else                                                        - (10)
    { ci = pi; cj = pj; }                                   - (11)
                              (b)
```

Figure 8: Latency hiding in *predator*: original code (a) and transformed code (b)

3.2 Predator

Scheduling the loads at the source-code level does not have to be as involved as described above. Figure 8 shows the original and the transformed code of a section of the BioPerf program *predator* (from prdfali.c). In the original code (Figure 8(a)), suppose the load `va[j]` in line 6 hits in the L1 cache. Because of line 5, we have a situation where the load of `va[j]` into variable `c` is immediately preceded by a control-flow decision. If line 6 is frequently executed and the branch in line 5 is hard to predict, we can anticipate a situation where the L1 hit latency will be exposed after a branch misprediction and hamper performance.

The transformed code in Figure 8(b) improves this situation. It hoists the load of `va[j]` before the FOR loop (line 2), allowing the load to execute non-speculatively. It uses the body of the FOR loop to hide the load latency. Lines 1, 6 and 7 in

8(b) represent corrective measures to restore the value of `c` to `k*m` in case `va[j]` was not supposed to be loaded.

Note that an optimizing compiler is unlikely to hoist the load in the way we propose. Even with feedback information telling the compiler that line 6 in the original code is indeed frequently executed, it would be hard to prove that variable `j` always contains a valid index for array `va` at line 1. If the compiler cannot prove that it is safe to move the load in question out of its guarding IF condition, it will not perform this optimization. However, at the source code level, we know that `j` is a local variable whose value is always within `va`'s array bounds (code not shown), thus making it safe to hoist the load.

3.3 Other Applications

Besides *hmmsearch* and *predator*, we discovered similar opportunities to optimize the load scheduling in the source code of four other applications, namely *dnapenny*, *hmmpfam*, *hmmcalibrate*, and *clustalw*. Table 6 shows, for the six applications, the number of load candidates we considered and the approximate number of lines of source code involved in the load transformations. For each application, we use the fastest, most optimized version as the baseline code on which we perform the load scheduling.

	Static loads considered	Lines of C code involved
dnapenny	3	10
hmmpfam	16	25
hmmsearch	19	30
hmmcalibrate	14	25
predator	1	5
clustalw	4	10

Table 6: Number of static loads and lines of C source code involved in load transformation

4. Evaluation Methodology

4.1 Evaluation Platforms

We use four evaluation platforms: Alpha 21264 - Tru64 UNIX 5.1B, PowerPC G5 - Mac OS X, Pentium 4 - Red Hat Linux, and Itanium 2 - Red Hat Enterprise Linux AS4. Details are presented in Table 7.

	Alpha 21264 - Tru64 Unix	Power PC - Mac OS X
Datapath	64 bits	64 bits
Clock speed	833 MHz	2.7 GHz
Register	32 GPR, 32 FPR	32 GPR, 32 FPR
L1 data cache size	64KB 2-way set associatve	32 KB 2-way set associative
L1 data cache hit latency	3 to 4 cycles (Int/FP)	3 to 5 cycles (Int/FP)
L2 cache size	4MB unified direct-mapped	512KB unified 8-way associativity
L2 hit latency	8 cycles	11 to 12 cycles
OS version	OSF V5.1	Mac OS X 10.3.9
	Pentium 4 - Linux	**Itanium 2 - Linux**
Datapath	32 bits	64 bits
Clock speed	2.0 GHz	1.6 GHz
Register	8 GPR, 8 FPR	128 GPR, 128 FPR
L1 data cache size	8 KB 4-way associative	16 KB 4-way associative
L1 data cache hit latency	2 to 6 cycles (Int/FP)	1 cycle (Int)
L2 cache size	256 KB 8-way associativity	256 KB 8-way associativity
L2 hit latency	7 to 10 cycles	5 to 7 cycles
OS version	Red Hat Linux 9.0	Red Hat Enterprise Linux AS4

Table 7: Information about the evaluation platforms

4.2 Baseline Optimizations

On the Alpha-Tru64 Unix platform, we compiled our benchmark programs with the DEC-Alpha C compiler V6.5-003 with the "-O3" optimization flag except for *clustalw*, which performs better with "-O4". "-O3" instructs the compiler to perform global common subexpression elimination, global code motion, strength reduction, test replacement, split lifetime analysis, global code scheduling, global inlining, loop unrolling, branch elimination, etc. The "-O4" flag adds software pipelining, loop vectorization, and more aggressive insertion of NOP instructions to improve scheduling.

