Compiler Transformations for Effectively Exploiting a Zero Overhead Loop Buffer[*]

Gang-Ryung Uh, Yuhong Wang[‡],
David Whalley[§],
Sanjay Jinturkar[¶],
Yunheung Paek[‖],
Vincent Cao[**], Chris Burns[**]

1910 University Drive, Boise, ID 83725

SUMMARY

A Zero Overhead Loop Buffer (ZOLB) is an architectural feature that is commonly found in DSP processors. This buffer can be viewed as a compiler managed cache that contains a sequence of instructions that will be executed a specified number of times without incurring any loop overhead. Unlike loop unrolling, a loop buffer can be used to minimize loop overhead without the penalty of increasing code size. In addition, a ZOLB requires relatively little space and power, which are both important considerations for

[*]Correspondence to: Gang-Ryung Uh, Computer Science Department of Boise State University, 1910 University Drive, Boise, ID 83725, U.S.A., uh@cs.boisestate.edu
[‡]LA Times, E-mail: Yuhong.Wang@latimes.com
[§]CS Dept of Florida State Univ., E-mail: whalley@cs.fsu.edu
[¶]Sandbridge Technologies, E-mail: Sanjay.Jinturkar@sandbridgetech.com
[‖]Seoul National University, E-mail : ypaek@ee.snu.ac.kr
[**]Agere Systems, E-mail: {vpcao,cpburns}@agere.com

Contract/grant sponsor: NASA; contract/grant number: 2004 Idaho EPSCoR grant
Contract/grant sponsor: NSF; contract/grant number: CCR-9904943
Contract/grant sponsor: NSF; contract/grant number: EIA-0072043
Contract/grant sponsor: NSF; contract/grant number: CCR-0208892
Contract/grant sponsor: KOSEF; contract/grant number: R08-2003-000-10781-0
Contract/grant sponsor: University IT Research Center Project; contract/grant number:

most DSP applications. This paper describes strategies for generating code to effectively use a ZOLB. We have found that many common code improving transformations used by optimizing compilers on conventional architectures can be easily used to (1) allow more loops to be placed in a ZOLB, (2) further reduce loop overhead of the loops placed in a ZOLB, and (3) avoid redundant loading of ZOLB loops. The results given in this paper demonstrate that this architectural feature can often be exploited with substantial improvements in execution time and slight reductions in code size for various signal processing applications.

KEY WORDS: *Compiler*; *Optimization*; *DSP*; *ZOLB*

1. INTRODUCTION

The common features of signal processing applications are intense numerical computations with hard real-time constraints. Digital signal processors are special purpose embedded processors to support such applications using less energy and small amount of memory. Thus, DSPs typically support heterogeneous multiple register files, that are placed in an arbitrary datapath, to reduce the processor size and irregular instruction sets optimized for instruction length.

Programmability and reconfigurability are important market requirements for DSPs along with performance. The DSP should be easily programmed to customize various feature sets. In addition, the DSP should be rapidly reconfigured to support frequently varying industry standards. In order to meet these two requirements, DSPs commonly support HLL (High-Level Language) compilers.

However, the irregularities, which are both present in microarchitectures and instructions sets of DSPs, make compiler code generation extremely difficult and challenging. Code generated by a compiler for applications written in a high level language typically runs significantly slower than hand-crafted DSP code unless target specific compiler transformations that are specially tailored and tuned for a given DSP are applied.

This paper describes a set of novel compiler optimization strategies that effectively improve the `C` compiler code generation quality. In particular, this paper presents in detail how we crafted/tailored a set of widely known conventional loop optimizations and interprocedural analysis to exploit the ZOLB (Zero Overhead Loop Buffer) that is available on Lucent DSP 16000 [3].

1.1. Zero Overhead Loop Buffer as an Architectural Feature of DSPs

Target signal processing applications require extensive arithmetic computation. For example, consider the following calculations typically found in communication and image processing.

$$
\begin{aligned}
FIR: \quad y_k &= \sum_{n=0}^{N} b_n x_{k-n} \\
FFT: \quad y_k &= \sum_{j=0}^{N-1} w^{jk} x_j \quad , where\ w = e^{\frac{-2i\pi}{N}} \\
2D-DCT: \quad F(u,v) &= \frac{1}{N^2} \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} f(m,n) cos[\frac{(2m+1)u\pi}{2N}] cos[\frac{(2n+1)v\pi}{2N}]
\end{aligned}
$$

The main computational engines (or kernels) of these algorithms can be easily programmed into tight small loops, and a large percentage of the execution time will be spent in the innermost loops [1]. Without any software and/or hardware support, the execution of kernels for the aforementioned algorithms would incur significant loop overhead [†], and stringent industry real-time constraints could be hardly met with current microprocessor fabrication technology.

In order to reduce loop incurred overhead, a Zero Overhead Loop Buffer (ZOLB) is commonly supported as an architectural feature in DSP processors.[‡] A ZOLB is a buffer that can contain a fixed number of instructions to be executed a specified number of times under program control. This buffer can be used to increase the speed of applications with no increase in code size and often with reduced power consumption since a ZOLB is significantly smaller than ROM and this buffer is placed on a chip.

Since the ZOLB is placed on a chip, instructions can be fetched faster from the buffer than from the conventional instruction memory. Thus, the same memory bus used to fetch instructions can be used to access data when certain registers are dereferenced. Thus, memory bus contention can be reduced when instructions are fetched from a ZOLB. Due to addressing complications, transfers of control instructions are not typically allowed in such buffers. Therefore, a compiler or assembly writer attempts to execute many of the innermost loops of programs from this buffer. A ZOLB can be viewed as a compiler (software) controlled cache since special instructions are used to load other instructions into it.

In order to improve the code generation quality of the Lucent DSP 16000 `C` compiler, we have designed and implemented compiler optimization strategies for exploiting the ZOLB that is available on the Lucent DSP16000 architecture [3]. Note that the proposed techniques could

[†]This overhead is due to the increment and branch instructions to initiate a new iteration of a loop.
[‡]The ZOLBs currently available in TI, ADI, and Lucent DSP processors are discussed in [5].

also be applied to several other DSP architectures with ZOLBs. We tested our implementation to demonstrate the effectiveness of the techniques for various Lucent wireless communication applications. We believe that these strategies have the potential for being readily adopted by compiler writers for DSP processors since they rely on the use of traditional compiler code improving transformations and data flow analysis techniques.

Figure 1 presents an overview of the compilation process that we used to generate and improve code for this architecture. Code is generated using a GNU C compiler retargeted to the DSP16000 [4]. Conventional code improving transformations in this C compiler are applied and assembly files are generated. Finally, the generated code is then processed by the optimizer that we developed. This optimizer performs a number of code improving transformations including those that exploit the ZOLB on this architecture.

