Techniques for Effectively Exploiting a Zero Overhead Loop Buffer

Gang-Ryung Uh[1], Yuhong Wang[2], David Whalley[2], Sanjay Jinturkar[1], Chris Burns[1], and Vincent Cao[1]

[1] Lucent Technologies, Allentown, PA 18103, U.S.A.
[2] Computer Science Dept., Florida State Univ., Tallahassee, FL 32306-4530, U.S.A.
{uh,sjinturkar,cpburns,vpcao}@lucent.com, {yuhong,whalley}@cs.fsu.edu

Abstract. 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. 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 most DSP applications. This paper describes strategies for generating code to effectively use a ZOLB. The authors have found that many common improving transformations used by optimizing compilers to improve code on conventional architectures can be exploited (1) to allow more loops to be placed in a ZOLB, (2) to further reduce loop overhead of the loops placed in a ZOLB, and (3) to 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.

1 Introduction

The number of DSP processors is growing every year at a much faster rate than general-purpose computer processors. For many applications, a large percentage of the execution time is spent in the innermost loops of a program [1]. The execution of these loops incur significant overhead, which is due to the increment and branch instructions to initiate a new iteration of a loop. Many code improving transformations and architectural features used to improve execution time for applications in general-purpose computers do so at the expense of substantial code growth and more power consumption. For instance, loop unrolling is a popular technique to decrease loop overhead [2]. Yet, this approach often requires a significant increase in code size. Likewise, VLIW instructions can be used to reduce loop overhead at the expense of more power. Space increasing transformations and power inefficient architectures are often unacceptable options for many DSP applications due to these limitations.

A zero overhead loop buffer (ZOLB) is an architectural feature commonly found in DSP processors. This buffer can be used to increase the speed of applications with no increase in code size and often with reduced power consumption. A

ZOLB is a buffer that can contain a fixed number of instructions to be executed a specified number of times under program control. Depending on the implementation of the DSP architecture, some instructions may be fetched faster from a ZOLB than from the conventional instruction memory. In addition, the same memory bus used to fetch instructions can sometimes 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 controlled cache since special instructions are used to load instructions into it.

This paper describes strategies for exploiting the ZOLB that is available on the DSP16000 architecture [3], which could also be applied to other DSP architectures that have ZOLBs. These strategies have the potential for being readily adopted by compiler writers for DSP processors since they rely on the use of traditional compiler improving transformations and data flow analysis techniques. Figure 1 presents an overview of the compilation process used by the authors to generate and improve code for this architecture. Code is generated using a C compiler retargeted to the DSP16000 [4]. Conventional improving transformations in this C compiler are applied and assembly files are generated. Finally, the generated code is then processed by another optimizer, which performs a number of 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.

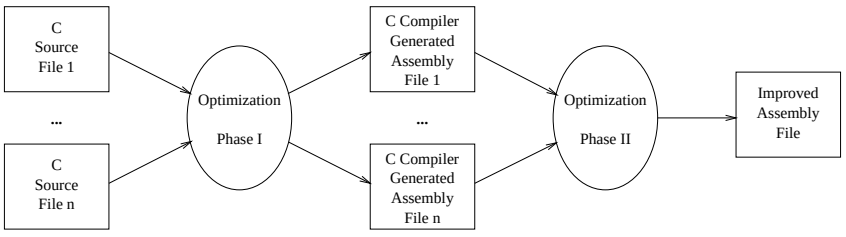

Fig. 1. Overview of the Compilation Process for the DSP16000

2 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 execution to allow other operations to execute in parallel with the loop overhead instructions [1]. 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 loop branch instruction that eliminates the incrementing of a loop counter and a comparison, but still require the branch instruction. Common software techniques include loop strength reduction with basic induction variable elimination and loop unrolling. Note that loop unrolling can significantly increase code size.

Currently available versions of ZOLBs in TI, ADI, and Lucent processors have been described [5]. Assembly language programmers for DSPs commonly use ZOLBs in the code that they write. However, optimizing compilers have been used only recently for DSP applications and programmers still tend to write critical sections by hand [6]. A preliminary version of this paper appeared in a workshop [12]. 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 improving transformations, and the performance benefits that can be achieved from using a ZOLB.

