Adding Operator Strength Reduction to LLVM

Brian N. West

October 20, 2011

Chapter 1

Introduction

As part of the Platform-Aware Compilation Environment (PACE) Project[1], Operator Strength Reduction (OSR) [3] was added to the LLVM Compiler Infrastructure Project (LLVM) as a optimization pass.

The goal of the PACE Project is to construct a portable compiler that produces code capable of achieving high levels of performance on new architectures. The multi-tiered architecture of the PACE compiler included high level and low level compilation components. LLVM was the choice for the low level optimizer and code generator.

The high-level components of the PACE compiler include a Polyhedral-based optimization pass. This optimization pass produces many opportunities for strength reduction. LLVM lacked a strength-reduction optimization pass[2], so this is the motivation for adding the OSR optimization to LLVM.

[1]This work is funded by the Defense Advanced Research Projects Agency through AFRL Contract FA8650-09-C-7915. The opinions and findings in this document do not necessarily reflect the views of either the United States Government or Rice University.

[2]LLVM advertises a strength reduction optimization, which is actually part of the code generation pass in its backend and not the stand alone optimization that PACE needed.

Chapter 2

Operator Strength Reduction

Operator strength reduction is a well known compiler optimization dating back to the very first production compilers [2]. The basic idea is that instructions that require many cycles to execute are replaced by those that require fewer cycles to complete. This optimization can be seen in the address calculation instructions for an array within a loop. The following C code sums the array elements within a loop:

```
float sum, A[N];
sum = 0.0;
for (i=0; i<N; i++) {
    sum += A[i];
}
```

Lowered to explicitly expose the address calculations, the code becomes:

```
float sum, A[N];
char *a_addr = &A;
char *a_i_address;
sum = 0.0;
i = 0;
while ( i < N ) {
    int offset = sizeof(float) * i;
    a_i_address = a_addr + offset;
    sum = sum + *((float *)a_i_address);
}
```

The address calculation of **A[i]** has a multiplication instruction that for many architectures requires multiple cycles, whereas an addition usually completes in a single cycle. The strength-reduction optimization replaces the above multiplication with an add instruction. After strength reduction, the code becomes:

```
float sum, A[N];
int offset;
char *a_i_address = &A;
sum = 0.0;
i = 0;
while ( i < N ) {
    sum = sum + *((float *)a_i_address);
    a_i_address += sizeof(float);
    i++;
}
```

The code introduced a new variable, **a_i_address**, which is a pointer into the **A** array and is incremented each loop iteration by the size of the array element.

The loop variable **i** after strength reduction is only needed for the end of loop test. An additional optimization, Linear Function Test Replacement (LFTR), can be performed to replace the loop test with a test using the variable that strength reduction added. After LFTR, the variable **i** then becomes useless and can be deleted.

Chapter 3

The OSR Algorithm

The OSR optimization of Cooper *et al.* [3] was developed as a replacement for the traditional strength-reduction optimization as described by Allen *et al.* [1], which was both hard to understand and hard to implement.

OSR as outlined in the paper is performed on the intermediate representation of the program in Static Single Assignment [4] (SSA) form. LLVM uses the SSA form exclusively for its optimization passes. Thus, OSR was a good fit for PACE and LLVM.

OSR discovers induction variables in the program by finding loops in the SSA graph. The SSA Graph contains nodes that map to the definitions and uses of values and contains edges that connect the definitions to the uses. To be more precise, the loops are Strongly Connected Components (SCCs).

From the example code in Chapter 2, the associated SSA form is roughly:

```
float sum, A[N];
char *a_addr = &A;
char *a_i_address;
sum0 = 0.0;
i0 = 0;
while ( true ) {
    sum01 = phi(sum0, sum2)
    i1 = phi(i0, i2)
    int offset = sizeof(float) * i1;
    char *a_i_address = a_addr + offset;
    sum2 = sum1 + *((float *)a_i_address);
    i2 = i1 + 1;
}
```

Both variables **i** and **sum** have SCCs in the SSA graph. For example, variable **i**'s SCC contains the definitions of $\mathbf{i_1}$ and $\mathbf{i_2}$. The definition of $\mathbf{i_1}$ uses $\mathbf{i_2}$ and the definition of $\mathbf{i_2}$ uses $\mathbf{i_1}$. Figure 3.1 displays the SCC in the SSA graph for the induction variable **i**.

Of these two variables, OSR will recognize variable **i** as an induction variable, since **i** changes in a uniform manner across each loop iteration, while **sum** does not. OSR uses the induction variables that it finds to create new induction variables. In this example, OSR finds the expression, $sizeof(float) * i_1$, to be a candidate for strength reduction. A new induction variable will be created that is incremented by $sizeof(float)$ (compile time constant). All uses of $sizeof(float) * i_1$ will replaced by uses of the new induction variable. The new induction variable will also have its own SCC in the SSA graph, which can be used to create additional induction variables.