There are advantages of attempting to exploit a ZOLB using this approach. First, the exact number of instructions in a loop will be known after code generation, which will ensure that the maximum number of instructions that can be contained in the ZOLB is not exceeded. While performing these transformations after code generation sometimes resulted in more complicated algorithms, the optimizer was able to apply transformations more frequently since it did not have to rely on conservative heuristics concerning the ratio of intermediate operations to machine instructions. Second, interprocedural analysis and transformations also proved to be valuable in exploiting a ZOLB, as will be shown later in this paper.

1.2. Organization of the Paper

This paper is organized as follows. In Section 2, detailed architectural features of the DSP 16000 ZOLB and instructions to manipulate the buffer will be explained. Section 3 will introduce a set

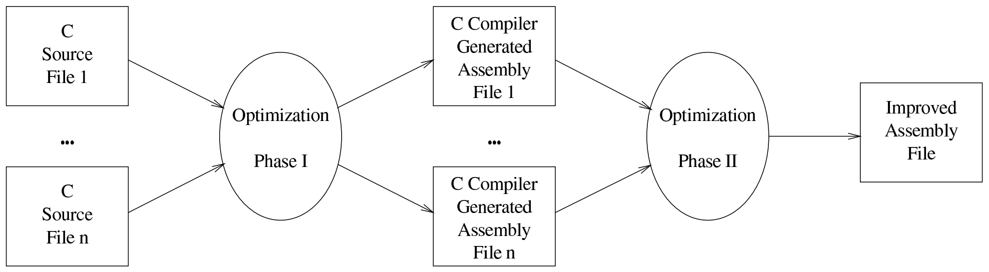


Figure 1. Overview of the Compilation Process for the DSP16000

of compiler optimization techniques, which perform transformations so that these loops could be placed in the ZOLB, which otherwise would not be possible. Sections 4 and 5 introduce a set of additional transformations to futher reduce loop overhead by carefully crafting the conventional loop transformations and the interprocedural flow analysis for exploiting the DSP16000 ZOLB. In Section 6, the order of transformations, which was carefully tuned to produce the desired results, will be described and benchmark performance results will be presented. Other software and hardware techniques to reduce loop overhead are described in Section 7. Finally, the conclusion from our experiments will be made in Section 8.

2. USING THE DSP16000 ZOLB

The target architecture for which we generated code was the DSP16000 developed at Lucent Technologies. This architecture contains a ZOLB that can hold up to 31 instructions. Two special instructions, the `do` and the `redo`, are used to control the ZOLB on the DSP16000 [7]. Figure 2(a) shows the assembly syntax for using the `do` instruction, which specifies that the n

```
...
do k {
instruction 1
...
instruction n
}
...
```

(a) Assembly Syntax for Using the do Instruction

```
...
redo k
...
```

(b) Assembly Syntax for Using the redo Instruction

Figure 2. DSP16000 Assembly Syntax for Using the ZOLB

instructions enclosed between the curly braces are to be executed k times. The actual encoding of the `do` instruction includes a value of n, which can range from 1 to 31, indicating the number of instructions following the `do` instruction that are to be placed in the ZOLB. The value k is also included in the encoding of the `do` instruction and represents the number of iterations associated with an innermost loop placed in the ZOLB. When k is a compile-time constant less than 128, it may be specified as an immediate value since it will be small enough to be encoded into the instruction. Otherwise, the value of zero is encoded and the number of times the instructions in the ZOLB will be executed is obtained from the `cloop` register, which can be assigned a value in a separate instruction. The first iteration results in the instructions enclosed between the curly braces being fetched from the memory system, executed, and loaded into the ZOLB. The remaining k-1 iterations are executed from the ZOLB. The `redo` instruction shown in Figure 2(b) is similar to the `do` instruction, except that the current contents of the ZOLB are executed k times.

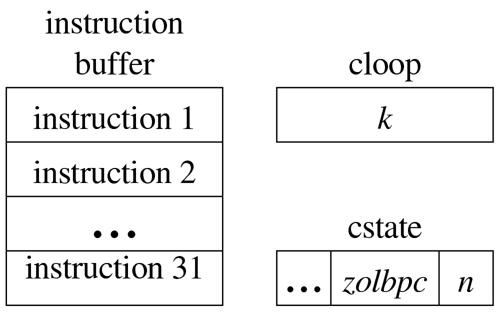


Figure 3. Example of Using the ZOLB on the DSP16000

Figure 3 depicts some of the hardware used for a ZOLB, which includes a 31 instruction buffer, a `cloop` register initially assigned the number of iterations (k) and implicitly decremented on each iteration, and a `cstate` register containing the number of instructions in the loop (n) and the pointer to the current instruction to execute (`zolbpc`). Performance benefits are achieved whenever the number of iterations executed is greater than one.

Figure 4 shows a simple example of exploiting the ZOLB on the DSP16000. Figure 4(a) contains the source code for a simple loop. Figure 4(b) depicts the corresponding code for the DSP16000 without placing instructions in the ZOLB. The effects of these instructions are also shown in this figure. The array in Figure 4(a) and the arrays in the other examples in the paper are of type `short`. Thus, the postincrement causes `r0` to be incremented by 2. Many DSP architectures use an instruction set that is highly specialized for known DSP applications. The DSP16000 is no exception and its instruction set has many complex features, which include separation of address (`r0-r7`) and accumulator (`a0-a7`) registers, postincrements of address registers, and implicit sets of condition codes from accumulator operations. Figure 4(b) also

```
for (i = 0; i < 10000; i++)
   a[i] = 0;
```

(a) Source Code of Loop

```
    r0 = _a        # r[0]=ADDR(_a);
    a2 = 0         # a[2]=0;
    a1 = -9999     # a[1]= -9999;
L5: *r0++ = a2     # M[r[0]]=a[2]; r[0]=r[0]+2;
    a1 = a1 + 1    # a[1]=a[1]+1; IC=a[1]+1?0;
    if le goto L5  # PC=IC<=0?L5:PC;
```

(b) DSP16000 Assembly and Corresponding RTLs without Using the ZOLB

```
cloop = 10000
r0 = _a
a2 = 0
do cloop {
*r0++ = a2
}
```

(c) After Using the ZOLB

Figure 4. ZOLB Hardware

shows that the loop variable is set to a negative value before the loop and is incremented on each loop iteration. This strategy allows an implicit comparison to zero with the increment to avoid performing a separate comparison instruction. Figure 4(c) shows the equivalent code after placing the loop in the ZOLB. The branch in the loop is deleted since the loop will be executed the desired number of iterations. After applying basic induction variable elimination and dead store elimination, the increment and initialization of `a1` are removed. Thus, the loop overhead has been eliminated.

