SHIM Optimization: Elimination Of Unstructured Loops

Ravindra Babu Ganapathi
Department of Computer Science
Columbia University
New York, USA
rg2547@cs.columbia.edu

Stephen A. Edwards
Department of Computer Science
Columbia University
New York, USA
sedwards@cs.columbia.edu

Abstract

The SHIM compiler [4] for IBM CELL processor [1] generates distinct code for the two processing units, PPE(Power Processor Element) and SPE(Synergistic Processor Elements). The SPE is specialized to give high throughput with computation intensive application operating on dense data. We propose mechanism to tune the code generated by the SHIM compiler to enable optimizing compilers to generate structured code.

Although, the discussion here is related to optimizing SHIM IR(Intermediate Representation) code, the techniques discussed here can be incorporated into compilers to convert unstructured loops consisting of goto statements to structured loops such as while and do-while statements to ease back end compiler optimizations.

Our research based SHIM compiler takes the code written in SHIM language and performs various static analysis and finally transforms it into C code. This generated code is compiled to binary using standard compilers available for IBM cell processor such as GCC and IBM XL compiler [2].

Keywords: goto, while, SHIM, IBM, XL, SIMD

1. Introduction

The two possible ways to gain performance in a multicore processor such as CELL without change in algorithm is through task level parallelism and Data level parallelism. Further, the task level parallelism is achieved by multithreading and data level parallelism is through code vectorization. The discussions hence forth here is limited to data level parallelism through code vectorization using streaming SIMD instructions.

The terms *data parallelization*, *auto-vectorization* and *converting into SIMD code* are used interchangeably here. The various steps involved in data parallelization are analyzing the data flow, grouping sequential data elements and executing those using SIMD instructions. The data parallelization can be achieved either by manual optimization of the code using SIMD instructions or auto-vectorization by compiler. However, the manual optimization requires expert programming skills and in cases where the programmer lacks the required skill results in poor performance. Hence, our approach here is to improve the compiler infrastructure by implementing optimization techniques in compiler so that the existing code performs better by just recompiling the code.

The state of the art IBM XL compiler does auto-vectorization where ever possible by analyzing data flow and packing the data as required and converts the instructions into SIMD instructions. However, there are cases when the code is unstructured and the compiler fails to identify the available data parallelism inherent in the code. The following section discusses on analyzing the code to find one such case where loops are represented as goto statements and we propose method to convert them into structured loops.

2. Problem Analysis

The sample code used in our analysis is JPEG decoder code [6] written in SHIM(C like syntax). Let us consider a small snippet of code extracted from IDCT(Inverse Discrete Cosine Transform) function listed in Figure 1. In this code snippet, line 3 and 4 are idle candidate for auto-vectorization where each input buffer element is left shifted by S_BITS and stored in output buffer. However, due to the complex code analysis and transformation carried out by the SHIM compiler, the final C code generated is as listed in Figure 2.

In the SHIM output listing, the code between lines 5 and 15 corresponds to that of the SHIM code in line 3 and 4 listed in Figure 1. However, the while loop structure is transformed and is represented by a set of goto statements, one forward goto(line 6) to check the loop condition and the other backward goto(14, 15) when the loop condition is

```
1. S_BITS=3
2. for (k = 0; k < 8; k++) {
3. for (l = 0; l < 8; l++)
4. output[8 * k + l] = (input[k*8+l]<< S_BITS);
5. foo(output, 8 * k);
6. }
```

Figure 1: *SHIM code*

```
1. S_BITS = 3;
2. k = 0;
3. goto _continue6;
4. _while5: ;
5. l = 0;
6. goto _continue4;
7. _while3: ;
8. _tmp2 = 8 * k;
9. index = _tmp2 + l;
10. output[index] = (input)[index] << S_BITS;
11. l = l + 1;
12. _continue4: ;
13. _tmp1 = l < 8;
14. if (_tmp1)
15. goto _while3;
16. _tmp3 = 8 * k;
17. _func_idct_1d(&output, _tmp3);
18. k = k + 1;
19. _continue6: ;
20. _tmp0 = k < 8;
21. if (_tmp0)
22. goto _while5;
```

Figure 2: *SHIM output C code*

true. The problem with this code is the loops are not easily inferred resulting in sub-optimal binary generated by the optimizing C compiler.

The details of the assembly code listed in Figure 3 are not essential for our discussion. However, to understand the underlying optimization problem, a trimmed down version of the code is shown here which is obtained by compiling the SHIM output code using IBM XL compiler. A few aspects to notice in this code listing is that the loop iterates for 8 times with line 1 initializing the loop variable, line 7 contains the comparison to exit from the loop, line 8 is the branch instruction to exit the loop and line 23 is an unconditional branch to the start of the loop. This is identical to the behavior of the code sequence found in the original source listing. Hence, it is clear that the compiler failed to unroll the loop and optimize the code for data parallelization.

