

Instruction Combining for Coalescing Memory Accesses Using Global Code Motion

Motohiro Kawahito

Hideaki Komatsu

Toshio Nakatani

IBM Tokyo Research Laboratory

1623-14, Shimotsuruma, Yamato, Kanagawa, 242-8502, Japan

{jl25131, komatsu, nakatani}@jp.ibm.com

ABSTRACT

Instruction combining is an optimization to replace a sequence of instructions with a more efficient instruction yielding the same result in a fewer machine cycles. When we use it for *coalescing memory accesses*, we can reduce the memory traffic by combining narrow memory references with contiguous addresses into a wider reference for taking advantage of a wide-bus architecture. Coalescing memory accesses can improve performance for two reasons: one by reducing the additional cycles required for moving data from caches to registers and the other by reducing the stall cycles caused by multiple outstanding memory access requests. Previous approaches for memory access coalescing focus only on array access instructions related to loop induction variables, and thus they miss many other opportunities. In this paper, we propose a new algorithm for instruction combining by applying global code motion to wider regions of the given program in search of more potential candidates. We implemented two optimizations for coalescing memory accesses, one combining two 32-bit integer loads and the other combining two single-precision floating-point loads, using our algorithm in the IBM Java™ JIT compiler for IA-64, and evaluated them by measuring the SPECjvm98 benchmark suite. In our experiment, we can improve the maximum performance by 5.5% with little additional compilation time overhead. Moreover, when we replace every declaration of `double` for an instance variable with `float`, we can improve the performance by 7.3% for the MolDyn benchmark in the JavaGrande benchmark suite. Our approach can be applied to a variety of architectures and to programming languages besides Java.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors – compilers, optimization.

General Terms

Algorithms, Performance, Design, Experimentation

Keywords

Instruction combining, memory access coalescing, 64-bit architectures, Java, JIT compilers, IA-64

1. INTRODUCTION

Instruction combining [28] is an optimization to replace a sequence of instructions with a more efficient instruction yielding the same

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSP 2004, June 8, 2004, Washington, DC, USA.

Copyright 2004 ACM 1-58113-941-1/04/06...\$5.00.

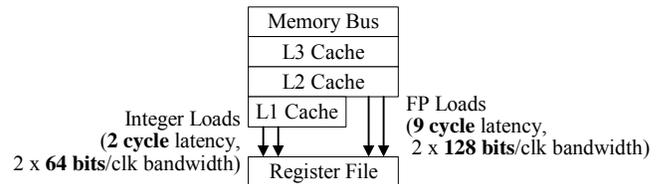


Figure 1. Characteristics of the memory hierarchy of Itanium

result in a fewer machine cycles. Previous approaches for instruction combining can be classified into two families.

The first family combines “two instructions that have a true dependence” (we call them *dependent instructions*) [28, 29]. This family uses global code motion, but it cannot combine instructions along the conditionally executed path.

The second family combines “multiple instructions that do not have a true dependence” (we call them *independent instructions*). This family includes *memory access coalescing* [6], which is an optimization to coalesce narrow memory references with contiguous addresses into a wider reference for taking advantage of a wide-bus architecture.

Coalescing memory accesses can improve performance for two reasons: one by reducing the additional cycles required for moving data from caches to registers and the other by reducing the stall cycles caused by multiple outstanding memory access requests.

In general, the latency of FP loads is longer than that of integer loads, and thus reducing FP loads is more effective. On the other hand, integer loads appear more frequently, and thus reducing integer loads is also effective. For example, on Itanium processor (IA-64) [17], FP loads always bypass the L1 cache and read from the L2 cache as shown in Figure 1 [18]. The latency of FP loads is 9 cycles, while the latency of integer loads is 2 cycles. To take another example, on Pentium 4 and Xeon processors (IA-32), both integer and FP loads are able to read from the L1 cache. However, the latency of FP loads is 6 cycles (for Model 0, 1, 2) or 12 cycles (for Model 3), while the latency of integer loads is 2 cycles (for Model 0, 1, 2) or 4 cycles (for Model 3) [19].

Previous approaches for memory access coalescing focus only on array access instructions related to loop induction variables [1, 6, 27, 31], and thus they miss many other opportunities.

In this paper, we propose a new algorithm for combining multiple instructions by using global code motion to combine both dependent instructions and independent ones along the conditionally executed path. We modify the Lazy Code Motion (LCM) algorithm [25] to attempt to combine those instructions that are located separately in a wider region to coalesce memory accesses.

We implemented two optimizations, one combining two 32-bit integer loads and the other combining two single-precision floating-point loads, using our algorithm in the IBM Java JIT compiler for IA-64, and evaluated them by measuring the SPECjvm98 benchmark suite. In our experiment, we can improve the maximum performance by 5.5% with little additional compilation time overhead. Moreover, when we replace every declaration of `double` for an instance variable with `float`, we can improve the performance gain by 7.3% for the MolDyn benchmark in the JavaGrande benchmark suite.

Although we implemented our algorithm on IA-64, we can also apply our algorithm to a variety of architectures. **Table 1** shows various architectures and their instructions to which we can apply our instruction combining. For PowerPC [15], S/390 [16], and ARM [2], we can combine some load operations by using a load-multiple instruction. For IA-32 architectures and IBM’s network processor PowerNP [14], we can combine some 8-bit or 16-bit load operations into a 32-bit load, because we can access a 32-bit register per 8-bit or 16-bit (we call it *partial register read*) on these architectures. For IA-64, PowerPC, and the TMS320C6000 [33], we can combine shift and mask operations by using special bit-wise instructions (e.g. *extract* or *rlwinm*).

Table 1. Instruction candidates for our instruction combining

General-Purpose Processors	
IA-64	64-bit integer load, single- and double-precision pair-load, bit-wise operation (extract)
PowerPC	Load-multiple, constant load, bit-wise operations (<i>rlwinm</i> , ...)
IA-32	MMX/SSE instructions, Partial register read (e.g. <i>AL/AH</i>), memory operand
S/390	Load-multiple, constant load, memory operand
Embedded Processors	
ARM	Load-multiple
PowerNP	Partial register read, constant load
Embedded PowerPC (405, 440)	Load-multiple, constant load, bit-wise operations (<i>rlwinm</i> , ...)
TMS320C6000	Bit-wise operation (extract)

The following sections describe our approach, experimental results, related work, and concluding remarks.

2. OUR APPROACH

It is intuitive to put dependent instructions together because of their data dependence. However, it is not obvious to put independent ones together because an instruction can be moved across other instructions that have no true dependence on that instruction. **Figure 2** shows differences between our approach and the lazy code motion (LCM) algorithm [25]. Since the LCM algorithm does not consider the combinable region, it moves instructions independently as shown in Figure 2(a). In contrast, our approach moves the target instructions to the last point of the region where they are combinable and whose execution frequency is low, and then it combines them as shown in Figure 2(b).