3. PLACING MORE LOOPS IN A ZOLB

In this section, we will discuss the limiting factors that can prevent exploiting a ZOLB for a loop nest, and show several techniques that we used to address each of these factors.

```
for (i = 0; i < 10000;
     i++)
   if (a[i] > 0)
      sum += a[i];
```

(a) Original Source Code

```
    r0 = _a
    a1 = -9999
L5: a0 = *r0
    a0 = a0
    if gt goto L4
    a2 = a2 + a0
L4: r0 = r0 + 2
    a1 = a1 + 1
    if le goto L5
```

(b) DSP16000 Assembly without Conditional Instructions

```
    r0 = _a
    a1 = -9999
L5: a0 = *r0
    a0 = a0
    if le a2 = a2 + a0
    r0 = r0 + 2
    a1 = a1 + 1
    if le goto L5
```

(c) DSP16000 Assembly with Conditional Instructions

Figure 5. Example of Using Conditional Instructions to Place More Loops in a ZOLB

3.1. Transfers of Control in a Loop

One limiting factor that prevents the exploitation of a ZOLB for many loops is that transfers of control cannot be executed from a ZOLB. This limitation can be partially overcome by the use of conditional instructions. Consider the example source code in Figure 5(a), which shows a loop with an assignment that is dependent on a condition. The assembly code in Figure 5(b) cannot be placed into a ZOLB since there is a conditional branch that is not associated with the exit condition of the loop.[§]

Our compiler used predicated execution when possible to avoid this problem [1]. Figure 5(c) depicts the same loop with a conditional instruction and this loop can be transformed to be executed from a ZOLB. Unfortunately, many potential loops could not be placed in a ZOLB

[§]The `a0 = a0` instruction is used to set the condition codes, which are not set by the previous load instruction.

```
int abs(int v)
{
   if (v < 0)
      v = -v;
   return v;
}
...
sum = 0;
for (i = 0; i < 10000; i++)
   sum += abs(a[i]);
...
```

(a) Source Code

```
_abs: a0 = a0
      if lt a0 = -a0
      return
      ...
      r4 = _a
      a5 = 0
      a4 = -9999
L5:   a0 = *r4++
      call _abs
      a5 = a5 + a0
      a4 = a4 + 1
      if le goto L5
```

(b) Before Inlining

```
   r4 = _a
   a5 = 0
   a4 = -9999
L5:a0 = *r4++
   a0 = a0
   if lt a0 = -a0
   a5 = a5 + a0
   a4 = a4 + 1
   if le goto L5
```

(c) After Inlining

Figure 6. Example of Inlining a Function to Allow a Loop to Be Placed in a ZOLB

since predicates are assigned to a single condition code register on the DSP16000 and only a subset of the DSP16000 instructions can be conditionally executed.

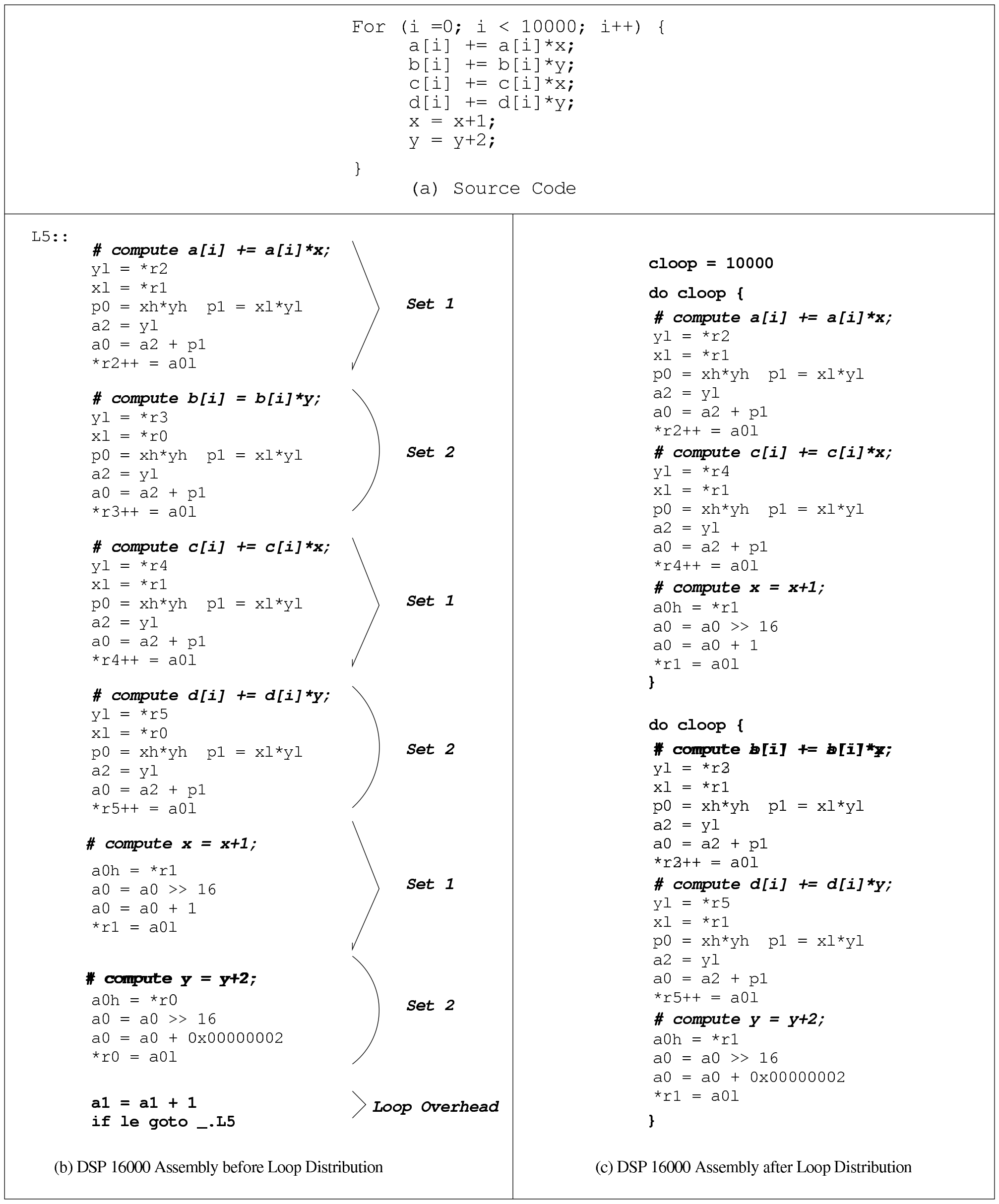
A call instruction is another transfer of control that cannot be placed in the DSP16000 ZOLB. Consider the source code and corresponding DSP16000 assembly in Figures 6(a) and 6(b). The loop cannot be placed in a ZOLB since it contains a call to `_abs`. However, the function can be inlined as shown in Figure 6(c) and the ZOLB can be used for the resulting loop. The DSP16000 optimizer does not inline indiscriminately due to potential growth in code size. However, the optimizer inlines functions that are called from a loop when the loop after inlining can be placed in the ZOLB (i.e. limited code growth for measurable performance benefits). Likewise, inlining of a function is performed when the function is only called from one site (i.e. no code growth) [13].

