Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses

JACK W. DAVIDSON and SANJAY JINTURKAR

{jwd,sj3e}@virginia.edu

Department of Computer Science, Thornton Hall

University of Virginia

Charlottesville, VA 22903 U. S. A.

ABSTRACT

As microprocessor speeds increase, memory bandwidth is increasingly the performance bottleneck for microprocessors. This has occurred because innovation and technological improvements in processor design have outpaced advances in memory design. Most attempts at addressing this problem have involved hardware solutions. Unfortunately, these solutions do little to help the situation with respect to current microprocessors. In previous work, we developed, implemented, and evaluated an algorithm that exploited the ability of newer machines with wide-buses to load/store multiple floating-point operands in a single memory reference. This paper describes a general code improvement algorithm that transforms code to better exploit the available memory bandwidth on existing microprocessors as well as wide-bus machines. Where possible and advantageous, the algorithm coalesces narrow memory references into wide ones. An interesting characteristic of the algorithm is that some decisions about the applicability of the transformation are made at run time. This dynamic analysis significantly increases the probability of the transformation being applied. The code improvement transformation was implemented and added to the repertoire of code improvements of an existing retargetable optimizing back end. Using three current architectures as evaluation platforms, the effectiveness of the transformation was measured on a set of compute- and memory-intensive programs. Interestingly, the effectiveness of the transformation varied significantly with respect to the instruction-set architecture of the tested platform. For one of the tested architectures, improvements in execution speed ranging from 5 to 40 percent were observed. For another, the improvements in execution speed ranged from 5 to 20 percent, while for yet another, the transformation resulted in slower code for all programs.

1 INTRODUCTION

Processor speeds are increasing much faster than memory speeds. For example, microprocessor performance has increased by 50 to 100 percent in the last decade, while memory performance has increased by only 10 to 15 percent. Additional hardware support such as larger, faster caches [Joup90], software-assisted caches [Call91], speculative loads [Roge92], stream memory controllers [McKe94], and machines with wider memory buses, helps, but the problem is serious enough that performance gains by any approach, including software, are worth pursuing. Furthermore, even with additional hardware, processors often do not obtain anywhere near their peak performance with respect to their memory systems.

This paper describes a code improvement transformation that attempts to utilize a processor's memory system more effectively by coalescing *narrow* loads and stores of width N bits into more efficient *wide* loads and stores of width $N \times c$ where c is a multiple of two and the processor can fetch $N \times c$ bits efficiently. The terms narrow and wide are relative to the target architecture. On a 16-bit architecture, for example, two narrow loads of bytes (8-bits) that are in consecutive memory locations might be coalesced into a single wide load of 16 bits. Similarly, on a 64-bit architecture, four narrow stores of words (16-bits) that are in consecutive memory locations and properly aligned might be coalesced into a single wide store of 64-bits.

As the paper shows, the analysis to perform such transformations is difficult but doable, and in many cases well worth the effort. The two questions that the analysis must answer are: Is the transformation safe and is the transformation profitable? Safety analysis determines whether the transformation can be done without changing the semantics of the program. The two key components of the safety analysis address aliasing and data alignment issues. Alias analysis, in particular, is extremely difficult when the source language contains unrestricted pointers [Land92, Land93]. The problem is further compounded because for many codes where this transformation would be beneficial, the code is structured so that aliasing and data alignment hazards cannot precisely be determined via interprocedural, compile-time analysis. The paper describes a new technique, called run-time alias and alignment analysis, that neatly solves this problem.

Profitability analysis determines whether the transformation will result in code that runs faster. This is perhaps the most difficult part of the analysis because memory coalescing interacts with other code improvements. For example, to expose more narrow, consecutive memory references for possible coalescing, loops are sometimes unrolled by the optimizer. However, naive loop unrolling may cause the size of a loop to grow larger than the instruction cache, and any gains in performance by memory coalescing may be more than offset by degraded cache performance. Similarly, memory coalescing collects memory accesses that are distributed throughout the loop into a single reference. This gathering of dependencies into a single instruction can adversely affect instruction scheduling. The paper discusses

how these and other issues are resolved so that the memory coalescing yields code that runs faster, not slower.

The following section briefly discusses work related to reducing memory bandwidth requirements of programs. Section 2 describes the algorithm with emphasis on the analyses required to apply the transformation safely and profitably. Section 3 describes the implementation of the algorithm in an existing retargetable back end called *vpo* [Beni89, Beni94]. Using a C front end and *vpo*, the effectiveness of the transformation was evaluated on three processors: DEC's Alpha [Digi92], Motorola's 88100 [Moto91], and Motorola's 68030 [Moto85]. Section 4 contains a summary.