On the PowerPC-Mac OS X and the Intel-Linux platform, we use the GNU C compiler (gcc) version 3.3.3 with the "-O3" optimization flag to compile the applications. It performs optimizations such as speculative motion of load instructions, interblock scheduling, basic block reordering, strength reduction, global common subexpression elimination, inlining, alignment of loops, functions, jumps, and labels.

On the Itanium-Linux platform, we use the Intel C compiler version 9.0 with the "-O3" optimization flag to compile the applications. The optimizations performed are similar to those of gcc.

In addition to the optimizations listed above, we also use feedback-directed optimization. On all four platforms, the smallest input sets are used to generate the feedback information. With feedback-directed optimization, we are able to improve the average runtime of our baseline codes by 8%, 4%, 2%, and 2% on the Alpha, Power PC, Pentium, and Itanium platforms, respectively.

5. Evaluation Results

5.1 Performance with Large Input Sets

This section evaluates the load-transformed version of the applications when run with large input sets, which correspond to the class-C input instances and databases in the BioPerf suite. For *hmmsearch*, BioPerf only provides small and medium input sets, so we used the large inputs for this program from the BioInfoMark suite. All load-transformed codes are subject to the same optimizations (including feedback-directed optimization) as the baseline codes (see previous section). All timing measurements refer to the sum of the user and the system time as reported by the UNIX shell's *time* command.

Table 8 lists the absolute runtime in seconds for the original and the load-transformed code of the six bioinformatics applications on our four evaluation platforms. We could not get *dnapenny* to compile on the Itanium platform. We note that the absolute runtimes range from minutes (*hmmcalibrate*) to over an hour (*clustalw*). The corresponding load-transformed code speedup, including the harmonic mean, is shown in Figure 9.

We find that our source-code modifications provide greater performance benefits on the Alpha and the Power PC than on the Pentium 4. There are two reasons for this behavior. First, the L1 data cache hit latency for integer loads is greater in the Alpha and Power PC (3 cycles) than in the Pentium 4 (2 cycles). Second, our manual scheduling introduces additional

variables, which increase the register pressure. For example, compare the sequence of machine instructions of the original and the load-transformed code for *hmmsearch* in Figure 7. Up to the first conditional branch in BB1 in Figure 7(a), there are only three registers defined. Whereas up to the corresponding conditional move instruction in Figure 7(b) there are seven registers defined. This can cause register spills in a register-scarce architecture such as the Pentium 4 (only eight logical registers), which diminishes the benefit of our scheduling. Overall, on the large input sets, the load-transformed programs achieve a harmonic mean speedup of 25.4%, 15.1%, and 4.3% on the Alpha, the Power PC, and the Pentium 4 machines, respectively.

		Alpha	Power PC	Pentium 4	Itanium
dnapenny	original	86.3	61.7	84.5	n.a.
	load-transformed	82.7	56.3	84.5	n.a.