Our first attempt to analyze the problem to enable compiler optimization is through manual tweaking of the code. This involves manually identifying the loops in code listed

```
1. il $2,0
2. shli $4,$40,3
3. brnz $3,.LC__116
4. or $3,$42,$42
5. or $5,$41,$41
6. .LC__158:
7. cgti $6,$2,7
8. brnz $6,.LC__115
9. ai $2,$2,1
10. lqd $6,32($5)
11. hbrr .LC__647, .LC__158
12. lqd $7,160($3)
13. cwd $8,0($3)
14. rotqby $6,$6,$5
15. ai $5,$5,2
16. rotmai $6,$6,−16
17. shli $6,$6,3
18. shufb $6,$6,$7,$8
19. nop $12
20. stqd $6,160($3)
21. ai $3,$3,4
22. .LC__647:
23. br .LC__158
24. .LC__115:
25. ai $3,$1,160
26. hbrr .LC__648,_func_idct_1d
```

Figure 3: *Assembly code generated by IBM XL compiler*

```
1. S_BITS = 3;
2. k = 0;
3. while(k<8)
4. {
5. l = 0;
6. while(l < 8)
7. {
8. _tmp2 = 8 * k;
9. index = _tmp2 + l;
10. output[index] = (input)[index] << S_BITS;
11. l = l + 1;
12. }
13. _tmp3 = 8 * k;
14. _func_idct_1d(&output, _tmp3);
15. k = k + 1;
16. }
```

Figure 4: *Manual optimized code*

in Figure 2 and replace the goto statements with structured while loop. The structured code enables the IBM XL compiler to analyze the loops, unroll them and parallelize them with SIMD instructions.

The code listed in Figure 4 is manual optimized version of the code in Figure 2 with the goto's replaced by struc-

```
cond_continue4 = 1;
do {
if(!cond_continue4)
{
_tmp2 = 8 * k;
index = _tmp2 + l;
Y[index] = (input.linear)[index] << S_BITS;
l = l + 1;
}
_continue4: cond_continue4 = 0;
_tmp1 = l < 8;
}while(_tmp1);
```

Figure 5: *Resulting code from Systematic* ***goto*** *elimination*

tured while loops and removing all the redundant labels and dead code. In the assembly code generated for this structured code the inner loop is completely unrolled and the code is converted into SIMD operations. The performance improvement observed here for a single thread code decoding JPEG image is over 20

3. Related Work

Our first attempt to optimize the code was through existing research available on goto elimination strategy published in the paper *Taming control flow: a structured approach to eliminating goto statements* [5]. They propose a structured approach to eliminate goto statements and hence by following the steps stated in [5] we derive the final code listed in figure 5. However, the IBM XL compiler fails to optimize this code due to the conditional statement embedded in the structured loop.

4. Algorithm to convert unstructured loops to structured loops

The solution we propose here is based on certain heuristics to infer loops with goto statements and convert them into structured while loops. Our solution does not optimize code with loops consisting of goto statements that are not properly nested or consisting of goto statements that jump across loops. This simplifies the problem and we optimize only the case where simple while loops are represented in the form of goto statements.

The first step involved is to build the control flow graph(CFG) for the IR code or any other code to be optimized, in our case we build the graph for the SHIM IR code. Further, the loops can be inferred by identifying backtracking edges in CFG to find patterns such as those listed in Figure 6. We store these identified backtracking edges from the CFG in a list, where each element of the list is an

```
Label:
...
...
...
...
...
if (condition) goto Label;
```

Figure 6*a*: ***do*−*while*** *loop represented* ***using goto***

```
goto Label1;
...
...
Label2:
...
...
...
Label1:
...
...
if(condition) goto Label2;
```

Figure 6*b*: ***while*** *loop represented* ***using goto***

```
1. eliminate_goto(back_edges)
2. Begin
3. for each (Head,Tail) in back_edges
4. Begin
5. if (dom(Tail) = Head)
6. construct_do_while(Head, Tail);
7. else if(check_goto(dom(Tail)))
8. verify_and_construct_while(Head, Tail);
10. End
11. End
```

Figure 7: *Algorithm to eliminate* ***goto*** *statements*

ordered pair with start(Tail) and end(Head) of the directed edge. The pseudo code for converting goto statement to structured while loops is shown in Figure 7. The functions used here are explained in the following sections.

The function eliminiate_goto iterates through each element in the list of backtracking edges(back_edges) and the code corresponding to the Head and Tail of the edge are analyzed. If the edge is inferred as a valid while or do-while loop without any conflict then the unstructured goto's are converted to structured loops.