3.2. Too Many Instructions within a Loop

Another factor that sometimes prevented loops from being placed in the DSP16000 ZOLB was the limit of 31 instructions in the buffer. Consider the loop in Figure 7(a). When translated to DSP16000 assembly, this loop requires 34 instructions as shown in Figure 7(b), which cannot fit into the ZOLB. However, not all of the statements in the loop are dependent. We used a conventional loop transformation technique, called *loop distribution*, to address this problem.

The technique splits loops exceeding the ZOLB limit if the sets of dependent instructions can be reorganized into separate loops that can all be placed in the ZOLB. It first finds all of the sets of dependent instructions. As an illustration, consider the code shown in Figure 7(b). The optimizer can detect two sets of instructions by applying dependence checking, where one is the code fragment denoted as *set 1*, and the other is the fragment denoted as *set 2*. Conditional branch and the instructions that contribute to setting the condition codes for that branch are treated separately since they will be placed with each set. Note that these instructions will typically be deleted once loops are placed in the ZOLB by applying basic induction variable elimination and dead store elimination transformations.

The technique then checks if each set of instructions will fit in the ZOLB and combines multiple sets together when they would not exceed the maximum instructions that the ZOLB can hold. Figure 7(c) shows the actual code after distributing the original loop into two. Now each of the two loops require 18 DSP16000 instructions and both can be placed in a ZOLB.



```
For (i =0; i < 10000; i++) {
     a[i] += a[i]*x;
     b[i] += b[i]*y;
     c[i] += c[i]*x;
     d[i] += d[i]*y;
     x = x+1;
     y = y+2;
}
```

(a) Source Code

```
L5::
     # compute a[i] += a[i]*x;
     yl = *r2
     xl = *r1
     p0 = xh*yh  p1 = xl*yl
     a2 = yl
     a0 = a2 + p1
     *r2++ = a0l

     # compute b[i] = b[i]*y;
     yl = *r3
     xl = *r0
     p0 = xh*yh  p1 = xl*yl
     a2 = yl
     a0 = a2 + p1
     *r3++ = a0l

     # compute c[i] += c[i]*x;
     yl = *r4
     xl = *r1
     p0 = xh*yh  p1 = xl*yl
     a2 = yl
     a0 = a2 + p1
     *r4++ = a0l

     # compute d[i] += d[i]*y;
     yl = *r5
     xl = *r0
     p0 = xh*yh  p1 = xl*yl
     a2 = yl
     a0 = a2 + p1
     *r5++ = a0l

    # compute x = x+1;
     a0h = *r1
     a0 = a0 >> 16
     a0 = a0 + 1
     *r1 = a0l

    # compute y = y+2;
     a0h = *r0
     a0 = a0 >> 16
     a0 = a0 + 0x00000002
     *r0 = a0l

     a1 = a1 + 1
     if le goto _.L5
```

(b) DSP 16000 Assembly before Loop Distribution

```
cloop = 10000
do cloop {
 # compute a[i] += a[i]*x;
 yl = *r2
 xl = *r1
 p0 = xh*yh  p1 = xl*yl
 a2 = yl
 a0 = a2 + p1
 *r2++ = a0l
 # compute c[i] += c[i]*x;
 yl = *r4
 xl = *r1
 p0 = xh*yh  p1 = xl*yl
 a2 = yl
 a0 = a2 + p1
 *r4++ = a0l
 # compute x = x+1;
 a0h = *r1
 a0 = a0 >> 16
 a0 = a0 + 1
 *r1 = a0l
}

do cloop {
 # compute b[i] += b[i]*y;
 yl = *r3
 xl = *r1
 p0 = xh*yh  p1 = xl*yl
 a2 = yl
 a0 = a2 + p1
 *r3++ = a0l
 # compute d[i] += d[i]*y;
 yl = *r5
 xl = *r1
 p0 = xh*yh  p1 = xl*yl
 a2 = yl
 a0 = a2 + p1
 *r5++ = a0l
 # compute y = y+2;
 a0h = *r1
 a0 = a0 >> 16
 a0 = a0 + 0x00000002
 *r1 = a0l
}
```

(c) DSP 16000 Assembly after Loop Distribution

Figure 7. Example of Distributing Loops to Allow More Loops to Be Placed in a ZOLB

3.3. Unknown Number of Iterations

A final factor preventing the use of the ZOLB is that often the number of iterations associated with a loop is unknown at run-time. However, sometimes such loops can still be placed in the ZOLB on the DSP16000. Consider the source code shown in Figure 8(a) and the corresponding DSP16000 assembly shown in Figure 8(b). The number of iterations is unknown since it is not known which will be the first element of array `a` that will be equal to `n`. For each iteration of a ZOLB loop on the DSP16000 the `cloop` register is implicitly decremented by one and then tested. The ZOLB is exited when this register is equal to zero. Thus, assigning a value of one to the `cloop` register will cause the loop to exit after the current iteration completes. The loop in Figure 8(b) can be transformed to be placed in the ZOLB since the `cloop` register can be conditionally assigned a value in a register. Figure 8(c) depicts the transformed code. The `cloop` register is initially set to the maximum value to which it can be assigned and a register, `a3`, is allocated to hold the value 1. The `a[i] != n` test is accomplished by the last three instructions in Figure 8(b).

To enforce an exit from the ZOLB on the DSP16000, the `cloop` register must be assigned a value of 1 at least three instructions before the end of the loop due to the latency requirements of the machine. Moving three instructions after the branch, comparison, and instructions that affect the comparison often required the optimizer to perform register renaming and adjust the displacements of memory references, as shown in Figure 8(c). Since the loop can eventually exit due to the `cloop` register being decremented to zero without being set in the conditional assignment, another loop is placed after the ZOLB loop that will repeatedly `redo` the ZOLB loop until the exit condition has been satisfied. Note that unlike ZOLB loops with a known

```
sum = 0;
for (i = 0; a[i] != n; i++)
   sum += a[i]*2;
```

(a) Source Code of Loop

```
    r0 = _a
    a2 = 0
    r1 = _n
    a0 = *r0
    a1 = *r1
    a0 - a1
    if eq goto L3
L5: a0 = *r0++
    a0 = a0 <<< 1
    a2 = a2 + a0
    a0 = *r0
    a0 - a1
    if ne goto L5
L3:
```

(b) DSP16000 Assembly without Using the ZOLB

```
     ...
     if eq goto L3
     cloop = <max value>
     a3 = 1
     do cloop {
     a4 = *(r0+2)
     a4 - a1
     if eq cloop = a3
     a0 = *r0++
     a0 = a0 <<< 1
     a2 = a2 + a0
     }
     goto L01
L02: cloop = <max value>
     redo cloop
L01: a4 - a1
     if ne goto L02
L3:
```

(c) DSP16000 Assembly after Using the ZOLB

Figure 8. Example of Placing a Loop with an Unknown Number of Iterations in a ZOLB

number of iterations, the number of instructions in this ZOLB loop is not less than the number of instructions before the loop was placed in the ZOLB. However, conditional branches on the DSP16000 require more cycles than conditional assignments. Other potential benefits include reducing power consumption and contention to the memory system in the loop. Thus, there is a performance benefit on the DSP16000 from placing loops with an unknown number of iterations in the ZOLB.

