IEIE Transactions on Smart Processing and Computing, vol. 4, no. 6, December 2015
http://dx.doi.org/10.5573/IEIESPC.2015.4.6.450

Getting Feedback on a Compiler's Optimization Decisions, Enabling More Code-Optimization Opportunities

Gyeong Il Min[1], Sewon Park[1], Miseon Han[2], and Seon Wook Kim[2]

[1] School of Electrical Engineering, Korea University / Seoul, South Korea {gyeong9m, psw9704}@korea.ac.kr
[2] Compiler and Microarchitecture Laboratory, School of Electrical and Computer Engineering, College of Engineering, Korea University / Seoul, South Korea {mesunyyam, seon}@korea.ac.kr

***** Corresponding Author: Seon Wook Kim

Received December 17, 2015; Accepted December 23, 2015; Published December 31, 2015

** Short Paper*

***Abstract*:** Short execution time is the major performance factor for computer systems. This performance factor is directly determined by code quality, which is influenced by the compiler's optimizations. However, a compiler has limitations when optimizing source code due to insufficient information. Thus, if programmers can learn the reasons why a compiler fails to apply optimizations, they can rewrite code that is more easily understood by the compiler, and thus improve performance. In this paper, we propose a compiler that provides a programmer with reasons for failed optimization and recognizes programmer's additional information to obtain better optimization. As a result, we obtain performance improvement, i.e., reducing execution time and code size, by taking advantage of additional optimization opportunities.

***Keywords*:** LLVM, Optimization, Performance, Code quality, Execution time

1. Introduction

In most computer systems, such as mobile devices, embedded systems, PCs, and server platforms, execution time is the most important factor in performance. Code quality determines the performance factors, and is greatly influenced by the compiler's optimizations. For example, a compiler can improve program performance by doing inline [1] to remove the overhead of a function call.

However, code optimization has limitations due to insufficient information. For example, the Low Level Virtual Machine (LLVM) [2] compiler analyzes alias information in the loop invariant code motion (LICM) [3] pass. When the compiler meets a loop-invariant variable referred to by a pointer during LICM optimization, the compiler shows a limited optimization capability because of memory ambiguity. In this case, if programmers know the reason for optimization failure, they can rewrite the source code to improve performance.

In this paper, we propose a compiler that provides programmers with optimization results and recognizes information provided by programmer in order to generate more optimized code. Effectiveness of feedback on a compiler's optimization was demonstrated by another paper that generates optimization results and recommends code modification on the Racket compiler [4]. However, we implement feedback on the LLVM compiler, which is more generalized compiler than Racket compiler. Furthermore, we propose a more convenient strategy to obtain additional optimization opportunities by not modifying source code but attaching supplementary information to the code.

The main contributions of this paper are as follows.

- We modified the LLVM compiler to provide programmers with detailed optimization reports. With these reports, programmers can find optimizable parts of their code.
- We modified the source code of SPEC CPU 2006 [5] to obtain additional optimization opportunities. As a result, we can achieve speedup in modified code. This performance improvement demonstrates the effectiveness of optimization reports.
- We also modified the LLVM compiler to recognize alias information provided by programmers. Using our compiler, we can obtain performance improvement more conveniently, i.e., not rewriting source code but just inserting additional information in the code.

The rest of this paper is organized as follows. Section 2 describes how we implemented the optimization reports. In Section 3, we analyze the optimization reports for SPEC CPU 2006 and rewrite code to obtain additional optimization opportunities. Section 4 describes how we modified the compiler to recognize supplementary information, and we evaluate it by comparing the original assembly against the code with information added. Finally, the conclusion is offered in Section 5.

2. Implementation of Optimization Reports

To produce optimization reports, we modified the LLVM compiler using the Rpass [6] infrastructure in Clang [7]. The original LLVM compiler prints out optimization results only from inline, loop unroll, and loop vectorization passes. Thus, analyzing optimization reports using the original LLVM compiler has limitations in finding additional optimization opportunities. Therefore, we modified the original optimization reports to be more specific, and implemented reports for other optimization passes by the LLVM.

Furthermore, we implemented the optimization reports from LLVM optimization passes. Because LLVM optimization passes include both front-end and back-end optimizations, our compiler prints out both optimization results, i.e., target-independent and target-specific optimization results.