1.1 RELATED WORK

Software approaches to the memory bandwidth problem focus on reducing the memory bandwidth requirements of programs. For example, there is a plethora of research describing algorithms for register allocation, a fundamental transformation for reducing a program's memory bandwidth requirements. Register allocation identifies variables that can be held in registers. By allocating the variable to a register, memory loads and stores previously necessary to access the variable are eliminated. An evaluation of the register coloring approach to register allocation showed that up to 75 percent of the scalar memory references can be removed using these techniques [Chow90].

Cache blocking and register blocking are code transformations that also reduce a program's memory bandwidth requirement. These transformations can profitably be applied to codes that process large sets of data held in arrays. For example, consider the multiplication of two large arrays. By large, we mean that the arrays are much larger than the size of the machine's data cache.

Because the arrays are larger than the cache, processing the entire array results in data being read from memory to the cache many times. Cache blocking, however, transforms the code so that a block of the array that will fit in the cache is read in once, used many times, and then replaced by the next block. The performance benefits from this transformation can be quite good. Lam, Rothberg, and Wolf [Lam91] show that for multiplication of large, floating-point arrays, cache blocking can easily triple the performance of a cache-based system.

Register blocking is similar in concept to cache blocking, but instead of transforming code to reduce the number of redundant loads of array elements into the cache, it transforms code so that unnecessary loads of array elements are eliminated. Register blocking can be considered a specific application of scalar replacement of subscripted variables [Call90, Dues93] and loop unrolling. Scalar replacement identifies reuse of subscripted variables and replaces them by references to temporary scalar variables. Unrolling the loop exposes a block of these subscripted variables to which scalar replacement can be applied. Register blocking and cache blocking can be used in combination to reduce the number of memory references and cache misses.

Another program transformation that reduces a program's memory bandwidth requirements is called recurrence detection and optimization [Beni91]. Knuth defines a recurrence relation as a rule that defines each element of a sequence in terms of the preceding elements [Knut73]. Recurrence relations appear in the solutions to many compute- and memory-intensive problems. Interestingly, codes containing recurrences often cannot be vectorized. Consider the following C code:

```
for (i = 2; i < n; i++)
   x[i] = z[i]* (y[i] - x[i-1]);
```

This is the fifth Livermore loop, which is a tri-diagonal elimination below the diagonal. It contains a recurrence since `x[i]` is defined in terms of `x[i-1]`. By detecting the fact that a recurrence is being evaluated, code can be generated so that the `x[i]` computed on one iteration of loop is held in a register and is obtained from that register on the next iteration of the loop. For this loop, the transformation yields code that saves one memory reference per loop iteration.

For machines with wide-buses (the size of the bus is greater than the size of a single-precision floating-point value), it is possible to compact some number of floating-point loads into a single reference [Alex93]. Indeed, the work reported here is a generalization and extension of this technique applied to data of any size. We call this technique memory access coalescing. This technique can be used with the techniques mentioned previously.

2 MEMORY ACCESS COALESCING

2.1 MOTIVATION

To describe memory access coalescing and highlight the potential hazards that must be handled by an optimizer, consider the C code in Figure 1a. The code computes the dot product of two vectors containing 16-bit integers. The code is taken from a signal processing application, and 16-bits was sufficient to represent the dynamic range of the sampled signal.

Figure 1b contains the DEC Alpha machine code in register transfer lists (RTLs) generated by our compiler. Because the DEC Alpha is a relatively new architecture, and because it has some interesting characteristics that affect code generation, a few relevant details of the architecture are described. There are 32 64-bit fixed point registers, and all operations are performed on 64-bit registers. The load and store instructions can move 32-bit (longword) or 64-bit (quadword) quantities from and to memory. Memory addresses must be *naturally aligned*. Data that is 2^N bytes in size is naturally aligned if it is stored at an address that is a multiple of 2^N. To accommodate loading and storing of data that is unaligned, the architecture contains unaligned loads and stores of 64-bits. These instructions fetch the aligned quadword that contains the unaligned data. There is a full complement of arithmetic and logical instructions that manipulate 64-bit values. In addition, there are three instructions, add, subtract, and multiply, that operate on 32-bit data. In a departure from other RISC architectures, the Alpha does not include instructions for loading and storing bytes or shortwords (16-bits). Instead, architectural support is provided for extracting 8-bit (byte) and 16-bit (shortword) quantities from 64-bit registers. For example, there are instructions for efficiently extracting and inserting bytes or shortwords from/to a register. The rationales for these design decisions are outlined in the Alpha architecture handbook [Dec92].