4. LOOP TRANSFORMATIONS TO MINIMIZE LOOP OVERHEAD

In Section 3.2, we showed how a conventional loop transformation technique was applied to better exploit the ZOLB for a loop nest in our work. In this section, we will discuss additional loop transformation techniques that we have used to further reduce loop overhead when exploiting a ZOLB.

4.1. Loop Sinking

As shown previously in Figure 4(c), basic induction variables and dead store elimination are invoked after placing a loop in a ZOLB since often assignments to the loop variable become useless due to the branch no longer being in the loop.

When the value of the basic induction variable is used after the loop and is used for no other purpose in the loop, the optimizer extracts these increments of the variable from the loop, and deletes the increments in the loop. The optimizer *sinks* a new increment of the variable by placing an instruction after the loop that computes the product of the original increment and the number of loop iterations, as shown in Figure 9. Loop sinking can be viewed as a special form of the basic induction variable elimination.

4.2. Loop Collapsing

Another approach that is often used to reduce the overhead associated with outer level loops is to *collapse* nested loops into a single loop. Figure 10(a) shows perfectly nested loops that initialize every element of a matrix. Figure 10(b) shows how the array is conceptually accessed after these loops are collapsed by our optimizer into a single loop. After the optimizer places the

```
cloop = 10000
r0 = _a
a2 = 0
do cloop {
*r0++ = a2
a1 = a1 + 1
}
```

(a) DSP16000 Assembly after Using the ZOLB with a1 Live after the Loop

```
cloop = 10000
r0 = _a
a2 = 0
do cloop {
*r0++ = a2
}
a1 = a1 + 10000
```

(b) DSP16000 Assembly after Extracting the Assignment to a1

Figure 9. Example of Extracting Increments of Basic Induction Variables from a ZOLB Loop

```
int a[50][100];

for (i = 0; i < 50; i++)
   for (j = 0; j < 100; j++)
      a[i][j] = 0;
```

(a) Original Nested Loops

```
int a[5000];

for (i = 0; i < 5000; i++)
   a[i] = 0;
```

(b) After Loop Collapsing

Figure 10. Example of Loop Collapsing to Eliminate Additional Loop Overhead

collapsed loop into the ZOLB, the loop overhead for both original loops are entirely eliminated. The optimizer collapses nested loops when it detects it is possible. Even when the inner loop cannot be placed in a ZOLB, the loop overhead is reduced since the outer loop is eliminated.

4.3. Loop Interchange

Figures 11(a) and 11(c) show the source and corresponding assembly code for an example of a loop nest that cannot be collapsed by our optimizer since not all of the elements of each row of the matrix are accessed. However, these two loops can be *interchanged*, as shown in

```
extern int a[200][100];

for (i=0; i<200; i++)
   for (j=0; j<50; j++)
      a[i][j]=0;
```

(a) Source Code of Nested Loops

```
extern int a[200][100];

for (j=0; j<50; j++)
   for (i=0; i<200; i++)
      a[i][j]=0;
```

(b) Source Code after Loop Interchange

```
    r1 = _a
    a3 = 0
    a2 = -199
L5: r0 = r1
    a1 = -49
L9: *r0++ = a3
    a1 = a1 + 1
    if le goto L9
    r1 = r1 + 200
    a2 = a2 + 1
    if le goto L5
```

(c) DSP16000 Assembly before Loop Interchange

```
    r1 = _a
    a3 = 0
    a2 = -49
L5: r0 = r1
    a1 = -199
    k = 200
L9: *r0++k = a3
    a1 = a1 + 1
    if le goto L9
    r1 = r1 + 2
    a2 = a2 + 1
    if le goto L5
```

(d) DSP16000 Assembly after Loop Interchange

```
    r1 = _a
    a3 = 0
    a2 = -49
L5: cloop = 200
    r0 = r1
    j = 200
    do cloop {
    *r0++k = a3
    }
    r1 = r1 + 2
    a2 = a2 + 1
    if le goto L5
```

(e) DSP16000 Assembly after Using the ZOLB

Figure 11. Example of Loop Interchange to Increase the Iterations Executed in the ZOLB

Figures 11(b) and 11(d). After interchanging the two loops, the inner loop now has a greater number of loop iterations, which can be executed from the ZOLB as shown in Figure 11(e). More loop overhead is now eliminated by placing the interchanged inner loop in the ZOLB as opposed to the original inner loop.

The optimizer attempts to interchange nested loops when the loops cannot be collapsed, the loops are perfectly nested, the number of iterations for the original inner loop is less than the number of iterations for the original outer loop, the number of instructions in the inner loop does not increase, and the resulting inner loop can be placed in the ZOLB. Figure 11(d) shows that register `k` was allocated to hold the value of the increment 200 so an additional instruction to increment `r0` would be unnecessary. This example illustrates the advantage of performing loop interchange after code generation since otherwise it would not be known if a register was available to be used to hold the increment and the transformation may result in more instructions in the inner loop. Note that interchanging loops should not be performed if it will degrade the performance of the memory hierarchy. This is not an issue for the DSP16000 since it has no data cache or virtual memory system and only a limited on-chip RAM.

5. Avoiding Redundant Loading of Loops into the ZOLB

The `do` instruction indicates that a specified number of instructions following the `do` will be loaded into the ZOLB. Depending upon the implementation of the DSP architecture, instructions may be fetched faster from a ZOLB than the conventional memory system. In addition, contention for the memory system may be reduced when a ZOLB is used. The `redo` instruction has similar semantics as the `do` instruction, except that the `redo` does not cause any instructions to be loaded into the ZOLB. Instead, the current contents of the ZOLB are simply executed the specified number of iterations.