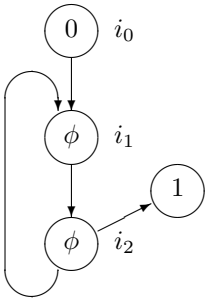

Figure 3.1: SSA Graph for Induction Variable **i**

Chapter 4

Strength Reduction on 64-bit Architectures

OSR was originally developed on a 32-bit architecture. Then, the most common integer size (e.g., C int) and address size were both the same size, 32-bits. OSR for LLVM has been developed for a 64-bit architecture, where the most common integer size is 32-bits and the address size is 64-bits. This size difference between common integer and address types displayed latent strength-reduction problems[1].

The crux of the problem is that the 32-bit integer (or smaller) induction variables can overflow. If OSR creates a 64-bit-address induction variable from a 32-bit-integer induction variable, an overflow in the 32-bit-integer arithmetic will not trigger similar behavior in the 64-bit-address arithmetic.

Some example code to display the overflow issue is:

```
int i;
unsigned char j;
float sum, A[256];
j = 0;
sum = 0.0;
for (i=0; i<8000; i++) {
    j += 3;
    sum += A[j];
}
// on loop exit, j should equal (8000*3) % 256 or 192
```

Within the loop, **j** always has an 8-bit value between 0 and 256. Likewise all accesses to array **A** via **A[j]** will be contained within **A**. The overflow behavior of **j** must be preserved while considering strength reduction. Naively adding 3 to a pointer within **A**, as strength reduction is wont to do, would not overflow on the 8-bit boundary, would access **A** outside of its limits, and thusly would not produce correct code.

Looking only at the overflow problems of **j** in the above code understates the problem. The real target for OSR is the address calculation for **A[j]**, which contains an implicit multiplication by 4 (if the size of the float type is 32-bits or 4 bytes). OSR will want to replace the multiplication by 4 with a new induction variable based on **j** which will get incremented by 4 each loop iteration. Adding by 4 gets to the 8-bit overflow boundary 4 times faster. To maintain the overflow characteristics of **j**, the multiplication would logically need to be:

```
offset = (j*4) % (256*4);
```

and the OSR-created induction variable would logically need to be:

$\texttt{offset}_2$ = ($\texttt{offset}_1$+4) % (256*4);

[1]The problem (with 8-bit integers) was first seen in LLVM's test suite, within file: simple_types_constant_folding.cpp.

4.1 Workarounds

An easy solution to this problem is to restrict strength reduction to those values of the same bit width. In this case, OSR would only strength reduce 64-bit integers (e.g., C type long), matching the address size. This decision might limit the number of opportunities, since legacy C code may have more 32-bit integers than 64-bit integers.

An alternative may be to force 32-bit integer (e.g., C type int) variables to use 64-bit integers. This might unfortunately break legacy code, if said code contains any strict assumptions regarding the size of integer types.

Chapter 5

Strength Reduction for ANSI C

ANSI C specifies overflow behavior for its signed and unsigned integer types. For unsigned integers, overflow is required to wrap. For signed integers, overflow is undefined, and the C compiler is free to handle the overflow as it sees fit. A rationale for this divergent behavior is that some embedded architectures need to support saturation instead of wrapping on integer overflow.

The existing strength-reduction optimization in LLVM's backend exploits the ANSI C overflow rule when handling signed 32-bit integers. Since the 32-bit signed integer overflow is undefined, LLVM makes a strong assumption that the overflow cannot happen and that the program is always defined. Since overflow cannot happen, signed 32-bit integer induction variables are used to create 64-bit address induction variables. LLVM does not handle signed 16 and 8-bit integers similarly.

Chapter 6

OSR for LLVM

The OSR paper gave pseudo code for the procedures: **OSR**, **ProcessSCC**, **DFS**, **ClassifyIV**, **RegionConst**, **Replace**, **Reduce**, and **Apply**. The implementation follows the pseudo code.

6.1 Pseudo-Code Infelicities

Due to the nature of pseudo code, some details are missing or implicit, while other times there is a divergence between the pseudo code and what it needs to be. Two cases of the latter were found during implementation.