With this information in mind, the code in Figure 1b can be explained. In the RTL code, `q[`*n*`]` and `r[`*n*`]` refer to fixed-point registers. `r[`*n*`]` is used when the operation is 32-bit. In the code, `Q[`*addr*`]` refers to quadword memory. The unaligned load instruction at line 12 fetches the aligned quadword that contains `a[i]`. It is necessary to use an unaligned load because the base addresses of `a` and `b` are not guaranteed to be aligned on a quadword boundary, but they are guaranteed to be aligned on a shortword boundary. The instructions at lines 14 through 16 extract the shortword from the quadword. Line 14 computes the offset of the shortword within the register.

```
int dotproduct(short a[], short b[], int n) {
    int c, i;

    c = 0;
    for (i = 0; i < n; i++)
        c += a[i] * b[i];
    return c;
    }
```

Figure 1a. Dot-product loop.

```
1.      r[4] = 0;
2.      // test n for zero-trip loop
3.      r[0] = r[31] - r[18];
4.      PC = r[0] >= 0 -> L15;
5.      // compute address of a+n*2
6.      q[6] = r[18] << 32;
7.      q[6] = q[6] >> 32;
8.      q[6] = q[6] << 1;
9.      q[6] = q[16] + r[6];
10. L17
11.     // load quad containing a[i]
12.     q[2] = Q[(q[16])&~7];
13.     // extract and sign extend a[i]
14.     q[8] = q[16] + 2;
15.     q[1] = EQH[q[2],q[8]];
16.     r[1] = q[1] >> 48;
17.     // load quad containing b[i]
18.     q[3] = Q[(q[17])&~7];
19.     // extract and sign extend b[i]
20.     q[8] = q[17] + 2;
21.     q[2] = EQH[q[3],q[8]];
22.     r[2] = q[2] >> 48;
23.     // compute product and accumulate
24.     r[1] = r[1] * r[2];
25.     r[4] = r[4] + r[1];
26.     / advance to next array elements
27.     q[17] = q[17] + 2;
28.     q[16] = q[16] + 2;
29.     // test for loop termination
30.     q[0] = q[16] - q[6];
31.     PC = q[0] < 0 -> L17;
32. L15
33.     r[0]=r[4];
```

Figure 1b. Original code for loop.

```
1.      r[4] = 0;
2.      // test for zero-trip loop
3.      r[0] = r[31] - r[18];
4.      PC = r[0] >= 0 -> L15;
5.      // compute loop termination
6.      q[6] = r[18] << 32;
7.      q[6] = q[6] >> 32;
8.      q[6] = q[6] << 1;
9.      q[6] = q[16] + q[6];
10. L17
11.     // load quad containing a[i]
12.     q[21] = Q[[16]];
13.     // extract a[i] (two bytes)
14.     q[8] = q[16] + 2;
15.     q[1] = EQH[q[21],q[8]];
16.     r[1] = q[1] >> 48;
17.     // load quad containing b[i]
18.     q[20] = Q[q[17]];
19.     // extract b[i] (two bytes)
20.     q[8] = q[17] + 2;
21.     q[2] = EQH[q[20],q[8]];
22.     r[2] = q[2] >> 48;
23.     // compute dot product
24.     r[1] = r[1] * r[2];
25.     r[4] = r[4] + r[1];
26.     // extract a[i+1]
27.     q[8] = q[16] + 4;
28.     q[1] = EQH[q[21],q[8]];
29.     r[1] = q[1] >> 48;
30.     // extract b[i+1]
31.     q[8] = q[17] + 4;
32.     q[2] = EQH[q[20],q[8]];
33.     r[2] = q[2] >> 48;
34.     // compute dot product
35.     r[1] = r[1] * r[2];
36.     r[4] = r[4] + r[1];
37.     ...
38.     ...
39.     // adv to next a[i] & b[i]
40.     q[16] = q[16] + 8;
41.     q[17] = q[17] + 8;
42.     // test for loop termination
43.     q[0] = q[16] - q[6];
44.     PC = q[0] < 0 -> L17;
45. L15
46.     r[0]=r[4];
```

Figure 1c. Unrolled loop with coalesced memory references.