The `redo` instruction can be used to avoid redundant loads of loops into the ZOLB. Consider the source code shown in Figure 12(a). It would appear that the two loops are quite different since they iterate a different number of times, access different variables, and access different types of data. However, the body of the two loops are identical as shown in Figure 12(b). The reason is that much of the characteristics of the loops have been abstracted out of the loop bodies. The number of iterations for ZOLB loops is encoded in the `do` instruction or assigned to the `cloop` register preceding the loop. After loop strength reduction is performed, the addresses of the arrays are assigned to registers associated with basic induction variables preceding the loop. In addition, data moves of the same size between registers and memory are

```
extern int a[100], b[100];
extern float c[200], d[200];
...
for (i = 0; i < 100; i++)
   a[i] = b[i];
...
for (i = 0; i < 200; i++)
   c[i] = d[i];
...
```

(a) Source Code of Two Different Loops

```
r1 = _a
r0 = _b
do 100 {
a0 = *r0++
*r1++ = a0
}
...
cloop = 200
r1 = _c
r0 = _d
do cloop {
a0 = *r0++
*r1++ = a0
}
```

(b) DSP16000 Assembly after Using the ZOLB

```
r1 = _a
r0 = _b
do 100 {
a0 = *r0++
*r1++ = a0
}
...
cloop = 200
r1 = _c
r0 = _d
redo cloop
```

(c) DSP16000 Assembly after Avoiding the Redundant ZOLB Load

Figure 12. Example of Avoiding a Redundant Load of the ZOLB

accomplished in the same manner on the DSP16000, regardless of the data types. Figure 12(c) shows the assembly code after the redundant loop is eliminated using the `redo` instruction.

The optimizer determines which ZOLB loops can reach each point in the control flow without the contents of the ZOLB being changed. We used data flow analysis to determine if the loading of each ZOLB loop was necessary. A bit was associated with each ZOLB loop and one bit was also reserved to indicate that no ZOLB loops could reach a given point. Equations (1) and (2) are used to determine which ZOLB loops could possibly reach each point in the control flow within a function.¶

¶Note that `B` represents a basic block in the program

In the actual implementation, interprocedural flow analysis was used to avoid redundant loading of ZOLB loops across function calls and returns. An adjustment was required when ZOLB loop information was propagated from a return block of a function. This adjustment prevented ZOLB loops that are propagated into the entry block of a function at one call site from being propagated to the block following a call to the same function at a different call site. Likewise, it was assumed that no ZOLB loops could reach the point after a library call since it was not known if the ZOLB would be used for a different ZOLB loop in the called library function.

$$\texttt{in[B]} = \begin{cases} \texttt{Null} & \texttt{if B is a function entry block} \\ \displaystyle\bigcup_{\texttt{P} \in \texttt{pred[B]}} \texttt{out[P]} & \texttt{otherwise} \end{cases} \tag{1}$$

$$\texttt{out[B]} = \begin{cases} \texttt{Null} & \texttt{if B contains a call} \\ \texttt{B} & \texttt{if B contains a ZOLB loop} \\ \texttt{in[B]} & \text{otherwise} \end{cases} \tag{2}$$

After all of the ZOLB loop reaching information is calculated, the optimizer determines which ZOLB loops do not need to be loaded into the ZOLB. If the `in[]` of a current block containing a ZOLB loop indicates that only a single other ZOLB loop is guaranteed to reach that point and if all of the instructions in the other ZOLB loop are identical with the instructions in the current ZOLB loop, then the entire current ZOLB loop is replaced with a `redo` instruction.

Even after using flow analysis to avoid redundant loads of ZOLB loops, many loops are repeatedly loaded into the ZOLB because they are in nested loops. The optimizer was modified

```
for (i = 0; i < 1000; i++)
   for (j = 0; j < 2000; j++)
      a[i][j] = 0;
```

(a) Source Code of Nested Loops

```
    r1 = _a
    a3 = 0
    a2 = -999
L5: r0 = r1
    a1 = -1999
L9: *r0++ = a3
    a1 = a1 + 1
    if le goto L9
    r1 = r1 + 200
    a2 = a2 + 1
    if le goto L5
```

(b) DSP16000 Assembly without Using the ZOLB

```
    r1 = _a
    a3 = 0
    a2 = -999
L5: cloop = 2000
    r0 = r1
    do cloop {
    *r0++ = a3
    }
    r1 = r1 + 200
    a2 = a2 + 1
    if le goto L5
```

(c) DSP16000 Assembly after Using the ZOLB

```
    r1 = _a
    a3 = 0
    a2 = -999
    cloop = 2000
    r0 = r1
    do cloop {
    *r0++ = a3
    }
    r1 = r1 + 200
    a2 = a2 + 1
L5: cloop = 2000
    r0 = r1
    redo cloop
    r1 = r1 + 200
    a2 = a2 + 1
    if le goto L5
```

(d) DSP16000 Assembly after Loop Peeling

Figure 13. Using Loop Peeling to Avoid Repeated Loads of the ZOLB

to have the ability to avoid these redundant loads as well. Figures 13(a) and 13(b) contain the source and corresponding DSP assembly for two nested loops, respectively.

Figure 13(c) shows the assembly after the inner loop is placed in the ZOLB. For each iteration of the outer loop, the inner loop is loaded into the ZOLB using the `do` instruction. Figure 13(d) shows how the optimizer avoids the repeated loading of the inner loop in the ZOLB by peeling an iteration of the outer loop. The optimizer avoids the repeated loading of the inner loop in the ZOLB by peeling an iteration of the outer loop. Only in the peeled iteration is the ZOLB loaded. All remaining iterations execute from the ZOLB using the `redo` instruction. The optimizer only performs the loop peeling transformation when the increase in code size is small and there are expected performance benefits (i.e. reducing memory bus

1. Build call graph for the program
2. Merge consecutive blocks
3. Find the loops in the program
4. Calculate live register info
5. Convert branches into conditional assignments
6. Find loop invariant & induction variables
7. Calculate the number of loop iterations
8. Perform inlining to support placing more loops in the ZOLB
9. Calculate ranges of addresses accessed by each memory reference
10. Perform loop distribution to place more loops in the ZOLB
11. Flatten perfectly nested loops
12. Perform loop interchange
13. Place loops in the ZOLB
14. Eliminate basic induction variable
15. Extract loop induction variable assignment
16. Avoid redundant loading of the ZOLB
17. Perform loop peeling to further avoid redundant ZOLB loading

Figure 14. Order of the Analysis and Transformations Used to Exploit a ZOLB

contention conflicts on the DSP16000) from avoiding the repeated load of the inner loop into the ZOLB.

6. RESULTS

The order in which these transformations are applied can affect how effectively a ZOLB can be exploited. Figure 14 shows the order of the pertinent analysis and transformations that are applied on the assembly code in the second optimization phase shown in Figure 1. The complete list of types of analysis and code improving transformations performed in this phase of optimization and a more thorough description and rationale for this order may be found elsewhere [8]. Likewise, a more general description of these analyses and transformations can also be obtained [14].