Figure 3 shows two examples to explain our optimizations. Previous algorithms [1, 6, 27, 31] cannot optimize either example. Figure 3(a) is an example in which two 32-bit integer loads are combined. For IA-64, we can transform two 32-bit integer loads and two sign-extensions into a combination of a 64-bit integer load

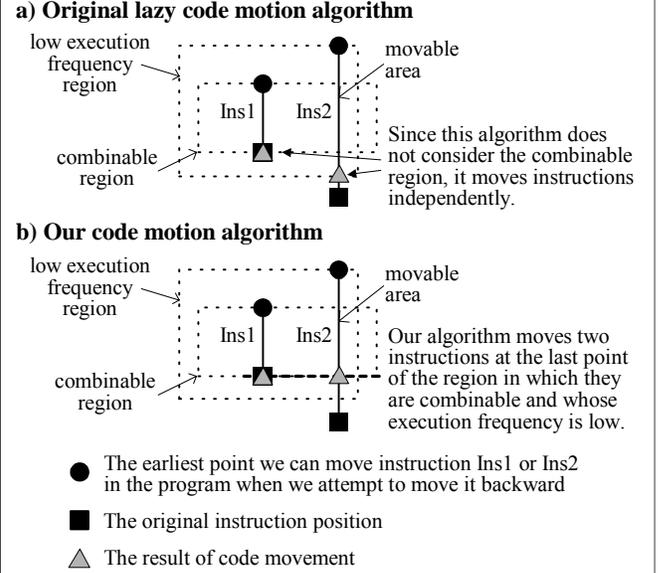


Figure 2. Our approach for independent instructions

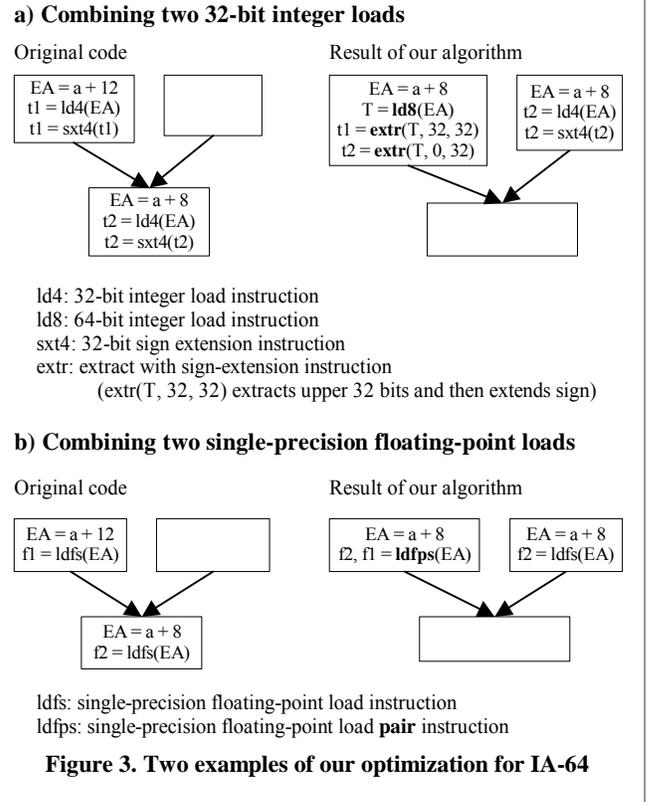


Figure 3. Two examples of our optimization for IA-64

and two “extract with sign-extension” instructions for each 32-bit value if their memory addresses are contiguous. As a result, we can get equal or better performance¹ along the left-hand path of Figure 3(a) than the previous algorithms. Since most of the programs

¹ The sign-extension instruction (*sxt4*) can be eliminated when we use the sign-extension elimination algorithm [21]. If both sign extensions can be eliminated, the performance of our approach will be equivalent to that of the previous approach.

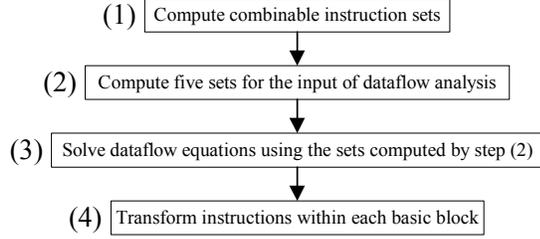


Figure 4. Flow diagram of our algorithm

available today are designed for the 32-bit architectures, 32-bit data types are still used frequently. For example, Java specifies “int” as a 32-bit data type [8]. Therefore, this optimization (Figure 3(a)) is quite effective for the 64-bit architectures.

Figure 3(b) is an example where two single-precision floating-point loads can be combined. IA-64 has pair-load instructions, *ldfps* and *ldfpd*, for single- and double-precisions, respectively, with which we can combine two floating-point loads that read contiguous memory locations. Our algorithm can take advantage of these instructions to improve the performance along the left-hand path in Figure 3(b). We note here that the target registers of a pair-load instruction must specify one odd- and one even-numbered register [17], but this restriction can be handled nicely by using a new register allocation approach based on preference-directed graph coloring [24].

In addition, our approach can combine instructions along the conditionally executed path. Both examples in Figure 3 offer an opportunity to apply instruction combining along the left-hand path, but not along the right-hand path. By applying our modified version of the LCM technique [25], we can optimize multiple instructions that are combinable along the conditionally executed path.

For either example in Figure 3, we need to take the memory alignment into account. After the optimization, the two contiguous memory locations need to be aligned at the 64-bit boundary, since they must be loaded as 64-bit data.

For dynamic compilers, it is important to use a fast algorithm and its efficient implementation to significantly reduce the compilation time for time consuming optimizations such as dataflow analyses. In particular, on a 64-bit architecture such as IA-64, a bit-vector implementation as we took is an attractive choice because of its longer word.

2.1 Our Algorithm

In this section, we describe a framework for putting the target instructions as close together as possible. **Figure 4** shows a flow diagram of the four steps of our core algorithm. We perform this algorithm on the intermediate language level. Note that our JIT compiler also performs traditional optimizations in other phases (such as copy propagation [28], dead store elimination [28], traditional PRE [25], null check optimization [20], scalar replacement [20], and sign-extension elimination [21]), though we do not describe them in Figure 4.