Figure 1: DEC Alpha code for dot-product.

```
1   // Main routine to coalesce memory accesses
2   proc CoalesceMemoryAccesses(CurrFunction) is
3       // Consider each loop in the current function.
4       ∀LOOP ∈ CurrFunction.Loop do
5           LOOP.InductionVars ← FindInductionVars(LOOP)
6           // Unroll the loop. If it fits in the cache, use it, else use the rolled loop.
7           CurrFunction.Loop ← {UnRollLoopIfProfitable(LOOP)} ∪ CurrFunction.Loop
8           // Classifies memory references into different partitions if a unique identifier is found to distinguish
9           // a set of such references. Thus, all references to an array A passed as a parameter will have a loop
10          // invariant register (most probably the register containing the start address of A) as their partition
11          // identifier.
12          ClassifyMemoryReferencesIntoPartitions(LOOP)
13          // Calculate relative offsets of the memory references belonging to same partition from the induction variable.
14          // If a constant offset is not found, it is not safe to do memory coalescing. Sort the offsets.
15          CalculateRelativeOffsets(LOOP)
16          EliminateInductionVariables(LOOP)
17          // Attempt Wide reference optimization
18          WideRefOptimization(LOOP, CurrFunction)
19          enddo
20      endproc
```

Figure 2: Memory access coalescing algorithm main loop.

The RTL

```
q[1] = EQH[q[2],q[8]]
```

shifts register `q[2]` left by the number of bytes specified by the low three bits of `q[8]`, inserts zeros into the vacated bit positions, and then extracts 8 bytes into register `q[1]`. Line 16 sign extends the shortword. Lines 18 through 22 perform a similar operation for `b[i]`.

The code in Figure 1b, as it stands, is fairly tight. However, the loop fetches the same quadwords for the respective arrays every four iterations. Thus, for every four iterations six redundant loads are executed. It is these redundant memory accesses that memory coalescing eliminates. By unrolling the loop four times, and applying the memory coalescing algorithm, the optimizer produces the code in Figure 1c. Notice that there are still two loads in the loop (lines 12 and 18), but the modified loop iterates one-fourth as many times as the loop of Figure 1b. Thus, the original loop performs $2 \times n$ memory references, while the coalesced loop performs $\frac{n}{2}$ memory references for a savings of 75 percent.

At this point, the transformation may seem rather straightforward. However, there are subtle details that must be addressed. First, the code in Figure 1c assumes that the starting address of the vectors a and b are aligned on a quadword boundary. This may, or may not be true. If it is not true, the first memory reference to an unaligned address will trap. Second, the code also assumes that the length of the vectors is divisible by four. This too may, or may not be true. If it is not true, the loop will fetch data outside the arrays and possibly fault (we'd be lucky if it did), but is more likely that silently an incorrect result will be computed. Third, the loop body has gotten larger, and the assumption is that any potential negative effects due to increasing the size of the loop will be offset by the gains resulting from reducing the number of memory references. This may or may not be a reasonable assumption.

2.2 MEMORY ACCESS COALESCING ALGORITHM

These safety and profitability issues mentioned above, and others, must be handled by the memory access coalescing algorithm. The C code in Figure 1a highlights the difficulty of this analysis. For this routine, standard intraprocedural analysis cannot gather the necessary information to safely coalesce memory references. The vectors and n, the number of elements in the arrays, are parameters. Interprocedural analysis would help, but often the routines of interest are part of a library and are not accessible until link time. One could limit the applicability of the algorithm to routines where static, compile-time analysis is sufficient, but our experience shows that this would eliminate most opportunities for applying the algorithm. The approach taken here attempts to do the analysis at compile time, if possible, but if it is not possible code is generated to check the safety issues at run time. Our evaluations show that it is generally possible to do this in a way that the impact of the extra code for checking is negligible.

Figure 2 contains the high-level portion of the algorithm. Due to space limitations, the entire algorithm cannot be presented. The focus here is the profitability and safety analysis. Line 7 determines if it is profitable to unroll the loop. Our heuristic is that if the original loop will fit in the instruction cache, then the algorithm must ensure that the unrolled loop will fit as well. In addition, this routine, if necessary, produces code to execute the loop body enough times so that the number of iterations of the main loop is a multiple of the unrolling factor. Line 12 analyzes the memory reference of the loop and partitions them into disjoint sets for later analysis [Beni91]. The memory access coalescing is done by WideRefOptimization. After identifying candidate memory references for coalescing, DoProfitabilityAnalysisAndModify is called. The algorithm is in Figure 3.