hmmpfam	original	2415.8	825.1	1314.0	922.6
	load-transformed	2025.2	738.7	1229.2	892.5
hmmsearch	original	2461.8	1387.2	1268.5	628.4
	load-transformed	1280.9	1089.9	1139.5	490.8
hmmcalibrate	original	63.3	34.4	45.6	15.4
	load-transformed	37.7	26.0	43.3	11.9
predator	original	673.7	269.8	389.2	344.2
	load-transformed	647.6	266.2	385.6	325.6
clustalw	original	3692.5	1887.8	1612.4	1142.4
	load-transformed	3367.3	1657.1	1580.4	1105.6

Table 8: Absolute runtime in seconds with the large input sets

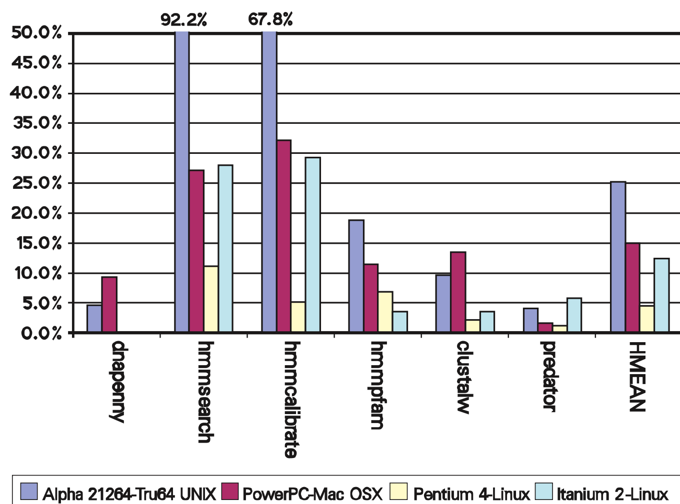


Figure 9: Speedup of load-transformed code over the original code and harmonic mean speedup

On the Itanium, which, unlike the other three platforms, is an in-order machine, we also see substantial speedups on the six bioinformatics applications with our transformed code. Note that the Itanium has only a single cycle L1 cache hit latency and supports speculative and advanced loads in software, which should allow the compiler to perform load scheduling the way we propose even without perfect disambiguation information. Nevertheless, we observe quite a significant speedup on this platform. The reason is that our scheduling expands the basic block length and allows more independent instructions to be issued together in a cycle. There is no speculative element involved. To move loads past a branch on the Itanium, the compiler has to use control speculation, which necessitates putting recovery code in place in case of a misspeculation. Since the branches in question are hard to predict, the recovery code is executed quite often, which degrades performance in the baseline case. Note, however, that using the `restrict` keyword to indicate that the variables (arrays) in question are non-overlapping allows the compiler to turn these speculative loads into non-speculative loads. As a result, the baseline code with `restricts` and our load-transformed code perform similarly. The `restrict` keyword does not help on the other three platforms we evaluated.

6. Related Work

Li et al. [14] characterize the bioinformatics applications in the BioInfoMark suite on an Intel Pentium 4 machine. For the same applications with the same input sets, the Pentium 4 load characteristics are quite similar to those of the Alpha (Section 2.1 and 2.2). Their work also covers other characteristics beyond what we focus on in this paper.

Golden and Mudge [8] recognize that the data-cache hit latency can adversely affect the performance of a five-stage in-order pipeline machine and propose a hardware structure called Load Target Buffer (LTB) to hide the cache hit latency. However, even with a large LTB, the resulting performance improvement is only moderate. Austin and Sohi [2] propose the idea of zero-cycle loads through a combination of instruction predecoding, base register caching, and fast address calculation. Those mechanisms can tolerate the load latency in an in-order issue machine well, but do not see much benefit in an out-of-order issue machine. Calder and Reinman [5] survey speculative techniques for hiding the load latency, namely dependence prediction, address prediction, value prediction, and memory renaming. They then propose a load speculation chooser, which outperforms any one of the surveyed techniques in isolation.

We scheduled load instructions at the source code level to hide the L1 latency. At the machine instruction level, the subject of scheduling load instructions has been studied extensively. Global instruction scheduling [4], [6] allows loads to be moved across basic block boundaries. Hardnett et al. [9] propose a load scheduling algorithm for VLIW machines. To be able to move a load safely past a store, memory disambiguation must be performed to ensure that there is no dependence between the load-store pair. This can be done purely in software with static compiler analysis or with software-only dynamic disambiguation where the compiler inserts disambiguation code in the instruction stream [15]. Yoaz et al. [18] propose a speculative hardware-assisted technique to handle memory disambiguation. IA-64 [7] provides hardware-assisted data and control speculation mechanisms that allow loads to be safely moved past stores and branches.

