Isothermal Speculative Partial Redundancy Elimination

Brent George
bdgeorge@umich.edu
University of Michigan
Ann Arbor, Michigan, United States

Sahil Surapaneni
ssurapan@umich.edu
University of Michigan
Ann Arbor, Michigan, United States

Priya Thanneermalai
tpriya@umich.edu
University of Michigan
Ann Arbor, Michigan, United States

Gretchen Zheng
zhengji@umich.edu
University of Michigan
Ann Arbor, Michigan, United States

Abstract

Partial Redundancy Elimination(PRE) is a standard program optimization that removes redundant computations via Code Motion. Traditional PRE is very conservative and does not account for hot or cold paths in the code. This causes traditional PRE implementations to miss potentially lucrative optimization opportunities. In our final project, we intend to implement a novel and efficient optimization of PRE: Isothermal Speculative Partial Redundancy Elimination(ISPRE), as proposed by R. Nigel Horspool et al [2]. While the classical PRE analysis is performed without any knowledge of the relative frequencies of execution of the different paths, ISPRE takes advantage of the program profiling information and is executed as an approximate technique of the classic code motion method. Our results show speedup over several benchmarks, as well as optimizing test cases that are ignored by traditional PRE.

ACM Reference Format:
Brent George, Sahil Surapaneni, Priya Thanneermalai, and Gretchen Zheng. 2022. Isothermal Speculative Partial Redundancy Elimination. In *Proceedings of University of Michigan EECS 583 Advanced Compilers Fall 2022 (UM EECS 583 F'22).* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Traditional Partial Redundancy Elimination is an optimization that allows for more aggressive use of redundancy elimination in the code.

UM EECS 583 F'22, Dec 14, 2022, Ann Arbor, MI, USA,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Redundancy elimination is the process of removing redundant computation that may be found across basic blocks in the code. For example, if 50 * 50 is computed twice, the compiler can change the IR to compute it just once and copy this result to the two desired destinations. This can provide incremental gains in overall run time with nearly zero code size expansion.

Traditional PRE has two main limitations [1]. The first is that front-end compilers can modify naming of variables in unique ways that may obscure potential redundant computations. The second is that the code shape, such as decisions about associativity and ordering of computational chains, can make certain redundancies non-obvious. Briggs and Cooper attempt to partially address these concerns in 1994 with global value numbering and global reassociation [1].

Global value numbering has since become a common strategy to improve the scope of PRE. The gvn pass in the LLVM compiler (gvn) is a library pass that combines global value numbering with redundant load elimination. The gvn pass is included in the O2 (and higher) levels of optimization in clang and gcc.

PRE is traditionally applied very conservatively. Traditional PRE does not use any profiling data, and will not optimize hot paths at the expense of cold paths - it will only optimize if there is no code size increase for any path. This can lead to missed optimization opportunities when considering control flows. Refer to 1 and 2 for an example of a case where PRE cannot optimize code, but ISPRE could. Since PRE has no knowledge of profiling data it cannot know whether hoisting up $a * a$ will result in code speed up or slow down so it must ignore it.

Speculative partial redundancy elimination (SPRE) is a technique that explicitly uses profiling information to find more aggressive optimization opportunities. It can be very effective, but very complex and difficult to implement, and as such many compilers stick to traditional PRE passes even if SPRE can provide large performance gains.

SPRE is much more computationally expensive than PRE as well. Its analysis time can be very long compared to other

```
for(int i = 0; i< n; i++){
     if(i%200 == 0){
         a = i;
     }else{
         sum = a*a;
     }
 }
```

Figure 1. Code that PRE cannot optimize

```
int t1 = a*a;
 for(int i = 0; i< n; i++){
     if(i%200 == 0){
         a = i;
         t1 = a*a;
     }else{
         sum = t1 ;
     }
 }
```

Figure 2. Optimized Code after ISPRE

code motion optimizations. Due to the complexity of SPRE's implementation, and the computational expense, Horspool et al. developed a novel approach called Isothermal Speculative Partial Redundancy Elimination (ISPRE) that delivers results similar to SPRE in practice, yet its analysis time is at least as good as current compiler techniques for code motion [2]. Our project is an implementation of ISPRE as described in their paper, as realized in a custom LLVM pass.