The algorithm makes a copy of the loop and performs memory coalescing on it. This involves not only coalescing the memory accesses, but inserting code to extract the required information from the coalesced memory reference. After doing this, it calls the scheduler to schedule both loops and does a comparison. If it appears advantageous to use the coalesced loop, then various

```
1   // Do Cost/Benefit analysis before doing memory coalescing
2   proc DoProfitabilityAnalysisAndModify(LOOP, S, T, WideSize, CurrFunction) is
3       Inst ← ∅
4       // Do data hazard analysis for possible aligned and unaligned wide references
5       AlignedWideType ← DoHazardAnalysis(LOOP, S, ALIGNED, LOOP.PossibleAliases, Inst, WideSize
                                    AlignedWideReferencePosition, AlignedWideReferenceAddress)
6       UnAlignedWideType ← DoHazardAnalysis(LOOP, S, ¬ALIGNED, LOOP.PossibleAliases, Inst,
                        WideSize, UnalignedWideReferencePosition, UnAlignedWideReferenceAddress)
7       // Chack if a valid Wide memory reference which can replace the narrow references is found
8       if IsValidType(AlignedWideType) ∨ IsValidType(UnAlignedWideType) then
9           // Make a copy of the loop. Schedule the instructions in the original loop and find the number of cycles necessary.
10          // Then insert appropriate wide references in the copy of the loop and schedule it too. If the latter requires
11          // less cycles, then go ahead.
12          LCOPY ← DoReplication(LOOP)
13          // Calculate the cycles required by the original loop by static scheduling
14          CyclesforOriginalLoop ← Schedule(LOOP)
15          if IsValidType(AlignedWideType) then
16              InsertWideReferences(LCOPY, S, AlignedWideReferencePosition,
                    AlignedWideReferenceAddress)
17              endif
18          if IsValidType(UnAlignedWideType) then
19              InsertWideReferences(LCOPY, T, UnAlignedWideReferencePosition,
                    UnAlignedWideReferenceAddress)
20              endif
21          // Calculate the cycles required by the loop after replacing the narrow references by wide ones.
22          CyclesforCopiedLoop ← Schedule(LCOPY)
23          if CyclesforCopiedLoop < CyclesforOriginalLoop then
24              // If the alignment checking for the aligned wide address under consideration is not there,
25              // then insert a check that will allow the execution of the LCOPY if the address is actually aligned
26              // at runtime
27              if ¬AlignmentCheckExists(LOOP, WideType, WideReferenceAddress) then
28                  InsertAlignmentCheckInPreheader(LOOP.Preheader, LOOP.Label, LCOPY.Label
                                                    WideReferenceAddress, WideType)
29                  InsertAliasingChecksInPreheader(LOOP.Preheader, LOOP.Label, LCOPY.Label, Inst)
30              else
31                  // Else just use the LCOPY instead of the original one, since this is better
32                  Target ← FindTargetOfUnalignedAddress(LOOP.Preheader)
33                  InsertAliasingChecksInPreheader(LOOP.Preheader, LCOPY.Label, Target.Label, Inst)
34                  ChangeATargetOfAlignmentCheck(LOOP.Preheader, LOOP.Label, LCOPY.Label)
35                  CurrFunction.Loop ← CurrFunction.Loop − {LOOP}
36                  endif
37              CurrFunction.Loop ← CurrFunction.Loop ∪ {LCOPY}
38              endif
39          if LOOP ∈ Curr.Function then
40              ∀Z ∈ S ∪ T do
41                  Z.Modify ← TRUE
42                  enddo
43              endif
44          return TRUE
45          endif
46      return FALSE
47      endproc
```