3 Using the DSP16000 ZOLB

The target architecture for which the authors 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 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 a value of zero is encoded and the number of times the instructions in the ZOLB will be executed is obtained from the `cloop` register. 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 depicts some of the hardware used for a ZOLB, which includes a 31 instruction buffer, a `cloop` register initially assigned the number of iterations and implicitly decremented

on each iteration, and a `cstate` register containing the number of instructions in the loop and the pointer to the current instruction to load or execute. Performance benefits are achieved whenever the number of iterations executed is greater than one.

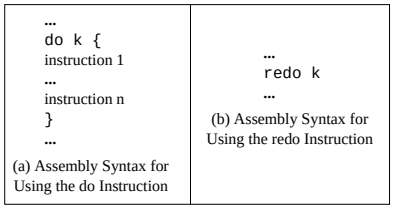

Fig. 2. DSP16000 Assembly Syntax for Using the ZOLB

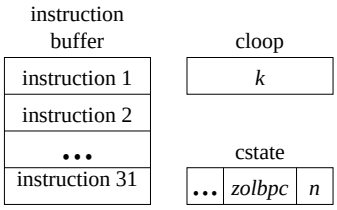

Fig. 3. Example of Using the ZOLB on the DSP16000

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 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.

```
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

Fig. 4. ZOLB Hardware

4 Placing More Loops in a ZOLB

The limiting factors that can prevent exploiting a ZOLB for an innermost loop are (1) transfers of control other than the loop branch, (2) the number of instructions in the loop exceeding the ZOLB limit, and (3) the number of iterations being unknown at run-time. In this section we describe techniques that can often address each of these factors.

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.[1] 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 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.

```
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

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

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) [8].

Another factor that sometimes prevented loops from being placed in the DSP16000 ZOLB was the limit of 31 instructions in the buffer. The authors

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

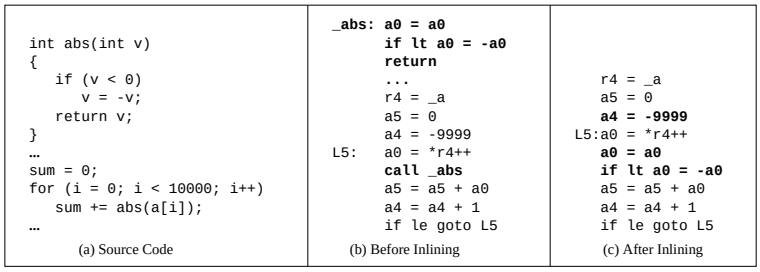

Fig. 6. Example of Inlining a Function to Allow a Loop to Be Placed in a ZOLB

implemented loop distribution to address this problem. The optimizer splits loops exceeding the ZOLB limit if the sets of dependent instructions can be reorganized into separate loops that can all be placed in a ZOLB. The optimizer first finds all of the sets of dependent instructions. The 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 and the basic induction variable elimination and dead store elimination transformations are applied. The optimizer 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.

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 7(a) and the corresponding DSP16000 assembly shown in Figure 7(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 7(b) can be transformed to be placed in the ZOLB since the `cloop` register can be conditionally assigned a value in a register. Figure 7(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 7(b). To force 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 7(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 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 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.

```
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

Fig. 7. Example of Placing a Loop with an Unknown Number of Iterations in a ZOLB

5 Further Reducing Loop Overhead

As shown previously in Figure 4(c), basic induction variable and dead store elimination are invoked after placing a loop in a ZOLB since often assignments to the loop variable become unnecessary 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. First, the increments in the loop are deleted. Next, a new increment of the variable is placed after the loop that is the product of the original increment and the number of loop iterations.

Another approach that is often used to reduce the overhead associated with outer level loops is to collapse nested loops. Figure 8(a) shows perfectly nested loops that initialize every element of a matrix. Figure 8(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 collapsed loop into the ZOLB, the loop overhead for both original loops are entirely eliminated. The optimizer collapses nested loops whenever possible. Even when the inner loop cannot be placed in a ZOLB, the loop overhead is reduced since the outer loop is eliminated.

Figures 9(a) and 9(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

```
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

Fig. 8. Example of Loop Collapsing to Eliminate Additional Loop Overhead

of the elements of each row of the matrix are accessed. However, these two loops can be interchanged, as shown in Figures 9(b) and 9(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 9(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 9(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. Interchanging loops will not degrade the performance of the memory hierarchy for the DSP16000 since it has no data cache or virtual memory system.

```
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

Fig. 9. Example of Loop Interchange to Increase the Iterations Executed in the ZOLB

6 Avoiding Redundant Loads of 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 10(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 10(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. The addresses of the arrays are assigned to registers associated with basic induction variables preceding the loop after loop strength reduction is performed. In addition, data moves of the same size between registers and memory are accomplished in the same manner on the DSP16000, regardless of the data types. Figure 10(c) shows the assembly code after the redundant loop is eliminated using the `redo` instruction.