For **Step 1**, we compute the combinable instructions sets (we call them *groups*) in the input code. We pre-define combining patterns in the compiler. Inputs of a pattern are right-hand side expressions (RHSEs) of instructions. The output of a pattern is an instruction

```

for (each  $n \in$  all basic blocks){
  N-COMPG( $n$ ) = N-COMP( $n$ )
  for (each  $e \in$  N-COMP( $n$ )){
     $g$  = group of  $e$ 
    N-COMPG( $n$ ) += all instructions within  $g$ 
  }
  X-COMPG( $n$ ) = X-COMP( $n$ )
  for (each  $e \in$  X-COMP( $n$ )){
     $g$  = group of  $e$ 
    X-COMPG( $n$ ) += all instructions within  $g$ 
  }
}
  
```

Figure 5. Algorithm for computing N-COMP_G and X-COMP_G

sequence in which the inputs are combined. We include only the cases in which combining inputs produces equal or better performance. The compilation for a method takes the following three steps:

- (1) We collect candidates of the RHSEs of each combining pattern. For example, we collect all loads for the examples in Figure 3. If the RHSEs are identical, we treat them as the same candidate. For example, all “load[L1+8]” are treated as the same candidate regardless of content of L1. For each candidate, we also sum up the execution frequency of each position in the method.
- (2) We sort candidates based on the total execution frequency computed by (1) and limit the number of candidates to reduce the compilation time.
- (3) We compute the combinable instruction sets from the candidates of (2).

Next, we attach a group attribute, represented in bit-vector form, to each instruction. As we mentioned before, if the RHSEs are identical, we allocate the same bit for them. For example, suppose that there are five instructions in a given method, and the two instructions corresponding to bits 0 and 1 can be combined, and the two instructions corresponding to bits 2 and 4 can also be combined. The former two instructions share the same attribute of {11000}, while the latter two instructions share the same attribute of {00101}. The instruction corresponding to bit 3 cannot be combined with any instruction, so that instruction has the special empty attribute of {00000}. For now, let us assume the simplest case where one instruction is always included only within a single group. This assumption sufficient for the two examples in Figure 3, because they require that the two contiguous memory locations must be aligned at a 64-bit boundary (that is, each instruction is always included only within a single group). We will describe a solution in Section 2.1.1 when one instruction is included within multiple groups.

For **Step 2**, we compute five sets for the input of dataflow analysis. Our code motion algorithm is based on the Lazy Code Motion (LCM) algorithm [25], which originally has three sets TRANSP, N-COMP, and X-COMP as the inputs, and two sets N-INSERT and X-INSERT as the outputs. These five sets are defined as follows (N- and X- represent the entry and the exit, respectively):

TRANSP(n): the set of instructions that are located in the given method and which can be moved through basic block n .

N-COMP(n): the set of instructions that are located in basic block n and which can be moved to the entry point of the basic block.

- (a) Execute the Busy Code Motion algorithm [25]. Inputs are TRANSP, N-COMP, and X-COMP. Outputs are N-EARLIEST and X-EARLIEST.
- (b) Delayability Analysis:

$$\mathbf{N-DELAYED}(n) = \mathbf{N-EARLIEST}(n) + \prod_{m \in \text{Pred}(n)} \overline{\mathbf{X-COMP}_G(m)} \bullet \mathbf{X-DELAYED}(m)$$

$$\mathbf{X-DELAYED}(n) = \mathbf{X-EARLIEST}(n) + \mathbf{N-DELAYED}(n) \bullet \overline{\mathbf{N-COMP}_G(n)}$$
- (c) Computation of Latestness:

$$\mathbf{N-LATEST}(n) = \mathbf{N-DELAYED}(n) \bullet \mathbf{N-COMP}_G(n)$$

$$\mathbf{X-LATEST}(n) = \mathbf{X-DELAYED}(n) \bullet (\mathbf{X-COMP}_G(n) + \sum_{m \in \text{Succ}(n)} \overline{\mathbf{N-DELAYED}(m)})$$
- (d) Isolation Analysis:

$$\mathbf{N-ISOLATED}(n) = \mathbf{X-EARLIEST}(n) + \mathbf{X-ISOLATED}(n)$$

$$\mathbf{X-ISOLATED}(n) = \prod_{m \in \text{Succ}(n)} \mathbf{N-EARLIEST}(m) + \overline{\mathbf{N-COMP}_G(m)} \bullet \mathbf{N-ISOLATED}(m)$$
- (e) Insertion Point:

$$\mathbf{X-INSERT}(n) = (\mathbf{N-LATEST}(n) \bullet \overline{\mathbf{N-ISOLATED}(n)}) + (\mathbf{X-LATEST}(n) \bullet \overline{\mathbf{X-ISOLATED}(n)} \bullet \mathbf{TRANSP}(n))$$
- (f) Availability Analysis:

$$\mathbf{N-AVAIL}(n) = \prod_{m \in \text{Pred}(n)} \mathbf{X-AVAIL}(m)$$

$$\mathbf{X-AVAIL}(n) = \mathbf{X-INSERT}(n) + (\mathbf{N-AVAIL}(n) \bullet \mathbf{TRANSP}(n))$$

Figure 6. Algorithm of our Group-Sensitive Code Motion
(Bold text denotes our modifications and additions to the original LCM algorithm)

X-COMP(n): the set of instructions that are located in basic block n and which can be moved to the exit point of the basic block.

N-INSERT(n): the set of instructions that will be inserted at the entry point of the basic block n .

X-INSERT(n): the set of instructions that will be inserted at the exit point of the basic block n .

In addition, our algorithm requires two new sets as additional inputs: **N-COMP**_G and **X-COMP**_G (G denotes group). We define these two sets as follows:

N-COMP_G (n): the set of instructions whose forward movements should be stopped at the entry point of n for instruction combining.

X-COMP_G (n): the set of instructions whose forward movements should be stopped at the exit point of n for instruction combining.

We first compute the three sets: TRANSP, N-COMP, and X-COMP, and then compute the two new sets: N-COMP_G and X-COMP_G, using the algorithm of **Figure 5**. Note that we need to correctly find the barriers for moving a memory load to compute the TRANSP set. The barriers are the same as those of *scalar replacement* [7, 13, 22], which improves the accesses to non-local variables by replacing them with accesses to local variables.

For **Step 3**, we solve the dataflow equations using the five sets computed in Step 2 in order to compute insertion points and redundant regions. The LCM algorithm consists of two parts. The first part is the Busy Code Motion (BCM) [25], which moves an instruction backward if its execution count is not increased. The second part is the lazy code motion, which moves an instruction

forward in order to minimize the register pressure. Since we use the original BCM, the execution count of a moved instruction is the same as that of BCM. We modified the lazy code motion part (which moves instructions forward) to put the instructions in the same group as close together as possible, as shown in Figure 2(b). We call our approach Group-Sensitive Code Motion (GSCM). Our GSCM stops the forward motion of an instruction B at the point where it reaches one of the instructions in the group of B . By these modifications, we achieve the code motion as shown in Figure 2(b).