Figure 3: Profitability analysis algorithm.

```
1   // Check if there are any data hazards
2   proc IsHazard(S, WideReferencePosition, ReferenceType, C, Inst) is
3       ∀M ∈ S do
4           // A wide load is inserted before the dominating loads of all the narrow loads. So the narrow load reference is
5           // called the BottomInst. A wide store is inserted after the dominated store of all the narrow stores. So the
6           // narrow store is called the TopInst.
7           if ReferenceType = LOAD then
8               BottomInst ← M
9               TopInst ← WideReferencePosition
10          else
11              TopInst ← M
12              BottomInst ← WideReferencePosition
13              endif
14          // The narrow and the wide reference have to lie in the same basic block
15          if BottomInst.BasicBlock ≠ TopInst.BasicBlock then
16              Return(TRUE)
17              endif
18          CurrInst ← BottomInst
19          // Check all the instructions between the BottomInst and TopInst
20          while CurrInst ← CurrInst.PrevInst ∧ CurrInst ≠ TopInst do
21              // We cannot allow a load between two stores, all belonging to same partition. If they do not lie in same
22              // partition, there is a possibility of aliasing, which can probably be detected only at run time.
23              if IsStore(M.Reference) ∧ IsLoad(CurrInst.Reference) then
24                  if (M.Paritition = CurrInst.Partition) then
25                      if ReferenceSameLocation(M.Reference, CurrInst.Reference) ∧
                                ¬IsNeededTodoNarrowStoreOnly(M.Reference, CurrInst.Reference) then
26                      Return(TRUE)
27                          endif
28                  else
29                      Inst ← Inst ∪ DoAliasDetection(CurrInst.Partition.Load, M.Partition.Store, C)
30                      endif
31              // We cannot allow a store between two load or store references.
32              elseif IsStore(CurrInst.Reference) then
33                  if (M.Paritition = CurrInst.Partition) then
34                      if ReferenceSameLocation(M.Reference, CurrInst.Reference)
35                          Return(TRUE)
36                          endif
37                  elseif IsLoad(M.Reference) then
38                      Inst ← Inst ∪ DoAliasDetection(CurrInst.Partition.Store, M.Partition.Load, C)
39                  else
40                      Inst ← Inst ∪ DoAliasDetection(CurrInst.Partition.Store, M.Partition.Store, C)
41                      endif
42                  endif
43              FindBaseAndDisplacementOfAddress(CurrInst.Reference, Base, Displacement)
44              // If the base register has been modified, then the coalescing may not be safe.
45              if IsModifiedBase(CurrInst.Reference, Base) then
46                  Return(TRUE)
47                  endif
48              endwhile
49          endfor
50      // No data hazards were found
51      Return(FALSE)
52      endproc
```

Figure 4: Hazard analysis algorithm.

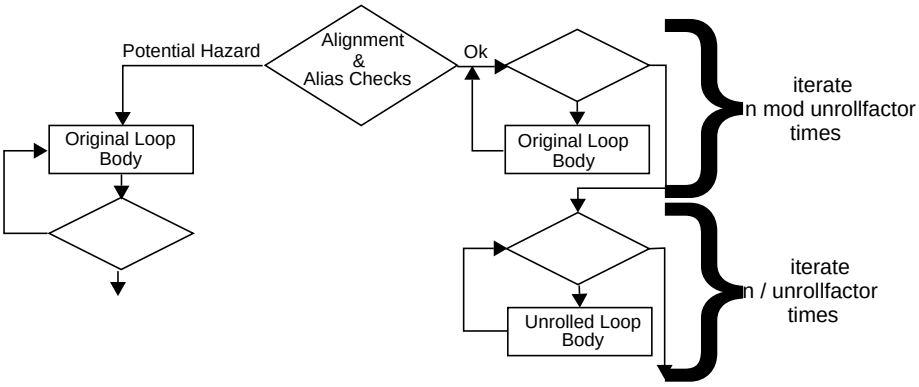

Figure 5: Flow graph showing alignment and alias checks.

checks are done to see if it is necessary to put alignment checks in the preheader of the loop. Additionally, if it is not possible to do static alias detection (for example, do the memory references overlap), then code is inserted in the preheader to do the checks at run time.

A second key algorithm is IsHazard that does the safety analysis. This routine is contained in Figure 4. Most of the analysis here is straightforward. The routine assures that coalesced memory references are in the same basic block and that sequential consistency is preserved. In addition if aliasing cannot be resolved statically, the routine DoAliasDetection is called which generates code that will be inserted in the loop preheader to check for potential aliasing problems (e.g., two arrays overlap in memory). If at run time, an alias condition is detected, the original safe loop is executed.

The result is code represented by the flow graph in Figure 5. For the example in Figure 1, this results in additional code being added to the loop preheader for each possible alias pair <a,b>. In particular, the following instructions are added:

```
//  q[16]: address of a;
//  q[17]: address of b; q[18]: n
    q[1] = q[17] + q[18];
    q[0] = q[16] < q[1];
    PC = q[0] <= 0 -> L16;
    q[1] = q[16] + q[18];
    q[0] = q[17] < q[1];
    PC = q[0] > 0 -> L13;
L16
    q[0] = q[18] % 4;
    PC = q[0] != 0 -> L13;
    q[0] = q[16] & 7;
    PC = q[0] != 0 -> L13;
    q[0] = q[17] & 7;
    PC = q[0] != 0 -> L13;
//  do memory coalesced unrolled loop
    ...
    ...
L13
//  do original “safe” loop
```

The code appearing before L16 checks to make sure the arrays do not overlap, while the code after L16 checks for the ability to unroll and that the arrays are properly aligned.