7. Conclusions

This paper studies the characteristics of load instructions in selected bioinformatics applications from the BioPerf benchmark suite. It shows that their loads, which almost always hit in the L1 data cache, still represent a serious performance bottleneck. Even in the presence of an out-of-order execution core and the use of an aggressive optimizing compiler, the two to three cycle L1 hit latency cannot always be hidden. When we hid this latency through manual load scheduling at the source-code level, we obtained substantial performance improvements. The six bioinformatics applications we investigated can be sped up in this way by 25.4%, 15.1%, 4.3%, and 12.7% on average on Alpha, Power PC, Pentium 4, and Itanium platforms, respectively. This speedup is achieved over baseline code that was compiled with the same high optimization level and with feedback-directed optimization.

8. References

[1] K. Albayraktaroglu, A. Jaleel, X. Wu, B. Jacob, M. Franklin, C.-W. Tseng and D. Yeung, BioBench: A Benchmark Suite of Bioinformatics Applications, IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX, 2005, pp. 2-9.

[2] T. Austin and G. Sohi, Zero-Cycle Loads: Microarchitecture Support for Reducing Load Latency, Proceedings of the 28th Annual International Symposium on Microarchitecture, Ann Arbor, MI, 1995, pp. 82-92.

[3] D. A. Bader, Y. Li, T. Li and V. Sachdeva, BioPerf: A Benchmark Suite to Evaluate High-Performance Computer Architecture on Bioinformatics Applications, IEEE International Symposium on Workload Characterization, 2005, pp. 163-173.

[4] D. Bernstein and M. Rodeh, Global Instruction Scheduling for Superscalar Machines, Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation, 1991, pp. 241-255.

[5] B. Calder and G. Reinman, A Comparative Survey of Load Speculation Architectures, Journal of Instruction-Level Parallelism (2000).

[6] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Water and W.-m. W. Hwu, IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors, Proceedings of the 18th Annual Int'l Symposium on Computer Architecture, Toronto, Canada, 1991, pp. 266-275.

[7] Intel Corporation, Intel IA-64 Architecture Software Developer's Manual, Santa Clara, CA, 2000.

[8] M. Golden and T. Mudge, Hardware Support for Hiding Cache Latency, University of Michigan Technical Report CSE-TR-152-93, 1995, pp. 1-21.

[9] C. R. Hardnett, K. V. Palem, R. M. Rabbah and W.-F. Wong, Scheduling Load Operations on VLIW Machines, Georgia Institute of Technology Technical Report GIT-CC-01-015, Georgia Institute of Technology 2001.

[10] http://developer.apple.com/hardware/ve/g5.html

[11] http://www.intel.com/design/Pentium4/documentation.htm

[12] R. E. Kessler, The Alpha 21264 microprocessor, IEEE Micro, 1999, pp. 24-36.

[13] Y. Li and T. Li, BioInfoMark: A Bioinformatic Benchmark Suite for Computer Architecture Research, Technical Report, IDEAL Research, ECE Dept, University of Florida, 2005.

[14] Y. Li, T. Li, T. Kahveci and J. A. B. Fortes, Workload Characterization of Bioinformatics Applications, IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005, pp. 15-22.

[15] A. Nicolau, Run-time disambiguation: Coping with statically unpredictable dependencies, IEEE Transactions on Computers (TOC), 38 (1989), pp. 664-678.

[16] P. Rice, I. Longden and A. Bleasby, EMBOSS: The European Molecular Biology Open Software Suite, Trends in Genetics, 16 (2000), pp. 276-277.

[17] A. Srivastava and A. Eustace, ATOM: A System for Building Customized Program Analysis Tools, In Proceedings of SIGPLAN 1994, pp. 196-205.

[18] A. Yoaz, M. Erez, R. Ronen and S. Jourdan, Speculation techniques for improving load related instruction scheduling, Proceedings of the 26th Annual International Symposium on Computer Architecture, 1999, pp. 42-53.