Figure 6 shows the details of our GSCM algorithm. Bold text denotes our modifications and additions to the LCM algorithm. First, the BCM algorithm produces two sets as output: N-EARLIEST and X-EARLIEST. They denote the earliest points to which an instruction can be moved backwards on the control flow graph. By modifying the steps (b) through (d) in Figure 6, the forward movement of an instruction B is stopped at the point where it reaches one of the instructions in the group of B .

The LCM algorithm first eliminates redundancies in every basic block, and then it performs global code motion to eliminate redundancies between basic blocks. In other words, it transforms the code in a program twice. As long as each instruction is independently optimized, this approach can be used. When some instructions are associated and optimized, it is efficient to transform code once in the last step. For that purpose, we modified Figure 6 Step (e) and added Step (f). The LCM algorithm computes two sets N-INSERT(n) and X-INSERT(n), but we combine them into X-INSERT(n) for local code transformation. Step (f) computes N-AVAIL and X-AVAIL, which denote sets of those instructions in the X-INSERT set that are available at the entry point and the exit point of each basic block, respectively.

```

/* Note: T[expr] has a temporary variable for the expr. */
inner = N-REACH(n);
for (each I from the first to the last instruction in block n){
  R = right-hand expression of I;
  inner = inner - all instructions by which
    movement is stopped by R;
  if (R ∈ inner){
    if (there is an R before I in this block && the result of R
      does not change in the meantime)
      Insert the code “T[R] = R” at the position;
    replace R at I with the temporary variable T[R];
  } else {
    g = group of R;
    if (there is an instruction in g before I in this block &&
      R can be moved to that instruction){
      Insert “the code C in which the instructions in g
        are combined” at the instruction;
      inner += all instructions in g;
      R is replaced with T[R];
    }
  }
  inner ∪= R;
  inner = inner - all instructions by which
    movement is stopped by the destination variable of I;
}

// Insert instructions included in X-INSERT(n)
ins = X-INSERT(n);
for (each e ∈ ins){
  e_g = group of e ∩ X-INSERT(n);
  if (instructions in e_g are combinable &&
    e_g - inner ≠ ∅){
    P = the end of block n;
    if (there is an instruction in e_g in block n &&
      the instruction can be moved to P)
      P = position of the instruction;
    Insert “the code C in which the instructions in e_g
      are combined” at P;
    inner += all instructions in e_g;
    ins = ins - all instructions in e_g;
  } else if (e ∉ inner){
    Insert “T[e] = e” at the end of block n;
  } else if (there is an instruction e in the block n that
    can be moved to the end of block n)
    Insert “T[e] = e” at the instruction e;
}

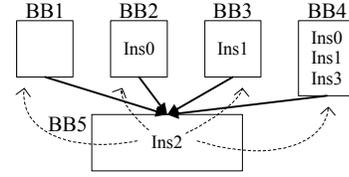
```

Figure 7. Code transformation algorithm in block n

For **Step 4**, we transform the code for each basic block using X-INSERT and N-AVAIL as computed in Step 3. **Figure 7** shows the algorithm for transforming the code in basic block n . This algorithm is roughly divided into two parts. The first part scans each instruction in the block n in order to perform instruction combining in the block n using N-AVAIL(n). The second part inserts instructions for X-INSERT(n) into the block n .

2.1.1 One Instruction is Included within Multiple Groups

This section describes a solution when one instruction is included within multiple groups. Since the two examples in Figure 3 require that the two contiguous memory locations must be aligned at a 64-



Ins0 and Ins1 are combinable (GROUP1)
 Ins1 and Ins2 are combinable (GROUP2)
 Ins2 and Ins3 are combinable (GROUP3)

(a) Set up only one group for one instruction

Corresponding group	
Ins0: {1100}	(GROUP1)
Ins1: {1100}	(GROUP1)
Ins2: {0011}	(GROUP3)
Ins3: {0011}	(GROUP3)

(b) Set up multiple groups for one instruction

Corresponding group	
Ins0: {1100}	(GROUP1)
Ins1: {1100}, {0110}	(GROUP1,2)
Ins2: {0110}, {0011}	(GROUP2,3)
Ins3: {0011}	(GROUP3)

Figure 8. Example in which some instructions are included in multiple groups

bit boundary, each instruction is always included only within a single group. However, in general, it is more common that one instruction might be included within multiple groups.

For example, we assume that GROUP1 (Ins0 and Ins1), GROUP2 (Ins1 and Ins2), and GROUP3 (Ins2 and Ins3) can be combined in **Figure 8**. The easiest solution is to set up only one group for each instruction as shown in (a). In this example, we exclude GROUP2. Using this solution, Ins2 in BB5 is moved to BB1 through BB4. Then the instructions of BB4, both in GROUP1 and GROUP3, can be combined. However, this solution misses the opportunity for combining Ins1 and Ins2. In this example, we cannot combine them in BB3 and BB4.

Therefore, we allow the same instruction to be included within multiple groups. In this case, each group contains a pair of combinable instructions. First, we sort the groups that include the same instruction, based on the effectiveness of combining for each group. We start combining based on that order. In Figure 8, we assume that the order of priority is GROUP2, GROUP1, and GROUP3. In this case, GROUP2 is transformed first. Next, for GROUP1, we attempt to combine Ins0 (which has not been transformed) with the combined result of GROUP2. If it cannot be combined, the original instruction is left alone. For GROUP3, we attempt to combine Ins3 in the same way.

Figure 9 shows an actual example corresponding to Figure 8. For a 32-bit constant load on the PowerPC, we generally need two instructions to set the upper and the lower 16-bit values. However, we can save one instruction by using an arithmetic instruction.

Figure 9(b) shows the results after transforming the instructions in BB4 in the order of GROUP2, GROUP1, and GROUP3. We first combine Ins1 and Ins2. Next, we transform Ins0 because we can compute the result of Ins0 by using the new Ins1. We can also transform Ins3 by using the new Ins2. If Ins0 or Ins3 cannot be combined, that instruction will be left as it is.