```
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

Fig. 10. Example of Avoiding a Redundant Load of the ZOLB

The optimizer determines which ZOLB loops can reach each point in the control flow without the contents of the ZOLB being changed. The authors used 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 to determine which ZOLB loops could possibly reach each point in the control flow within a function.[2] 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

[2] Note that `B` represents a basic block in the program

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.

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

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

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 to have the ability to avoid these redundant loads as well. 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 contention conflicts on the DSP16000) from avoiding the repeated load of the inner loop into the ZOLB.

7 Analysis and Transformations

The order in which these transformations are applied can affect how effectively a ZOLB can be exploited. Figure 11 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 improving transformations performed in this phase of optimization and a more thorough description and rationale for this order may be found elsewhere [9]. Likewise, a more general description of these analyses and transformations can also be obtained [13].

Basic blocks are merged (#2) when possible. This transformation does not usually improve the code directly but may provide additional opportunities for other improving transformations. For instance, placing loops in a ZOLB (#13) is only applied to loops containing a single basic block. Merging basic blocks (#2) also reduces the overhead of most types of global analysis.

Analysis is performed to allow optimizations to be performed A call graph (#1) is built to perform various types of interprocedural improving transformations [8], which includes inlining (#8) to support placing loops in a ZOLB. Loops in the program are detected (#3) to support a variety of improving transformations, which of course includes placing loops in a ZOLB (#13). Live register information is calculated (#4) since many improving transformations require allocation of registers. For instance, placing a loop with an unknown number of iterations in the ZOLB (#13) requires renaming registers to newly allocated registers to accomplish the scheduling required to force an exit from the loop at the appropriate time. Loop invariant values and basic induction variables are detected (#6) so the number of iterations for a loop may be calculated (#7). Note that detecting the number of loop iterations is a much more challenging task at the assembly level as compared to examining source level loop statements.

Some instructions with immediate values cannot be executed conditionally. When these instructions are inside a loop and a register is available, the compiler replaces the immediate value with the register and assigns the immediate value to the register outside the loop. Therefore, branches are converted into conditional assignments (#5) after finding loops (#3) and calculating live register information (#4). Branches are converted into conditional assignments (#5) before analysis is performed to determine if a loop can be placed in the ZOLB (#13) since loops with branches not associated with the exit condition of the loop cannot be placed in the ZOLB.

Inlining (#8) also removes transfers of control from a loop, namely a call instruction. Inlining (#8) was performed after detecting the number of loop iterations (#7) since it could be determined at this point if the inlining would allow the loop to be placed in the ZOLB (#13) so unnecessary code growth could be avoided.

Ranges of addresses were calculated (#9) for each memory reference to allow independent instructions in a loop to be separated via loop distribution (#10). Both loop flattening (#11) and loop interchange (#12) are performed after calculating the number of loop iterations (#7) since these transformations require this information. Perfectly nested loops are flattened (#11) before loop interchange (#12) is performed since flattening loops places more iterations in a ZOLB than interchanging loops.

Basic induction variable elimination (#14) was performed after placing loops in the ZOLB (#13) since the assignments were often unnecessary at that point. The remaining assignments to basic induction variables are extracted from loops (#15) after basic induction variable elimination (#14) to prevent unnecessary extractions of instructions.

Avoiding redundant loading of the ZOLB using flow analysis was performed after loops were placed in the ZOLB so redundant loads could be detected. Finally, loop peeling was only considered for the loops whose loading could not be avoided using flow analysis since loop peeling requires a code size increase.

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

Fig. 11. Order of the Analysis and Transformations Used to Exploit a ZOLB

8 Results

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 1. 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	iir	iir filtering
inverse8	invert an 8-bit image	jpegdct	jpeg discrete cosine transform
lms	lms adaptive filter	scale8	scale an 8-bit image
sumabsd	sum of absolute differences of two images	trellis	trellis convolutional encoder
vec_mpy	simple vector multiply		

Table 2 contrasts the results for loop unrolling and exploiting the DSP16000 ZOLB.[3] Execution measurements were obtained by accessing a cycle count from a DSP16000 simulator [10]. Code size measurements were gathered by obtaining diagnostic information provided by the assembler [11]. The authors compared

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

the performance of using the ZOLB against loop unrolling, 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.[4] 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.