Table I. Test Programs

Program	Description	Program	Description
add8	add two 8-bit images	conv	convolution code
copy8	copy one 8-bit image to another	fft	128 point complex fft
fir	finite impulse response filter	fir_no	fir filter with redundant load elimination
fire	fire encoder		
inverse8	invert an 8-bit image	iir	iir filtering
lms	lms adaptive filter	jpegdct	jpeg discrete cosine transform
sumabsd	sum of absolute differences of two images	scale8	scale an 8-bit image
		trellis	trellis convolutional encoder
vec_mpy	simple vector multiply		

Table 1 describes the benchmarks and applications used to evaluate the impact of using the ZOLB on the DSP16000. All of these test programs are either DSP benchmarks used in industry or typical DSP applications. Many DSP benchmarks represent kernels of programs where most of the cycles occur. Such kernels in DSP applications have been historically optimized in assembly code by hand to ensure high performance [6]. Thus, many established DSP industrial benchmarks are small since they were traditionally hand coded. Standard benchmarks (e.g. SPEC) were not used since the DSP16000 was not designed to support operations on floating-point values or integers larger than two bytes.

Table 2 contrasts the results for loop unrolling and exploiting the DSP16000 ZOLB.‖ Execution measurements were obtained by accessing a cycle count from a DSP16000 simulator [9]. Code size measurements were gathered by obtaining diagnostic information provided by the assembler [10]. We compared the performance of using the ZOLB against loop unrolling,

‖Only relative performance results could be given due to disclosure restrictions for these test programs.

which is a common approach for reducing loop overhead. The loop unrolling showed in Table 2 was performed on all innermost loops when the number of iterations was known statically or dynamically. As shown in the results, using the ZOLB typically resulted in fewer execution cycles as compared to loop unrolling. Sometimes loop unrolling did have benefits over using a ZOLB. This occurred when an innermost loop had too many instructions or had transfers of control that would prevent it from being placed in a ZOLB. In addition, sometimes loop unrolling provided other benefits, such as additional scheduling and instruction selection opportunities, that would not otherwise be possible.** However, the average performance benefits of using a ZOLB are impressive, particularly when code size is important. As shown in the table, loop unrolling caused significant code size increases, while using the ZOLB resulted in slight code size decreases. The code size decreases when using the ZOLB came from the combination of eliminating branches by placing the loops in the ZOLB and applying induction variable elimination and dead store elimination afterwards. Occasionally, code size decreases were obtained by avoiding redundant loads of the ZOLB loops using the flow analysis described in Section 5. Loop peeling, which increases code size, was rarely applied since memory contentions did not occur that frequently.

**The production version of the optimizer does limited unrolling of loops. For instance, loop unrolling is applied when memory references and multiplies can be coalesced. However, unrolling is not performed when it would cause the number of instructions to exceed the limit that the ZOLB can hold [8]. Note the measurements presented in this paper did not include loop unrolling while placing loops in the ZOLB since it would make the comparison of applying loop unrolling and using a ZOLB less clear. Likewise, the production version of the optimizer performs other optimizations, such as multiply and memory coalescing and software pipelining, that were not applied for the results in this paper.

Table II. Contrasting Loop Unrolling and Using a ZOLB

Program	Unroll Factor = 2		Unroll Factor = 4		Unroll Factor = 8		Exploiting ZOLB	
	Cycles	Code Size	Cycles	Code Size	Cycles	Code Size	Cycle	Code Size
add8	-11.5%	+7.8%	-23.1%	+62.8%	-27.5%	+90.2%	-36.3%	-3.9%
conv	-33.4%	+22.6%	-47.6%	+29.0%	-54.6%	+41.9%	-47.8%	-3.2%
copy8	-23.1%	+6.3%	-42.3%	+12.5%	-52.0%	+25.0%	-62.4%	-4.2%
fft	-6.2%	+32.1%	-10.6%	+92.9%	-12.7%	+214.3%	-8.7%	-3.6%
fir	-20.4%	+21.1%	-35.3%	+147.4%	-42.0%	+255.3%	-48.4%	-10.5%
fir_no	-4.0%	+34.9%	-7.1%	+109.3%	-9.1%	+258.1%	-31.4%	-4.7%
fire	-0.8%	+36.3%	-4.2%	+110.8%	-6.2%	+255.9%	-26.9%	-21.6%
iir	-11.1%	+14.6%	-15.4%	+51.0%	-15.7%	+88.5%	-19.6%	-4.2%
inverse8	-20.3%	+8.2%	-37.3%	+18.4%	-46.6%	+49.0%	-55.5%	-4.1%
jpegdct	-8.3%	+17.6%	-8.4%	+59.5%	-8.4%	+59.5%	0.0%	0.0%
lms	-1.8%	+0.5%	-10.5%	+1.8%	-10.5%	+1.8%	-8.3%	0.0%
scale8	-4.9%	+38.5%	-9.4%	+93.9%	-11.6%	+204.6%	-14.3%	-1.5%
sumabsd	-14.7%	+8.6%	-19.6%	+25.7%	-22.0%	+60.0%	-58.8%	-8.6%
trellis	-11.5%	+0.1%	-19.1%	+0.3%	-22.8%	+0.8%	-20.2%	-0.2%
vec_mpy	-19.1%	+63.2%	-28.5%	+336.8%	-31.2%	+531.6%	-38.2%	-15.8%
Average	**-12.7%**	**+20.8%**	**-21.2%**	**+76.8%**	**-24.9%**	**+142.4%**	**-31.8%**	**-5.7%**

Table 3 depicts the benefit of applying the code improving transformations described in Sections 4 and 5. Only some of the code improving transformations applied without using a ZOLB (column 2) had a performance benefit on their own. These transformations include the use of conditional instructions, inlining, and loop collapsing. The characteristics of the DSP16000 prevented conditional instructions from being used frequently. Inlining only had occasional benefits for the test programs since the optimizer only inlined functions when the function was called from a loop and inlining would allow the loop to be placed in the ZOLB. Inlining was not performed when a function had transfers of control other than a return instruction, which was the common case. Loop collapsing was applied most frequently of these transformations. The results shown in column 3 include basic induction variable elimination since it was quite obvious that this transformation could almost always be applied when a loop is placed in the ZOLB. The combination of using the ZOLB with the code improving

Table III. The Impact of Code Improving Transformations on Using a ZOLB

Program	**Impact on Execution Cycles**		
	Transformations without Using the ZOLB	Using the ZOLB without Transformations	Using the ZOLB with Transformations
add8	-2.2%	-35.1%	-37.8%
conv	-8.2%	-43.5%	-52.1%
copy8	-1.8%	-60.4%	-63.1%
fft	0.0%	-8.7%	-8.7%
fir	0.0%	-48.4%	-48.4%
fir_no	0.0%	-31.4%	-31.4%
fire	-7.4%	0.0%	-32.3%
iir	0.0%	-19.6%	-19.6%
inverse8	-1.6%	-53.8%	-56.2%
jpegdct	0.0%	0.0%	0.0%
lms	0.0%	-8.3%	-8.3%
scale8	-3.8%	-16.9%	-17.5%
sumabsd	-23.1%	0.0%	-51.7%
trellis	-8.8%	-7.4%	-20.2%
vec_mpy	0.0%	-38.2%	-38.2%
Average	**-3.8%**	**-25.3%**	**-33.0%**

transformations (column 4) sometimes resulted in greater benefits than the sum of the benefits (columns 2 and 3) when applied separately. Most of the additional benefit came from the new opportunities for placing more loops in the ZOLB (transformations described in Section 4).