In procedure **ClassifyIV**, the input SCC is filtered to see if it qualifies as an induction variable. The first test is to see if the operator qualifies and the second is to see if the operands qualify. The ClassifyIV pseudo-code in question is:

```
for each n ∈ SCC
    if n.op ∉ {φ, +, -, COPY}
        SCC is not an induction variable
    else
        for each o ∈ {operands of n}
            if o ∉ SCC and not RegionConstant(o, header)
                SCC is not an induction variable
```

The operator and operand tests in combination are imprecise. For example, the expression:

```
iv + iv
```

would pass the filter when it should not. A more precise filter would use the induction variable formula, as it seen elsewhere in the pseudo code, like:

```
iv ± rc or rc + iv
```

Thus, the modified **ClassifyIV** pseudo-code is:

```
for each n ∈ SCC
    if n.op is φ or COPY
        if all operands are not members of the SCC and are not a region constant
            SCC is not an induction variable
    else if n.op is + or -
        if n is not of the form x ← iv ± rc or x ← iv - rc
            SCC is not an induction variable
    else
        SCC is not an induction variable
```

In procedure **Reduce**, the input SCC is cloned with a slight modification to create a new induction variable. While looking at each operand for a cloned instruction, a test is made to see if the operand definition is part of the SCC, so that **Reduce** can be recursively called using the operand. The test checks to see if the SCC and operand are both induction variables with the same loop header. This test is not precise enough. Not only should the operand and SCC share the same header, but they must also share the same **phi** node in that header. Our implementation created another operand attribute, **sccphi**, to allow this more precise test. The **sccphi** attribute was also needed for LFTR.

The original **Reduce** pseudo-code:

```
if o.header = v.header
    Replace o with Reduce(opcode, o, rc)
```

needs to be replace with:

```
if o.header = v.header and o.sccphi = v.sccphi
    Replace o with Reduce(opcode, o, rc)
```

6.2 Address Calculation in the LLVM IR

The address calculations in the OSR paper are explicit. The address calculations in the LLVM IR are not. For example, an explicit address calculation into a floating point array, **A[i]**, looks like:

```
&A + (i * 4)
```

In LLVM IR, the same calculation looks like:

```
getelementptr float* %A, i64 i
```

The multiplication by 4 (which is the size of a float) and the addition operator are not present in the LLVM IR **getelementptr** instruction, which actually resembles explicit address calculations in ANSI C:

```
(A + i) /* aka A[i] */
```

The LLVM IR **getelementptr** instruction is a higher-level abstraction than what is presented in the OSR paper. Mapping the OSR paper IR instruction to the **getelementptr** instruction is not always straightforward. Given a loop that descends, the induction variable update may look like:

$$i_2 = i_1 - stride;$$

The **getelementptr** instruction performs an implicit add between its operands. To map to **getelementptr**, the second operand needs to be negated before it can be added, e.g.,

```
tmp1 = sub i64 0, %stride
tmp2 = getelementptr float* %A, i64 %tmp1
```

Chapter 7

LFTR for LLVM

After OSR has completed for a function, Linear Function Test Replacement (LFTR) is run. LFTR examines each induction variable (IV) that existed before OSR was run, to see if it is a candidate for replacement. The primary condition for LFTR candidacy is when the only use of the pre-existing IV outside of its SCC is an end-of-loop compare.

OSR records which newly created IV is derived from which other IV. The derived relationship is stored in a derivation tree. The roots of the derivation tree are the pre-existing IVs. Each node of the derived tree saves its associated IV and the transformation needed to get from its parent to itself.

If a LFTR candidate was found, the end-of-loop test is replaced with a new test using an OSR created IV. The OSR IV is selected from one of the leaf nodes of the IV derivation tree. The series of transformation needed to get from the root to the leaf are applied to the existing end-of-loop test. Next the old test in the end-of-loop branch is replaced with a newly created test, leaving the SCC of the pre-existing IV useless and deletable.

Recall that the OSR motivating example from Chapter 1 after strength reduction is:

```
float sum, A[N];
int offset;
char *a_i_address = &A;
sum = 0.0;
i = 0;
while ( i < N ) {
    sum = sum + *((float *)a_i_address);
    a_i_address += sizeof(float);
    i++;
}
```

LFTR will find that pre-existing IV **i** is a candidate. The derivation tree with IV **i** as the root will contain one leaf node for IV **a_i_address**. LFTR will create a new end-of-loop test using **a_i_address** and with it replace the existing test. LFTR will then delete all occurrences of IV **i**. The example then becomes:

```
float sum, A[N];
int offset;
char *a_i_address = &A;
sum = 0.0;
while ( a_i_address < &A[N] ) {
    sum = sum + *((float *)a_i_address);
    a_i_address += sizeof(float);
}
```

The OSR paper describes a LFTR algorithm but gives no pseudo code. What the paper describes and what is implemented are similar. A LFTR algorithm can also be found here [5].

Chapter 8

OSR Limitations

We have observed several limitations with OSR. Some of the limitations could possibly be addressed by using a cost function to be more selective in creating induction variables. How to add a cost function to the current OSR algorithm is an open question.