Table 2. 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	-11.47%	+7.84%	-23.11%	+62.75%	-27.46%	+90.20%	-36.33%	-3.92%
conv	-33.42%	+22.58%	-47.56%	+29.03%	-54.63%	+41.94%	-47.84%	-3.23%
copy8	-23.11%	+6.25%	-42.32%	+12.50%	-51.92%	+25.00%	-62.44%	-4.17%
fft	-6.22%	+32.14%	-10.56%	+92.86%	-12.73%	+214.29%	-8.69%	-3.57%
fir	-20.35%	+21.05%	-35.25%	+147.37%	-41.98%	+255.26%	-48.42%	-10.53%
fir_no	-3.97%	+34.88%	-7.07%	+109.30%	-9.14%	+258.14%	-31.35%	-4.65%
fire	-0.75%	+36.27%	-4.22%	+110.78%	-6.20%	+255.88%	-26.88%	-21.57%
iir	-11.10%	+14.58%	-15.43%	+51.04%	-15.67%	+88.54%	-19.61%	-4.17%
inverse8	-20.27%	+8.16%	-37.34%	+18.37%	-46.64%	+48.98%	-55.50%	-4.08%
jpegdct	-8.26%	+17.56%	-8.44%	+59.54%	-8.44%	+59.54%	0.00%	0.00%
lms	-1.75%	+0.48%	-10.52%	+1.78%	-10.52%	+1.78%	-8.33%	-0.04%
scale8	-4.90%	+38.46%	-9.37%	+93.85%	-11.60%	+204.62%	-14.28%	-1.54%
sumabsd	-14.69%	+8.57%	-19.57%	+25.71%	-22.03%	+60.00%	-58.83%	-8.57%
trellis	-11.52%	+0.11%	-19.10%	+0.33%	-22.79%	+0.78%	-20.16%	-0.17%
vec_mpy	-19.08%	+63.16%	-28.49%	+336.84%	-31.15%	+531.58%	-38.16%	-15.79%
Average	**-12.72%**	**+20.81%**	**-21.22%**	**+76.80%**	**-24.86%**	**+142.44%**	**-31.79%**	**-5.73%**

Table 3 depicts the benefit of applying the improving transformations described in Sections 4 and 5. Only some of the improving transformations applied

[4] 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 [9]. 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.

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 improving 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).

Table 3. The Impact of 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	-2.24%	-35.09%	-37.76%
conv	-8.22%	-43.48%	-52.13%
copy8	-1.84%	-60.39%	-63.13%
fft	0.00%	-8.69%	-8.69%
fir	0.00%	-48.42%	-48.42%
fir_no	-0.03%	-31.37%	-31.37%
fire	-7.44%	0.00%	-32.31%
iir	0.00%	-19.61%	-19.61%
inverse8	-1.64%	-53.80%	-56.23%
jpegdct	0.00%	0.00%	0.00%
lms	0.00%	-8.33%	-8.33%
scale8	-3.79%	-16.92%	-17.52%
sumabsd	-23.11%	0.00%	-51.70%
trellis	-8.75%	-7.36%	-20.16%
vec_mpy	0.00%	-38.16%	-38.16%
Average	**-3.83%**	**-25.34%**	**-32.97%**

The authors 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 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.

9 Conclusion

This paper described strategies for generating code and utilizing improving transformations to exploit a ZOLB. The authors found that many conventional improving transformations used in optimizing compilers had significant effects on 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 and extraction, loop collapsing, and loop interchange. The authors 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 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, CA (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 (1997).
4. Lucent Technologies: DSP16000 C Compiler User Guide (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 (1997).
8. Wang, Y.: Interprocedural Optimizations for Embedded Systems, Masters Project, Florida State University, Tallahassee, FL (1999).
9. Whalley, D.: DSP16000 C OPtimizer Overview and Rationale, Lucent Technologies, Allentown, PA (1998).
10. Lucent Technologies.: DSP16000 LuxWorks Debugger (1997).
11. Lucent Technologies.: DSP16000 Assembly Language User Guide (1997).
12. 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).
13. Bacon, D., Graham, S., Sharp, O.: Compiler Transformations for High-Performance Computing, ACM Computing Surveys, Volume 26 Number 4, 345–420 (1994).