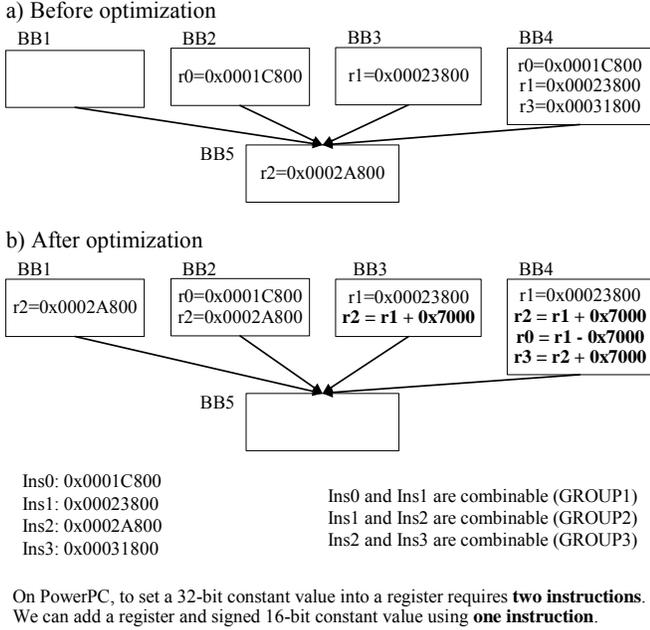


Figure 9. Constant load optimization for PowerPC

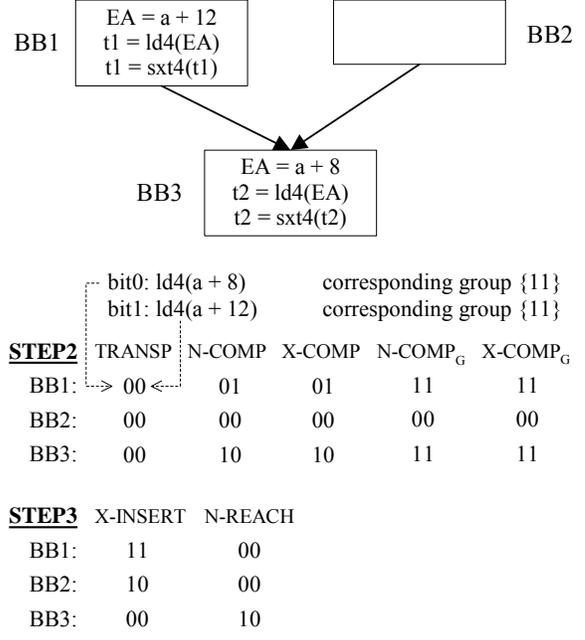


Figure 11. Applying Steps 2 and 3 to Figure 3(a)

a) For Figure 3(a):
 Output instruction sequence in which two instruction sequences $ld4(a+8)$ and $ld4(a+12)$ are combined: { $EA=a+8$; $T=ld8(EA)$; $T1=extr(T, 32, 32)$; $T1=sxt4(T1)$; $T2=extr(T, 0, 32)$; $T2=sxt4(T2)$; }

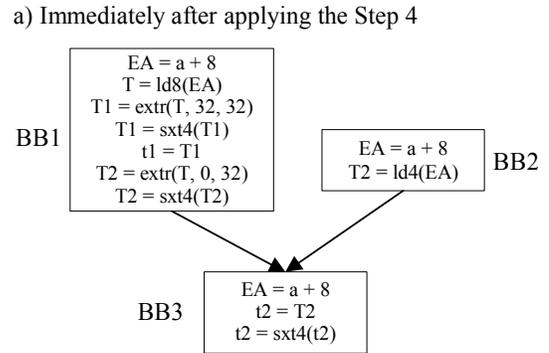
b) For Figure 3(b):
 Output instruction sequence in which two instruction sequences $ldfs(a+8)$ and $ldfs(a+12)$ are combined: { $EA=a+8$; $T2, T1=ldf8(EA)$; }

Figure 10. Step 1 for Figure 3(a) and (b)

2.2 Two Examples from Figure 3

In this section, we demonstrate how our algorithm transforms the two examples in Figure 3. Figure 10 shows the output instruction sequences for the examples of Figure 3 (a) and (b). We note here that we deliberately generate a redundant sign-extension ($sxt4$) in the instruction sequence in Figure 10(a). Since the previous extract instruction ($extr$) also performs a sign-extension, the sign-extension instruction ($sxt4$) is obviously redundant but necessary for effectively optimizing the sign-extensions. We will explain this in more detail using Figure 12.

Figure 11 shows the results after applying Steps 2 and 3 to Figure 3(a). As regards Figure 3(b), if $ld4$ is read as $ldfs$, the same result can be obtained. From Step 2, the five sets, TRANSP, N-COMP, X-COMP, N-COMP_G, and X-COMP_G, will be obtained as shown in STEP2 of Figure 11. As the next step, by solving the dataflow equations as shown in Figure 6 with the five sets computed in Step 2, two sets, X-INSERT and N-AVAIL, will be obtained as shown in STEP3 of Figure 11. When we perform the LCM algorithm, the result of X-INSERT and N-AVAIL will be “{00}” for every basic block.



b) Applying copy propagation and dead store elimination to (a)

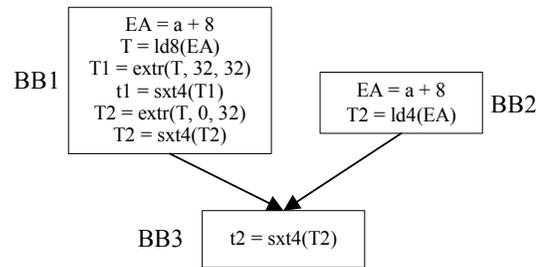
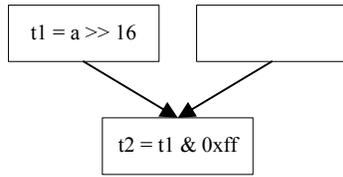


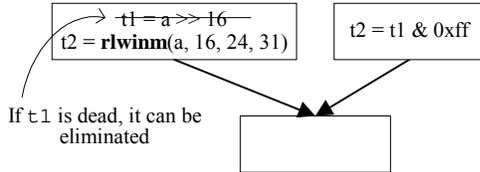
Figure 12. Applying Step 4 to Figure 11

Figure 12(a) shows the transformation result immediately after applying Step 4 (that is, the output of our optimization) with the results (X-INSERT and N-AVAIL) computed by Step 3. In Step 4, the instruction sequence shown in Figure 10(a) is used.

a) Before optimization



b) After optimization



rlwinm: Rotate Left Word Immediate Then AND with Mask instruction (rlwinm(a, 16, 24, 31) rotates left 'a' 16 bits and then masks between the 24th and 31st bits)

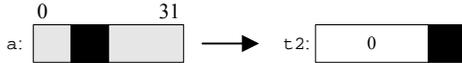


Figure 13. Example where dependent instructions are optimized on PowerPC

Next we apply both copy propagation and dead store elimination (Figure 12(b)). As we mentioned before, we deliberately generate a redundant sign-extension to effectively optimize sign-extensions. In this example, “*sxt4(T2)*” in BB3 becomes partially redundant because it appears in BB1. Thus, the original PRE technique can move it from BB3 to BB2. Finally, this example can be transformed to Figure 3(a) by performing several traditional optimizations, such as a sign-extension elimination [21], copy propagation, and dead store elimination. Regarding Figure 3(b), we can obtain the result of our approach in the same way as in (a) by using the code sequence in Figure 10(b).