3 IMPLEMENTATION AND RESULTS

A prototype implementation of the memory access coalescing algorithm has been implemented in an existing retargetable compiler, and tested on platforms containing the following processors: DEC Alpha, Motorola 88100, and Motorola 68020. Using a set of compute- and memory-intensive kernel loops listed in Table I, the effectiveness of the algorithm was evaluated. These benchmarks were chosen because they represent realistic code, and they contain loops that are memory-intensive and contain memory references that are candidates for memory access coalescing. Memory access coalescing, unlike register allocation, code motion, induction variable elimination, etc., is not a code improvement that applies to code in general. However, like cache blocking, register blocking, iteration space tiling, software pipelining, and recurrence detection and optimization, it does apply to a small set of important codes, and as the results show, it provides high payoff when it does apply.

The results for the DEC Alpha are presented in Table II. All timings were gathered by running each program ten times on a single-user machine. The two highest execution times and the two lowest were discarded, and an average of the remaining six times was taken. Column 2 (labeled cc -O) is the execution time taken by code produced by the native compiler with the loop unrolled. Column 3 is the execution time taken by the code produced by our compiler again with the loop unrolled. The loops were unrolled so that the effect of memory access coalescing could be isolated and observed. Column 4 contains the average execution time for the benchmark when loads were coalesced, and column 5 contains the average execution time when both loads and stores were coalesced. Column 6 contains the percentage speedup. The first thing to notice is that the optimizing compiler in which the memory access coalescing algorithm is embedded is comparable to the native compiler. This indicates that the speedups in column 6 are not artifacts of embedding the algorithm in a poor compiler. The second thing to notice is that, in general, the percentage speedup is quite good.

Table III contains similar timing information for the Motorola 88100-based platform. It is interesting to note that the code with both loads and stores coalesced runs slower than the code with just loads coalesced. The reason is that the Motorola 88100 has efficient instructions for extracting bytes and words from a 32-bit

Program	Description	Lines of Code
Convolution	Gradient Directional Edge Convolution of a 500 by 500 black and white Image [Lind91]	154
Image Add	Image addition of two 500 by 500 black and white frames	48
Image xor	Image addition of two 500 by 500 black and white frames	48
Translate	Translate 500 by 500 black and white image image to a new position	48
Eqntott	Part of the SPEC 89 benchmark suite	146
Mirror	Generate mirror image of 500 by 500 black and white image	50

Table I: Compute- and memory-intensive benchmarks.

Program	cc -O	vpcc/vpo -O	vpcc/vpo -O (coalesce loads)	vpcc/vpo -O (coalesce loads and stores)	Percent Savings $\frac{(Col_3 - Col_5)}{Col_2} \times 100$
Convolution	16.67	17.76	15.62	15.76	11.26
Image add	17.41	17.71	11.48	10.44	41.05
Image add (16-bit)	12.03	12.02	8.97	8.13	32.36
Image xor	17.43	17.49	11.48	10.48	40.08
Translate	11.46	10.52	8.45	7.04	33.11
Eqntott	19.17	21.55	20.72	20.72	3.86
Mirror	15.62	14.49	12.63	9.84	32.09

Table II: DEC Alpha execution times (in seconds) and percent improvement.

register, but there are no instructions for inserting bytes and words into a register without affecting the other bytes or words in the register. Thus, code must be generated using logical instructions to place the word into the proper position in the register. These sequences outweigh the gains of coalescing stores. However, coalescing loads was profitable exhibiting speedups of up to 25 percent.

We also implemented the algorithm in a compiler for the Motorola 68030. Unfortunately, in all cases the code ran slower. Inspection of the code revealed that while the Motorola 68030 has instructions for extracting bytes and words, these are much more expensive than simply loading the bytes and words directly. This again highlights how most optimizations are machine dependent.