Fig. 1 shows examples of optimization reports. Our optimization reports contain optimization results and the corresponding locations in the source code. Optimization results are classified as *success*, *failure*, and *analysis*. *Success* contains the results of successful optimizations, *failure* explains why optimization cannot be done, and *analysis* explains the compiler's decisions as to whether or not to transform the code and those reasons in detail. Analyzing these reports, programmers can find out how the compiler optimizes their code and how they can obtain additional optimization opportunities.

3. Optimization Opportunity Analysis and Results of Rewriting Source Code

This section presents optimization opportunities by following these procedures: analyzing optimization reports, rewriting source code according to the analyzed information, and comparing assembly code in the original source to that of rewritten code.

We chose the SPEC CPU 2006 benchmark as our test benchmark, and got optimization reports on the benchmark using our compiler, which was modified as described in Section 2. Also, we used Vtune [8] to find the hotspot of the benchmark and analyzed its optimization reports.

3.1 Mcf

Figs. 2 and 3 represent optimization reports for the functions *price_out_impl* and *primal_bea_mpp*, which are hotspots of *mcf* (hereafter, the *price_out_impl* function is

```
a)
bzip2.c:904:4: remark: showFileNames inlined
into crcError [-Rpass=inline]
   showFileNames();
b)
bzip2.c:906:4: remark: cleanUpAndFail will not
be inlined into crcError [-Rpass-
missed=inline]
   cleanUpAndFail( 2 );
c)
bzip2.c:906:4: remark: cleanUpAndFail too
costly to inline (cost=5, threshold=1) [-
Rpass-analysis=inline]
   cleanUpAndFail( 2 );
```

Fig. 1. Examples of optimization reports for (a) success, (b) failure, and (c) analysis.

```
implicit.c:293:53: remark: Unable to hoist
load(potential) instruction because of
insufficient alias information
[-Rpass-missed=licm]
   red_cost = arc_cost - tail->potential
              + head->potential;

implicit.c:297:32: remark: Unable to hoist
load(max_residual_new_m) instruction because
of insufficient alias information
[-Rpass-missed=licm]
   if( new_arcs < net->max_residual_new_m )
```

Fig. 2. Optimization reports in price_out_impl (function A).

```
pbeampp.c:165:36: remark: Unable to hoist
load(nr_group) instruction because of
insufficient alias information
[-Rpass-missed=licm]
   for( ; arc < stop_arcs; arc += nr_group )
```

Fig. 3. Optimization reports in primal_bea_mpp (function B).

```
original code
    while (arcin){
       …
       red_cost = arc_cose – tail->potential
                  + head->potential;
       if( red_cose <0 )
          if( new_arcs < net->max_residual_new_m)
            …
    }

modified code
   head_potential = head->potential
   net_max_residual = net_>max_residual_new_m;
   while (arcin){
      …
      red_cost = arc_cose – tail->potential
                 + head_potential;
      if( red_cose <0 )
         if( new_arcs < net_max_residual;)
           …
   }
```

Fig. 4. Original and modified source code of function A.

```
original code
   for( ; arc < stop_arcs; arc += nr_group)
     …

modified code
   long nr_group_non_static = nr_gourp;
   for( ; arc < stop_arcs;
                  arc += nr_group_non_static)
     …
```

Fig. 5. Original and modified source code of function B.

```
original assembly code
  Block 21
    …
  Block 24
   subq (%rbx), %r9
   addq (%r11), %r9
   jns 0x401c0b <Block 34>
  Block 25
   movq 0x20(%rsp), %rax
   cmpq 0x1c0(%rax), %r15
   jnl 0x401bd4 <Block 31>
    …
  Block 35
   movq 0x48(%rbx), %rax
   test %rax, %rax
   jnz 0x401ad0 <Block 21>

modified assembly code
  Block 20
   movq(%rcx), %rdx
   movq 0x20(%rsp), %rcx
   movq 0x1c0(%rcx), %rcx
   movq %rcx, 0x28(%rsp)
  Block 21
    …
  Block 24
   subq (%rbx), %r9
   add %rdx, %r9
   js 0x401b15 <Block 26>
  Block 26
   cmpq 0x28(%rsp), %r15
   jnl 0x401bf4 <Block 32>
    …
  Block 37
   movq 0x48(%rbx), %rax
   test %rax, %rax
   jnz 0x401ae0 <Block 21>
```

Fig. 6. Assembly code of function A.

denoted as function A, and *primal_bea_mpp* is denoted as function B). As shown in these figures, the LICM pass cannot perform optimization due to insufficient alias information. In function A, structure pointers cause memory ambiguity. In function B, *nr_group* is declared a static variable, and the LLVM compiler has difficulty analyzing alias information of static variables. In order to provide additional alias information, we rewrote the code as shown in Figs. 4 and 5.