2.3 Other Optimizations Using Our Algorithm

In the following section, we describe some optimizations that can be made by performing additional transactions with the algorithm described in Section 2.1. Section 2.3.1 describes instruction combining for dependent instructions. Section 2.3.2 describes a combination of loop transformations and instruction combining.

2.3.1 Optimizing Dependent Instructions

Although our approach is characterized by optimizing independent instructions, it is also possible to optimize two dependent instructions. Figure 13 shows an example on the PowerPC. By using an `rlwinm` instruction, we can eliminate one instruction along the left-hand path as shown in (b) as long as `t1` is dead.

Note that we need to consider the order of instructions if two dependent instructions are optimized. Here, we call “the instruction that must proceed” *FIRST*, and we call “the instruction that must follow” *SECOND*. In Figure 13, *FIRST* is “`t1 = a >> 16`” and *SECOND* is “`t2 = t1 & 0xff`”. In order to avoid a situation in which we apply an incorrect optimization in the reverse order (that is,

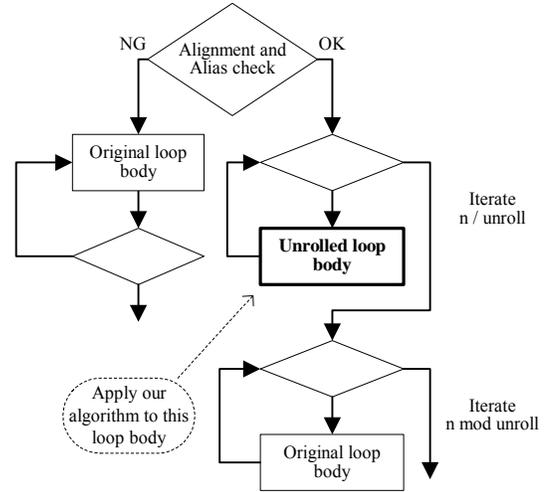
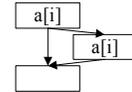


Figure 14. Combination with loop transformations

a) Original program

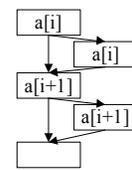
```
float a[ ];
for (i = 0; i < n; i++)
    if (max < a[i]) max = a[i];
```

Memory loads in the loop body



b) After loop transformations

```
for (i = 0; i < n-1; i+=2) {
    if (max < a[i]) max = a[i];
    if (max < a[i+1]) max = a[i+1];
}
if ((n & 1) == 1) {
    if (max < a[n-1]) max = a[n-1];
}
```



c) After our optimization

```
for (i = 0; i < n-1; i+=2) {
    T1, T2 = ldtps(a[i]);
    if (max < T1) max = T1;
    if (max < T2) max = T2;
}
if ((n & 1) == 1) {
    T = a[n-1];
    if (max < T) max = T;
}
```

Our approach simultaneously performs both scalar replacement and instruction combining

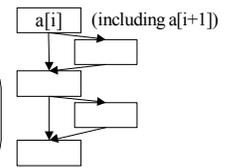


Figure 15. Combination of our approach and loop transformations

SECOND comes before *FIRST*), we assume that there is a barrier for *FIRST* immediately before *FIRST*.

2.3.2 Combination with Loop Transformations

We can also optimize array accesses between loop iterations in combination with loop transformations. Davidson *et al.* [6] described two loop transformations to that end. These transformations first perform loop versioning to create two versions of the loop using both alignment and alias checks as shown in Figure 14. Next, loop unrolling expands the loop body of the safe version. If we perform these two loop transformations, we can combine array accesses between loop iterations by applying the GSCM algorithm to the unrolled loop body.

Applying both our approach and loop transformations together can generate even more highly optimized code than previous ap-

proaches [1, 6, 27, 31], which combine array accesses between loop iterations by unrolling loops, because of the code motion of our approach. **Figure 15** shows an example. Previous approaches cannot combine the two memory loads, $a[i]$ and $a[i+1]$ in the loop body of **Figure 15(b)**, because they do not exploit global code motion. In contrast, our approach can combine them as shown in **Figure 15(c)** by using the GSCM algorithm. Moreover, because our approach can simultaneously perform both scalar replacement and instruction combining in one phase, we can reduce four memory loads to one in the loop body in **Figure 15(b)**. Since our approach globally optimizes the whole method, we can also reduce two memory loads to one after loop ($a[n-1]$) in the same phase.

3. EXPERIMENTS

We chose the SPECjvm98 benchmark suite [32] for evaluating our optimizations in the IBM Developers Kit for IA-64, Java Technology Edition, Version 1.4. The basic GC algorithm is based on a mark and sweep algorithm [3]. We ran each benchmark program from the command line with the problem size of 100, and with the initial and maximum heap sizes of 96 MB. Each benchmark program was executed 10 times consecutively for each independent run. We implemented two optimizations in **Figure 3** using the GSCM algorithm in the IBM Java JIT Compiler. As we explained in **Section 2**, previous algorithms [1, 6, 27, 31] cannot handle these optimizations. All of the experiments were conducted on an IBM IntelliStation Z Pro model 6894-12X (two Intel Itanium 800 MHz processors with 2 GB of RAM), running under Windows.

3.1 Performance Improvement

We measured the following two versions to evaluate our approach. Both versions performed two optimizations in **Figure 3**, but we have not implemented yet either combining for double-precision floating-point loads or other optimizations described in **Section 2.3**.

- **Baseline:** Perform instruction combining with the original LCM algorithm [25]. The other optimizations, including copy propagation [28], dead store elimination [28], traditional PRE [25], null check optimization [20], scalar replacement [22], and sign-extension elimination [21], are enabled.
- **Our approach:** Perform instruction combining with our GSCM algorithm. The other optimizations described in the Baseline are enabled.

Previous approaches [28, 29] for combining dependent instructions cannot optimize the examples given in **Figure 3** since they are independent instructions. Without applying loop transformations, previous approaches [1, 6, 27, 31] for combining independent instructions are equivalent to our baseline, and thus it is fair to examine the performance improvement by our algorithm over the baseline. This is because the dynamic compiler has budget limitations, particularly for compilation time and thus loop transformations such as loop unrolling are usually avoided to limit the code expansion. It is interesting to see how the performance will be improved by instruction combining when loop transformations such as loop unrolling are performed before combining, but that is beyond the scope of our paper.