4 SUMMARY

We have described an algorithm for coalescing redundant memory accesses in loops. If possible, static analysis is used to resolve safety and profitability issues. However, in most interesting cases, it is necessary to rely on runtime tests to handle aliasing and alignment issues. Such code is relatively easy to generate. Typically, 10 to 15 instructions must be added in the loop preheader to check for possible hazards. The results on two machines show that the technique can result in substantial speedups. For the DEC

Program	cc -O	vpcc/vpo -O	vpcc/vpo -O (coalesce loads)	vpcc/vpo -O (coalesce loads and stores)	Percent Savings $\frac{(Col_3 - Col_4)}{Col_2} \times 100$
Convolution	22.86	22.82	18.87	22.64	17.3
Image add	15.33	25.74	12.97	13.45	15.39
Image xor	15.34	15.34	12.94	13.7	15.64
Translate	16.32	17.52	13.49	16.91	24.46
Eqntott	130.3	145.0	143.2	143.2	1.3
Mirror	20.52	19.23	16.03	16.89	16.64

Table III: Motorola 88100 execution times (in seconds) and percent improvement.

Alpha, we observed speed ups ranging from 3 percent up to 40 percent. For the Motorola 88100, we observed speed ups of a few percent up to 25 percent, while for the Motorola 68030 the technique resulted in slower code.

ACKNOWLEDGEMENTS

This work was supported in part by National Science Foundation grant CCR-9214904.

REFERENCES

[Alex93] Alexander, M. J., Bailey, M. W., Childers, B. R., Davidson, J. W., Jinturkar, S., "Memory Bandwidth Optimizations for Wide-Bus Machines", *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, Maui, HI, January 1993, pp. 466—475.

[Aho86] Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, 1986.

[Beni94] Benitez, M. E., and Davidson J. W., "The Advantages of Machine-Dependent Global Optimization", *Proceedings of the International Conference on Programming Languages and System Architectures*, Springer Verlag Lecture Notes in Computer Science, Zurich, Switzerland, March, 1994, pp. 105—124.

[Beni91] Benitez, M. E., and Davidson, J. W., "Code Generation for Streaming: an Access/Execute Mechanism", *Proceedings of the Fourth International Symposium on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991, pp. 132—141.

[Beni89] Benitez, M. E., and Davidson J. W., "A Portable Global Optimizer and Linker", *Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, GA, June, 1988, pp.329—338.

[Call91] Callahan, D., Kennedy, K., and Porterfield, A., "Software Prefetching", *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991, pp. 40—52.

[Call90] Callahan, D. and Carr, S. and Kennedy, K. Improving Register Allocation for Subscripted Variables. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, White Plains, NY, June, 1990, pp 53—65.

[Chow90] Chow, F. C. and Hennessy, J. L. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems* 12(4):501—536 October 1990.

[Digi92] *Alpha Architecture Handbook*, Digital Equipment Corporation, 1992.

[Dues93] Duesterwald, E., Gupta, R., and Soffa, M. L., "A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimizations", *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993, pp. 68—77.

[Joup90] Jouppi, N., “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers”, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, Seattle, WA, May 1990, pp. 364—373.

[Knut73] Knuth, D. E., Volume 1: *Fundamental Algorithms*. Addison-Wesley, Reading, MA, 1973.

[Lam91] Lam, M. and Rothberg, E. E. and Wolf, M. E., “The Cache Performance and Optimizations of Blocked Algorithms”, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April, 1991, pp 63—74.

[Land93] Landi, W., Ryder, B. G., and Zhang, S., “Interprocedural Modification Side Effect Analysis with Pointer Aliasing”, *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993, pp. 56—67.

[Land92] Landi, W., and Ryder, B. G., “A Safe Approximation Algorithm for Interprocedural Pointer Aliasing”, *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992, pp. 235—248.

[Lind91] Lindley, C. A., *Practical Image Processing in C*, John Wiley and Sons, Inc., New York, NY, 1991.

[McFa91] McFarling, S., “Procedure Merging with Instruction Caches”, *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, June 1991, pp. 71—79.

[McKe94] McKee, S. A., Klenke, R. H., Schwab, A. J., Wulf, W. A., Moyer, S. A., and Aylor, J. H., “Experimental Implementation of Dynamic Access Ordering”, *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, Maui, HI, January 1994.

[Moto91] *MC88110: Second Generation RISC Microprocessor User's Manual*. Motorola, Inc., Phoenix, AZ, 1991.

[Moto85] MC68020 32-bit Microprocessor User’s Manual, Prentice-Hall, Inc. Englewood Cliffs, N. J. 07632.

[Roge92] Rogers, A., and Li, K., “Software Support for Speculative Loads”, *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1992, pp. 38—50.

[Spec89] Systems Performance Evaluation Cooperative, c/o Waterside Associates, Fremont, CA, 1989.