Figs. 6 and 7 show original and modified assembly code. The modified assembly code for functions A and B have fewer memory accesses and mov instructions, respectively, than the original. As a result, we achieved speedup of 1.22% in function A and 8.82% in function B. In total, we obtained a 3.30% speed increase.

```
original assembly code
  Block 22
    …
  Block 30
   movq 0x204306(%rip), %rdx
   mov %rdx, %rax
   shl $0x6, %rax
   add %rax, %rbx
   cmp %rcx, %rbx
   jb 0x402940 <Block 22>

modified assembly code
  Block 13
   movq 0x20447e(%rip), %rdi
   jmp 0x40294b <Block 19>
  Block 22
    …
  Block 30
   add %rdi, %rdx
   cmp %r10, %rdx
   jb 0x402980 <Block 22>
```

Fig. 7. Assembly code of function B.

```
mv-search.c:419:18: remark: Unable to hoist
load(Pelyline_11) instruction because of
insufficient alias information
[-Rpass-missed=licm]
refptr = PelYline_11 (ref_pic, abs_y++, abs_x,
                      img_height, img_width);
```

Fig. 8. Optimization reports in *h264ref*.

```
original code
for (pos = 0; pos < max_pos; pos++) {
   if …
      Pelyline_11 = FastLine16Y_11;
   else
      Pelyline_11 = UMVLine16Y_11;
   for (blky = 0; blky <4; blky++)
      for (y = 0; y < 4; y++)
         refpter = Pelyline_11 (…)
}

modified code
for (pos = 0; pos < max_pos; pos++) {
   if …
      for (blky = 0; blky <4; blky++)
         for (y = 0; y < 4; y++)
            refpter = FastLine16Y _11 (…)
   else
      for (blky = 0; blky <4; blky++)
         for (y = 0; y < 4; y++)
            refpter = UMVLine16Y _11 (…)
}
```

Fig. 9. Original and modified source code relevant to Fig. 8.

3.2 h264ref

Fig. 8 shows optimization reports in the hotspot *h264ref*. The reports indicate that the LICM pass failed because of insufficient alias information due to a function pointer.

Thus, we rewrote source code as shown in Fig. 9 to make the compiler analyze alias information of the function pointer. Thus, as shown in Fig. 10, overhead to find a function address is reduced, i.e., the callq

```
original assembly code
   mov %r11d, %edx
   movq 0x98(%rsp), %rcx
   movq 0xa0(%rsp), %r8
   mov %r11d, %r15d
   callq 0x24e162(%rip)

modified assembly code
   mov %r9d, %edx
   movq 0xa8(%rsp), %rcx
   movq 0xb0(%rsp), %r8
   callq 0x4613e0 <FastLine16Y_11>
```

Fig. 10. Assembly code for Fig. 9.

```
for (i = 50; i < 100; i++)
   #pragma optmz noalias(A)
   for (j = 0; j <50; j++)
       A[j] = A[i];
```

Fig. 11. Usage of the new pragma.

instruction's operand is changed from memory location to offset. As a result, we achieved a 2.05% speed increase.

4. More Optimized Code with Additional Information

In order to achieve further optimization opportunities, we rewrote the source code from Section 3. However, it is more convenient for programmers to attach supplementary information to the source code. Thus, we modified the compiler to parse added information. In this section, we describe how we modified the compiler and evaluated performance improvement by comparing the original assembly code against code with information inserted.

4.1 Compiler Modification

To provide additional information to our compiler, we made a new pragma and modified our compiler to parse it:

#pragma *optmz* '*type*' ('*parameter*')

optmz represents the existence of programmer information, *type* represents the information provided, and *parameter* represents variables containing *type*'s information. Fig. 11 shows the pragma usage. In this example, the pragma means that variable 'A' has no alias in the following loop.

In order to perform optimization using the pragma, we modified our compiler to process it. When the pragma is parsed, the front end builds new metadata [9] and adds the metadata to intermediate representatives (IRs). Then, the optimizer decides whether to optimize the IRs or not based on the metadata.