Figure 16 shows the performance improvement in the best time over the baseline for SPECjvm98. Because the SPECjvm98 metric is calculated from the best runs, we took the best time from repetitive runs for the comparison. Thus, these results do not include compilation time. Our experimental results show that our algorithm

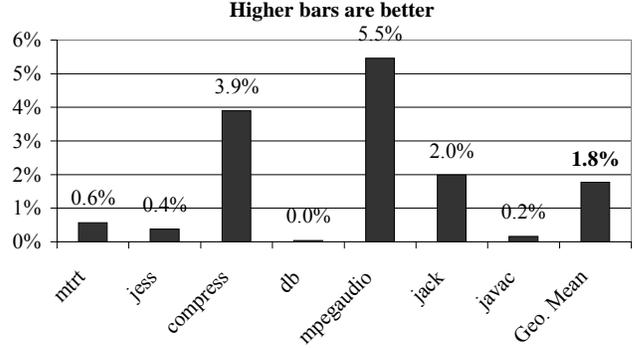


Figure 16. Performance improvement in the best time for SPECjvm98 over the baseline

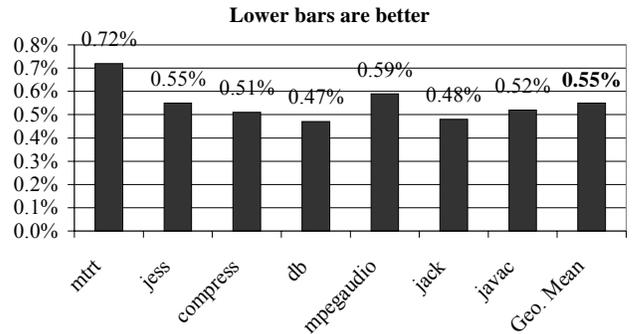


Figure 17. Compilation time increases for SPECjvm98 over the baseline

improves the geometric mean (maximum) performance by 1.8% (5.5%) over the baseline. Our approach is particularly effective for *compress*, *mpegaudio*, and *jack*. We find that combining integer loads (**Figure 3(a)**) for instance variables is quite effective for *compress* and *jack*, and that combining floating-point loads (**Figure 3(b)**) for array accesses whose indices are constant is similarly effective for *mpegaudio*. Therefore, our approach is effective even for those instructions that are not related to any loop induction variable.

3.2 JIT Compilation Time

This section describes how our approach affects the JIT compilation time. For **Figure 17**, we measured the breakdown of the JIT compilation time during 10 repetitive runs by using a trace tool on IA-64. In summary, our approach increased the total compilation time by 0.55% (0.72%) for the geometric mean (maximum), while achieving significant performance improvement as shown in **Figure 16**. In addition, our approach caused little increase (0.47% to 0.72%) of the compilation time regardless of the benchmark.

3.3 Discussions

There are three categories of memory loads we can potentially combine for IA-64. They are integer loads, single-precision floating-point loads, and double-precision floating-point loads. For integer loads, the gain was 3.9% for *compress* and 2.0% for *jack*. For single-precision floating-point loads, the gain was 5.5% for *mpegaudio*.

For *double*-precision floating-point loads, we can expect a larger gain since they are more often used than single-precision floating-point loads, but we have not fully implemented combining them yet. In order to support this, we would need to modify the JVM. This is because any operand of a paired load instruction for double-precision (the *ldfpd* instruction) on IA-64 must be aligned at a 128-bit boundary, but the current JVM aligns objects at the 64-bit boundaries. In order to estimate the effectiveness of combining *double*-precision floating-point loads, we performed an experiment replacing every declaration for `double` of an instance variable with `float` in the MolDyn (Molecular Dynamics simulation) benchmark in the JavaGrande benchmark suite. The result was that our algorithm improved this benchmark by 7.3% over our baseline (using the LCM algorithm).

Finally, for any of the three categories, we could further enhance performance with combining, if we additionally supported the loop transformations such as loop unrolling described in Section 2.3.2 in our JIT compiler. Once we fully support all the features mentioned above, we will be able to achieve greater performance improvements with combining.

4. RELATED WORK

Previous approaches for instruction combining can be classified into two families. The first family combines dependent instructions [28, 29]. This family moves a single instruction backward to the location immediately after another instruction that has a true dependence on that instruction, and then it combines these two instructions. This relies on data dependence for moving an instruction, and thus it cannot combine independent instructions. It performs global code motion, but it cannot combine instructions along the conditionally executed path as in the example in Figure 13.

The second family combines independent instructions. This family combines array accesses between loop iterations by unrolling the loop [1, 6, 27, 31]. This combines the independent instructions for array accesses related to a loop induction variable, but it does not perform global code motion. Because this approach is limited to a loop whose body consists of a single basic block, it cannot optimize memory accesses included in a complex loop as in the example in Figure 15.

In contrast, our approach combines both dependent instructions and independent instructions. It can also combine instructions along a conditionally executed path by using global code motion (as shown in Figure 3 and Figure 13). Moreover, it is not limited to memory accesses, but it can also be applied to other instructions. We already discussed some variations of our approach in Section 2.3.

Recently, Nandivada *et al.* proposed an approach that reorders the variables in the spill area for maximizing a chance of combining consecutive spill codes into a load- or a store-multiple instruction after a register allocation. We can use a similar approach for further performance improvement. Let us note here about register constraints. Because load- and store-multiple instructions require specific numbered registers, their approach needs to generate register swapping code. In our optimization, a pair-load instruction also requires specific numbered (odd and even) registers as mentioned in Section 2. We solve this register constraint problem by using preference-directed graph coloring [24] after instruction

combining is performed, and thereby we systematically reduce the need to generate the register swapping code.

Strength reduction is similar to instruction combining, but it converts a rather expensive instruction, such as a multiplication or a division, into a less expensive one, such as an addition or a subtraction. There are some strength reduction algorithms using a partial redundancy elimination (PRE) technique [11, 23, 26]. Basically, they move a single instruction backward to the location immediately after another instruction that has a true dependence on that instruction in order to determine whether its complexity can be reduced. In other words, these approaches only optimize the dependent instructions but not independent ones.

5. CONCLUSION

In this paper, we propose a new algorithm for instruction combining by using global code motion in order to apply instruction combining in a wider region. Our group-sensitive code motion (GSCM) algorithm is based on the Lazy Code Motion (LCM) algorithm [25]. We modified it to search for more potential candidates and to put the target instructions together for combining in a wider region. By using this code motion algorithm, we can optimize both dependent instructions and independent ones. When we use instruction combining to coalesce memory accesses, we can reduce the memory traffic by combining narrow memory references with contiguous addresses into a wider reference for taking advantage of a wide-bus architecture. We implemented two optimizations for coalescing memory access, one combining two 32-bit integer loads and the other combining two single-precision floating-point loads, using our algorithm in the IBM Java JIT compiler for IA-64, and evaluated these optimizations by measuring the SPECjvm98 benchmark suite. In our experiment, we can improve the maximum performance by 5.5% with little additional compilation time overhead. Moreover, when we replace every declaration for `double` of instance variables with `float`, we can improve the performance gain by 7.3% for the MolDyn benchmark in the JavaGrande benchmark suite. Our approach can be applied to a variety of architectures and to programming languages besides Java.

6. ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their helpful comments.

7. REFERENCES

- [1] M.J. Alexander, M.W. Bailey, B.R. Childers, J.W. Davidson, and S. Jinturkar. Memory Bandwidth Optimizations, In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, pp. 466-475, 1993.
- [2] ARM, "ARM Instruction Set Quick Reference Card", http://www.arm.com/pdfs/QRC_ARM.pdf
- [3] K. Barabash, Y. Ossia, and E. Petrank. Mostly Concurrent Garbage Collection Revisited, *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 255-268, 2003
- [4] R. Bernstein. Multiplication by integer constants, *Software - Practice and Experience*, Vol. 16, No. 7, pp. 641-652, 1986.
- [5] P. Briggs and T. Harvey. Multiplication by integer constants, <http://citeseer.nj.nec.com/briggs94multiplication.html>
- [6] J.W. Davidson and S. Jinturkar. Memory Access Coalescing: A technique for Eliminating Redundant memory Accesses, *Con-*

- ference on Programming Language Design and Implementation, pp. 186-195, 1994.
- [7] S.J. Fink, K. Knobe, and V. Sarkar, "Unified Analysis of Array and Object References in Strongly Typed Languages" *Static Analysis Symposium*, pp.155-174, 2000.
- [8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*, Addison-Wesley Publishing Co., Reading, 1996.
- [9] T. Granlund and P.L. Montgomery. Division by invariant integers using multiplication, *Conference on Programming Language Design and Implementation*, pp. 61-72, 1994.
- [10] R. Gupta, D.A. Berson, J.Z. Fang. Path Profile Guided Partial Redundancy Elimination Using Speculation, In *IEEE Conference on Computer Languages*, 1998.
- [11] M. Hailperin. Cost-optimal code motion, *Transactions on Programming Languages and Systems*, Vol. 20, No. 6, pp. 1297-1322, 1998.
- [12] R.N. Horspool and H.C. Ho. Partial redundancy elimination driven by a cost-benefit analysis, *8th Israeli Conference on Computer Systems and Software Engineering*, pp. 111-118, 1997.
- [13] A.L. Hosking, N. Nystrom, D. Whitlock, Q. Cutts, and A. Diwan. Partial redundancy elimination for access path expressions, *Software-Practice and Experience*, Vol. 31, No. 6, pp. 577-600, 2001.
- [14] IBM Corp.: PowerNP network processors. http://www-3.ibm.com/chips/products/wired/products/network_processors.html
- [15] IBM Corp.: PowerPC Homepage, <http://www-3.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC>
- [16] IBM Corp.: z/Architecture Principles of Operation PDF files, <http://www-1.ibm.com/servers/s390/os390/bkserv/vmpdf/zarchpops.html>
- [17] Intel Corp.: Itanium Architecture - Manuals. <http://www.intel.com/design/itanium/manuals.htm>
- [18] Intel Corp.: Intel Itanium Processor Reference Manual for Software Optimization. <http://www.intel.com/design/itanium/downloads/245474.htm>
- [19] Intel Corp.: IA-32 Intel Architecture Optimization Reference Manual. <http://www.intel.com/design/Pentium4/manuals/248966.htm>
- [20] M. Kawahito, H. Komatsu, and T. Nakatani. Effective Null Pointer Check Elimination Utilizing Hardware Trap, *Conference on Architectural Support for Programming Language and Operating Systems*, pp. 139-149, 2000.
- [21] M. Kawahito, H. Komatsu, and T. Nakatani. Effective Sign Extension Elimination, *Conference on Programming Language Design and Implementation*, pp. 187-198, 2002.
- [22] M. Kawahito, H. Komatsu, and T. Nakatani. Partial redundancy elimination for access expressions by speculative code motion, To appear, *Software: Practice and Experience*, 2004
- [23] R. Kennedy, F.C. Chow, P. Dahl, S. Liu, R. Lo, and M. Streich. Strength Reduction via SSAPRE, *Computational Complexity*, pp. 144-158, 1998
- [24] A. Koseki, H. Komatsu, and T. Nakatani. Preference-directed graph coloring, *Conference on Programming Language Design and Implementation*, pp. 33-44, 2002.
- [25] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, Vol. 17, No. 5, pp. 777-802, 1995.
- [26] J. Knoop, O. Rüthing, and B. Steffen. Lazy Strength Reduction. *Journal of Programming Languages*, Vol. 1, No. 1, pp. 71-91, 1993.
- [27] S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets, *Conference on Programming Language Design and Implementation*, pp. 145-156, 2000.
- [28] S.S. Muchnick. *Advanced compiler design and implementation*, Morgan Kaufmann Publishers, Inc., 1997.
- [29] T. Nakatani and K. Ebcioğlu. "Combining" as a Compilation Technique for VLIW Architectures, *International Workshop on Microprogramming and Microarchitecture*, pp. 43-55, 1989
- [30] V.K. Nandivada and J. Palsberg. Efficient spill code for SDRAM, *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, San Jose, California, October 2003.
- [31] J. Shin, J. Chame, and M.W. Hall. Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures, *Conference on Parallel Architectures and Compilation Techniques*, pp. 45-55, 2002.
- [32] Standard Performance Evaluation Corp. "SPEC JVM98 Benchmarks," <http://www.spec.org/osg/jvm98/>
- [33] Texas Instruments, "TMS320C6000 CPU and instruction set reference guide", Lit. Num. SPRU189F, <http://www.tij.co.jp/jsc/docs/dsps/support/download/c6000/c6000pdf/spru189f.pdf>

8. APPENDIX

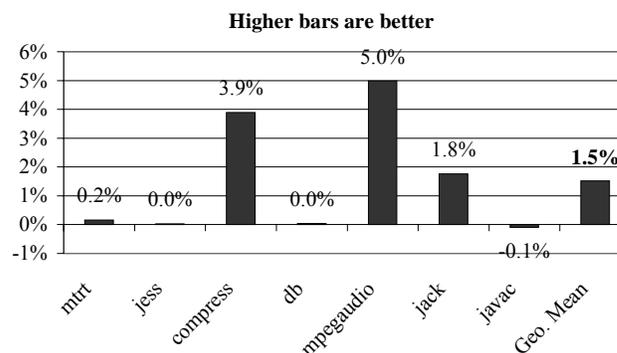


Figure 18. Performance improvement for the overall time for SPECjvm98 over the baseline

Although we took the best times in Figure 16 to conform to the SPECjvm98 metric, it is interesting to compare the overall times, which include the compilation times and GC times for 10 repetitive runs. Figure 18 shows the performance improvements for the overall times over the baseline. Results are slightly worse than in Figure 16 because of the additional compilation time overhead.