We also obtained the percentage of the innermost loops that were placed in the ZOLB. On average 71.56% of the innermost loops could be placed in the ZOLB without applying the code improving transformations described in Section 4. However, 84.89% of the innermost loops could be placed in the ZOLB with these improving transformations applied. Transfers of control was the most common factor that prevented the use of a ZOLB. The use of conditional instructions, inlining, and the transformation on loops with an unknown number of iterations all occasionally resulted in additional loops being placed in the ZOLB.

7. RELATED WORK

A number of hardware and software techniques have been used to reduce loop overhead. Common hardware techniques include branch prediction hardware to reduce branch mispredictions and superscalar or VLIW logic to allow other operations to execute in parallel with the loop overhead instructions. However, the use of complex hardware mechanisms to minimize branch overhead results in the consumption of more power.

Some current general-purpose processors have a single instruction that increments a loop counter, compares the counter to a constant, and performs a branch. Common software techniques include loop strength reduction with basic induction variable elimination, loop unrolling [2] and software pipelining [15, 16, 17, 18]. Note that loop unrolling and software pipelining can significantly increase code size, which can be unacceptable for many DSPs with limited on-chip memory. Therefore, DSP programmers have manually tuned their assembly code. However, maintaining the required software in assembly language by manually exploiting the ZOLB is labor intensive given the frequent changes in communication signal processing algorithms with frequent architecture/instruction set modifications.

To meet the fast time-to-market window without violating stringent performance and code size constraints on software, optimizing compilers are becoming more widely available for DSP applications. As an illustration of successful DSP compilers, the TI VLIW TMS320C6x optimizing compiler has been widely cited. This optimizing compiler is specially designed to tune the Huff's iterative modulo scheduler to exploit the slack created from different latencies in a *multiply* instruction and a *load* instruction [16, 19].

As an illustration for successful DSP compiler construction tools, CoSy (COmpiler SYstem) provides a novel framework for flexible combination and embedding of compiler phases such that the construction of parallel and inter-procedural optimizing compilers is facilitated. This framework is designed to automatically generate a compiler infrastructure that solves the phase ordering problem in the presence of highly coupled optimization phases, which are common for low power DSPs [20].

To the best of our knowledge, no other work describes how a ZOLB can be exploited by a compiler, the interaction of exploiting a ZOLB with other code improving transformations, and the performance benefits that can be achieved from using a ZOLB.[††].

8. CONCLUSION

This paper described strategies for generating code and utilizing code improving transformations to exploit a ZOLB. We found that many conventional code improving transformations used in optimizing compilers significantly affect how a ZOLB can be exploited. The use of predicated execution, loop distribution, and function inlining allowed more loops to be placed in a ZOLB. The overhead of loops placed in a ZOLB was further reduced by basic induction variable elimination, loop sinking, loop collapsing, and loop interchange.

We also found that a ZOLB can improve performance in ways probably not intended by the architects who originally designed this feature. The use of conditional instructions and instruction scheduling with register renaming allowed some loops with an unknown number of

[††]Preliminary versions of this paper have appeared in [11] and [12]

iterations to be placed in a ZOLB. Interprocedural flow analysis and loop peeling were used with the `redo` instruction to avoid redundant loading of a ZOLB. The results obtained from test programs indicate that these transformations allowed a ZOLB to be often exploited with significant improvements in execution time and small reductions in code size.

REFERENCES

1. Hennessy, J., Patterson, D. Computer Architecture: *A Quantitative Approach, Second Edition.* Morgan Kaufmann, San Francisco, 1996.
2. Davidson, J.W., Jinturkar, S. *Aggressive Loop Unrolling in a Retargetable, Optimizing Compiler.* Proceedings of Compiler Construction Conference, 59–73, April 1996.
3. Lucent Technologies. *DSP16000 Digital Signal Processor Core Information Manual.* Core Reference Manual, 1997.
4. Lucent Technologies. *DSP16000 C Compiler User Guide.* Tools Reference Manual, 1997.
5. Lapsley, P., Bier, J., Lee, E. *DSP Processor Fundamentals - Architecture and Features.* IEEE Press, 1996.
6. Eyre, J., Bier, J. *DSP Processors Hit the Mainstream.* IEEE Computer 31(8), 51–59, August 1998.
7. Lucent Technologies. *DSP16000 Digital Signal Processor Core Instruction Set Manual.* Core Reference Manual, 1997.
8. Whalley, D. *DSP16000 C OPtimizer Overview and Rationale.* Lucent Technologies, Allentown, 1998.
9. Lucent Technologies. *DSP16000 LuxWorks Debugger* Tools Reference Manual, 1997.
10. Lucent Technologies.: *DSP16000 Assembly Language User Guide.* Tools Reference Manual, 1997.
11. Uh, G.R., Wang, Y., Whalley, D. Jinturkar, S., Burns, C., and Cao, V. *Effective Exploitation of a Zero Overhead Loop Buffer.* ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems, 10–19, 1999.
12. Uh, G.R., Wang, Y., Whalley, D. Jinturkar, S., Burns, C., and Cao, V. *Techniques for Effective Exploitation of a Zero Overhead Loop Buffer.* 9th International Conference on Compiler Construction, March 2000.
13. Wang, Y. *Interprocedural Optimizations for Embedded Systems.* Masters Project, Florida State University, Tallahassee, FL, 1999.

14. Bacon, D., Graham, S., Sharp, O. *Compiler Transformations for High-Performance Computing* ACM Computing Surveys, Volume 26 Number 4, 345–420, 1994.

15. Eichenberger, A. E. *Modulo Scheduling, Machine Representations, and Register-Sensitive Algorithms* Ph.D Thesis, University of Michigan, 1997

16. Huff, R. A. *Lifetime-Sensitive Modulo Scheduling* Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 258-267, June 1993

17. Lam, M. *Software Pipelining: An Effective Scheduling Technique for VLIW Machines* Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 318-329, June 1988

18. Rau, B. R. *Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops* Proceedings of the 27th Annual International Symposium on Microarchitecture, pp. 63-74, November 1994

19. Stotzer, E. and Leiss, E. *Modulo Scheduling for the TMS320C6x VLIW DSP Architecture* Proceedings of the AC¡ SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99) pp. 28-34, May 1999

20. Alt, M. and Someren, H. V. *Cosy Compiler Phase Embedding with the CoSy Compiler Model* Compiler Construction (CC), vol. 786, Lecture Notes in Computer Science, pp. 278-293, Heidelberg, Springer, April 1994