4.2 Evaluation in Example Code

In order to evaluate optimization opportunities with additional information, we compiled original code for Fig. 11 with our modified compiler.

```
(a)
    movl  (%rsi), %edi
    movl  %edi, -128(%rsp,%rax,4)
    incq  %rax
    cmpq  $50, %rax
    jne .LBB0_5
(b)
    movl  %ecx, 64(%rsp,%rsi,4)
    incq  %rsi
    cmpq  $2, %rsi
    jne .LBB0_3
```

Fig. 12. Assembly code of the inner loop in Fig. 11: (a) before, and (b) after providing alias information.

In the code from Fig. 11, loading A[i] is invariant in the inner loop because the range of i and j has no intersection. The LICM pass, however, cannot hoist the load instruction due to poor alias analysis without the pragma. Therefore, the LICM pass can achieve additional opportunities if the pragma is inserted, as shown in Fig. 11.

Fig. 12 represents assembly code of the inner loop in Fig. 11. As shown in Fig. 12, one mov instruction is removed after the pragma is inserted. To evaluate performance improvement, we executed the nested loop 10 million times in the original code and modified code and achieved a speedup of 46.18%. This performance improvement shows that programmers can efficiently obtain performance improvement with our pragma.

5. Conclusion

In this paper, we modified the LLVM compiler to generate optimization reports and found additional optimization opportunities by analyzing the reports. In order to figure out the opportunities, we rewrote source code for the SPEC CPU 2006 so that the compiler generated more optimized executable code. As a result, we achieved a speedup of 3.30% in *mcf* and 2.03% in *h264ref*. These performance improvements showed the effectiveness of optimization reports.

Furthermore, we proposed a more convenient strategy to achieve additional optimization opportunities by attaching supplementary information to source code using a pragma. We evaluated this strategy using example code and obtained a 46.18% speedup by inserting the pragma.

For future work, we are going to compile Android source code in order to find additional optimization opportunities. Then, we will modify Android source code by referencing our optimization reports. We expect that we can achieve performance improvement on the Android platform.

References

[1] P. P. Chang, et al., "Inline function expansion for compiling C programs," *in Proc. of PLDI 1989*, pp. 246-257, Jul. 1989. Article (CrossRef Link)

[2] The LLVM Compiler Infrastructure. http://llvm.org/

[3] David F. Bacon, et al., "Compiler transformations

for high-performance computing," *ACM Comput. Surv.*, Vol. 26, pp. 345-420, Dec. 1994. Article (CrossRef Link)
[4] Vincent St-Amour, et al., "Optimization coaching: optimizers learn to communicate with programmers," in Proc. Of OOPSLA 2012, Oct. 2012. Article (CrossRefLink)
[5] John L. Henning, "SPEC CPU2006 benchmark descriptions," SIGARCH Comput. Archit. News, Vol. 34, pp. 1-17, Sep. 2006. Article (CrossRef Link)
[6] The Rpass Infrastructure. Article (CrossRef Link)
[7] Lattner, Chris, "LLVM and Clang: Next generation compiler technology," *The BSD Conference* 2008, May. 2008. Article (CrossRef Link)
[8] Reinders, James, "VTune performance analyzer essentials," Intel Press, 2005. Article (CrossRef Link)
[9] Hal Finkel, "Intrinsics, Metadata and Attributes: Now, more than ever!," 2014 LLVM Developers' Meeting. Article (CrossRef Link)

Gyeong Il Min is currently a student in a B.E. course in Electrical Engineering at Korea University, Seoul, Rep. of Korea. He will receive his BE in February 2016. His research interests include compiler support and performance analysis in Android mobile systems.

Sewon Park is currently a student in a BE course in Electrical Engineering at Korea University, Seoul, Rep. of Korea. He will receive his B.E. in February 2016. His research interests include compiler support and performance analysis in Android mobile systems.

Miseon Han received her B.E. in Electrical Engineering from Korea University, Seoul, Rep. of Korea, in February 2012. Currently, she is a PhD candidate at the same department. Her research interests include compiler support, microarchitecture, and memory designs.

Seon Wook Kim received a BSc in Electronics and Computer Engineering from Korea University, Seoul, Rep. of Korea, in 1988. He received an MSc in Electrical Engineering from Ohio State University, Columbus, Ohio, USA, in 1990, and a Ph.D. in Electrical and Computer Engineering from Pu-rdue University, West Lafayette, Indiana, USA, in 2001. He was a senior researcher at the Agency for Defense Development from 1990 to 1995, and a staff software engineer at Inter/KSL from 2001 to 2002. Currently, he is a professor with the School of Electrical and Computer Engineering at Korea University. His research interests include compiler construction, microarchitecture, and SoC design. He is a senior member of ACM and IEEE.