8.1 Speculation

OSR converts everything that matches its criteria, which may lead to creating induction variables that in turn do not participate in an address calculation. For example, given an induction variable **i**, we could see the following SSA code:

```
         i0 = 0
loop_header:
         i1 = phi(i0, i2)
         = i1 + 3
         i2 = i1 + 1
```

OSR will want to create a new induction variable from the expression **(i+3)** which will look like:

```
         i0 = 0
         iplus30 = 0
loop_header:
         i1 = phi(i0, i2)
         iplus31 = phi(iplus30, iplus32)
         = iplus31
         i2 = i1 + 1
         iplus32 = iplus31 + 1
```

In the loop, one addition **(i_1+3)** gets replaced by another **($iplus3_1$+1)**; thus, there was no strength reduction performed. OSR performs this transformation speculatively in hopes that an address calculation like **A[i+3]** will be reached and reduced.

Note also from the above example that the newly created induction variable **$iplus3_1$** has a lifetime that spans the entire loop instead of potentially being limited to just at the definition and use. Increasing the lifetime increases the register pressure in the loop, which complicates register allocation.

8.2 Pessimize Input Code Shape

The input code may be more optimal before OSR is performed, as compared to afterwords. Consider a simple loop body:

```
A[i] = B[i] + C[i];
```

Unrolling this loop body by a factor of four will yield the new loop body:

```
A[i]   = B[i]   + C[i];
A[i+1] = B[i+1] + C[i+1];
A[i+2] = B[i+3] + C[i+2];
A[i+3] = B[i+3] + C[i+3];
```

OSR for the original loop will create three induction variable for **&A[i]**, **&B[i]**, and **&C[i]**. OSR for the unrolled loop will create three induction variables for expressions **(i+1)**, **(i+2)**, and **(i+3)** and will create twelve induction variables for the addressing of the **A**, **B**, and **C** arrays. This is arguably not the right thing to do. Having fifteen induction variables live across a single loop could lead to spilling during register allocation.

A similar argument could be made for the stencil calculation found in some high performance codes, like the five stencil pattern:

```
A[i][j] = (            A[i-1][j] +
           A[i][j-1] + A[ i ][j] + A[i][j+1] +
                       A[i+1][j]) / 5;
```

A reassociation optimization pass, in both the loop unrolling and the stencil examples, may change the code shape and reduce the number of multiplications via anchoring array referencing around **&A[i]** or **&A[i][j]**. In the loop unrolling example, the reassociation optimization pass may calculate **&A[i+1]** by adding a constant to **&A[i]** versus calculating **i+1**, multiplying by the cell size and then adding the address of **A** to the multiplication product. In this case, OSR has no multiplication to reduce, but will still create a new induction variable for the expression, **&A[i]** plus a constant. Arguably, the reassociation optimization pass produced near optimal code for the expression**&A[i+1]**, which does not need to be strength reduced.

8.3 May Not Reduce Strength

Multiplying by a power of two can be replaced by a shift operator, which should cost the same as the add with which OSR would want to replace it. This is a case where multiplication will not be expensive. This case turns out to be common, since the sizes of intrinsic numerical types tend to be powers of two. Thus, single-dimension numeric arrays likely do not need their strength reduced.

8.4 Move Code Into a Loop

Sometimes OSR creates an induction variable nested deeper than the targeted instruction that it replaces. The following code displays this infelicity:

```
float sum = 0.0, A[N];
for (unsigned long i = 0; i < n; i++) {
    sum += A[i];
}
return A[i-3];
```

OSR will recognize the expression **A[i-3]** as a candidate for strength reduction. OSR will create a new induction variable for **A[i-3]** based on the **i** induction variable within the loop that precedes the expression **A[i-3]**. Thus, the multiplication that is removed is replaced by an addition, which can be executed multiple times in the preceding loop.

OSR will transform the above code to be:

```
float sum = 0.0, A[N];
for (unsigned long i = 0, char *a_i_minus_three_address = &A[-3];
       i < n;
       i++, a_i_minus_three_address += sizeof(float)) {
    sum += A[i];
}
return *((float *)a_i_minus_three_address);
```

Chapter 9

Conclusions

Our implementation of OSR in LLVM follows the algorithm of Cooper *et al.* [3]. Inherent shortcomings introduced into the transformed code suggest that this optimization should only be used in a system that provides feedback to the compiler so that unprofitable transforms can be discarded.

Bibliography

[1] F E Allen, John Cocke, and Ken Kennedy. Reduction of operator strength. In Steven S Muchnick and Neil D Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 3, pages 79–101. Prentice-Hall, 1981.

[2] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The fortran automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, IRE-AIEE-ACM '57 (Western), pages 188–198, New York, NY, USA, 1957. ACM.

[3] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, 23:603–625, September 2001.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991.

[5] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.