In the function eliminiate_goto , dom(node) returns the immediate dominator [3] for a given node. The algorithm for finding dominator is explained in section 9.6 of [3]. If the return value by dom in line 5 is "Head" then the pattern found is a perfect do...while loop. Hence, the con-

```
1. verify_and_construct_while(Head, Tail)
2. Begin
3. //node corresponding to next statement in code sequence
4. top:=next_node(dom(Tail));
5. //get the condition expression in tail node
6. cond:=getCond(Tail);
7. //get the target label of the goto statement at (dom(Tail))
8. target:=getLabel(dom(Tail));
9. while(top != Head)
10. Begin
11. if(isvalid_stmt(top))
12. return;
13. else
14. next_node(top);
15. End
16. while(top != target)
17. Begin
18. if(getExpr(temp) related to cond)
19. return;
20. else
21. next_node(top);
22. End
23. //node of previous statement in code sequence
24. temp:=prev_node(Tail);
25. while(temp != top)
26. Begin
27. if(getExpr(temp) related to cond)
28. cond:=merge(cond, expr)
29. prev_node(temp);
30. End
31. construct_while(dom(Tail), Tail, cond);
32. End
```

Figure 8: *Algorithm to infer* ***and*** *consturct* ***while*** *loops*

```
1. S_BITS = 3;
2. k = 0;
3. while (k < 8)
4. {
5. _while5: ;
6. l = 0;
7. while (l < 8)
8. {
9. _while3: ;
10. _tmp2 = 8 * k;
11. index = _tmp2 + l;
12. output[index] = (input)[index] << S_BITS;
13. l = l + 1;
14. _continue4: ;
15. _tmp1 = l < 8;
16. }//while (l < 8)
17. _tmp3 = 8 * k;
18. _func_idct_1d(&Y, _tmp3);
19. k = k + 1;
20. _continue6: ;
21. _tmp0 = k < 8;
22. }//while (k < 8)
```

Figure 9: *New SHIM generated code*

struct_do_while routine is called. This routine contains simple code to replace the "Head" node and the "Tail" node with the start and end of the do...while statement. Further, the following condition at line 7 calls the routine check_goto(node) passing dom(Tail) as an argument, which checks if the dom(Tail) is an unconditional goto statements, if yes, then the routine verify_and_construct_while is called.

The routine verify_and_construct_while checks if the sequence of statements between the dom(Tail) and Tail of the backtracking edge forms a valid while statement, if the check is positive then the goto statements are replaced with the structured while loop. The algorithm is as listed in Figure 8, the first while loop starting at line 9 checks if there is any valid executable code other than labels and dead code, if one such statement found then the inferred code is not a valid while loop and hence exits without changing the code. Further, the second while loop in the code beginning at line 16 checks if the condition statement for the inferred while loop is dependent on any of the instruction before the target of the forward unconditional goto statement. If yes, then the identified goto combination might not be valid while loop and hence returns. Finally, the last while loop beginning at line 25 walks backwards starting from the Tail node to identify the expressions that build the conditional statement stored in cond. If any expression is found then that expression is merged with the existing conditional statement forming a new expression. This results in the conditional being complete statement without any dependency on any other statement within the Head and Tail node. This completes the process of identification of valid while statement and the condition on which the while loop iterates.

The final part of the code in figure 8 calls construct_while with the arguments dom(Tail), the goto statement which is the beginning of the inferred while loop, the Tail node, which is the backtracking conditional goto denoting the end of the while loop and the cond, the conditional statement on which the inferred while loop iterates. This is a simple function which replaces the unconditional goto statement at dom(Tail) with the start of while loop containing the condition cond. The Head node is replaced with the end of while statement and thus the construction of the inferred while loop is complete. The dead code related to the cond located in the code can be ignored since these statements will be eventually removed by any optimizing compiler. In our case, the IBM XL compiler does good job of cleaning up the dead code without any performance impli-

cations.

The final code obtained after applying the algorithm to eliminate goto to the source code in Figure 2 is as listed in figure 9. This optimized code performs on par with the manually optimized code in Figure 4. The statements in line 15 and 21 are dead code and can be removed by analyzing the CFG but in our case we pass on this task to the optimizing compiler that compiles SHIM output code to executable binary.

5. Conclusion and Future work

Here, we proposed solution to eliminate goto statements by inferring loops by analyzing CFG. Our current tests are limited to JPEG decoder which gave a performance boost of over 20 percent, our future work in this respect involves testing with other applications and measuring the performance improvement.

Further, we have other issues which limit auto-vectorization such as conditional statements within the loops which hinders analyzing the data flow in the program. This can be optimized manually by using combination of certain comparison and logical operations available in SIMD instruction set. However, this requires expert programming skills and is a slow process. Additionally, this is very complex to implement in compilers and will be good research work to pursue as a follow on work in data parallelism.

Although, our current focus is on structuring the loops to enhance data parallelism, during our tests we found that the thread overhead is significant for multi-threaded JPEG decoding. Hence, one of the performance challenges to deal with is optimizing these threads overhead. Thus, another area to focus on is optimizing task level parallelism in SHIM.

References

[1] The cell project at ibm research. http://www.research.ibm.com/cell/.

[2] Ibm xl c/c++ for multicore acceleration for linux on x86 systems, v9.0 delivers cell broadband engine architecture application development capability. http://www-01.ibm.com/common/ssi/rep_ca/2/897/ENUS207-252/index.html.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] Stephen A. Edwards. Shim: A language for hardware/-software integration. December 2004.

[5] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *In Proceedings of 1994 IEEE International Conference on Computer Languages*, pages 229–240. IEEE Computer Society Press, 1994.

[6] Nalini Vasudevan and Stephen A. Edwards. A jpeg decoder in shim. (CUCS–048–06), December 2006.