2 Methods

ISPRE uses dataflow analyses (similar to PRE) therefore bridging the gap between the faster speeds of PRE compared the SPRE and the better results of SPRE. In fact, as we will demonstrate in our results, in most cases ISPRE is faster than normal PRE. ISPRE uses profile information to divide a CFG G into two subgraphs - a hot region G_{hot} consisting of the nodes and the edges executed more frequently than a given threshold frequency θ, and a cold region G_{cold} consisting of the remaining nodes and edges. It is important to note that these hot and cold regions can be composed of multiple disconnected components of the CFG.

$$G_{hot} =< N_H, E_H >$$

$$G_{cold} =< N_C, E_C >$$

Where N_H, E, H are the hot nodes and edges and N_C, E, C are the cold nodes and edges respectively. We can then define the set of Ingress edges as

$$Ingress = \{(u, v) \mid u \in N_C \wedge v \in N_H\}$$

Essentially, the Ingress edges consist of all the edges in the program that flow from cold to hot regions of the CFG. ISPRE works by inserting expressions on edges in the Ingress set, which will then make some expressions in hot nodes become fully redundant. If those fully redundant expressions are then replaced by references to previously saved values of those same expressions, we have achieved code motion from G_{hot} to G_{cold}. This may increase the code size, but will optimize the hot path and lead to runtime performance gains.

This code motion is performed based on the results of two analyses: removability and necessity. Removability determines instances of computations in the hot region than can be deleted, and necessity determines edges in the Ingress set where these deleted computations must be inserted to maintain correctness of the code. Both of these analysis can be performed through dataflow analysis.

2.0.1 Removability Analysis. Removability analysis can be done by computed XUse, Gen, and Kill sets for each basic block in a function (or loop, or whatever scope you determine. Our implementation is a LLVM FunctionPass, and so analyses functions independently). These sets inform a forward dataflow analyses that create a set of removable expressions. The following equations define each of these terms.

$$\text{XUSES}(b) \stackrel{def}{=} \{\ e \mid \text{expression } e \text{ occurs in } b \text{ and is not preceded by any redefinitions of operands of } e\ \} \quad (1)$$

$$\text{GEN}(b) \stackrel{def}{=} \{\ e \mid \text{expression } e \text{ occurs in } b \text{ and is not followed by any redefinitions of operands of } e\ \}$$

$$\text{KILL}(b) \stackrel{def}{=} \{\ e \mid \text{block } b \text{ contains a statement which maybe redefine an operand of } e\ \}$$

$$\forall b \in N :$$

$$\text{AVOUT}(b) = (\text{AVIN}(b) - \text{KILL}(b)) \cup \text{GEN}(b)$$

$$\text{AVIN}(b) = \bigcap_{p \in preds(b)} \begin{cases} \text{Candidates} & \text{if } (p, b) \in \text{Ingress} \\ \text{AVOUT(p)} & \text{otherwise} \end{cases} \quad (2)$$

$$\forall b \in N_H :$$

$$\text{Removable}(b) = \text{AVIN}(b) \cap \text{XUSES}(b) \quad (3)$$

2.0.2 Necessity Analysis. The solutions for the *Removable* sets assume that computations of all candidate expressions are available on the *Ingress* edges. However, some candidate expressions are not necessary to insert on the ingress edges as they might be useless or redundant.

Necessity analysis is thus conducted to compute which expressions are required to be inserted on the ingress edges to maintain correctness. The set of expressions to be inserted on each ingress edge can be computed by backwards dataflow analyses, using the following equations.

$$\forall b \in N_H :$$

$$\text{NEEDIN}(b) = (\text{NEEDOUT}(b) - \text{GEN}(b)) \cup \text{Removable}(b)$$

$$\text{NEEDOUT}(b) = \bigcup_{s \in succs(b)} \text{NEEDIN}(s)$$

$$\forall (u, v) \in \text{Ingress}: \quad \text{Insert}(u, v) = \text{NEEDIN}(v) - \text{AVOUT}(u)$$

2.1 Multipass ISPRE

The methodology described above details a Single Pass ISPRE implementation. However, after the code motion resulting from the process above, we can apply subsequent passes of ISPRE with lower thresholds, Θ in order to find other regions of code or other instructions where code motion could be beneficial. By lowering Θ we select a larger G_{HOT}, then ISPRE will again move code from the hot region to the cold region, thus improving program performance. The algorithm for how we implemented Multipass ISPRE is as follows

Algorithm 1 Multipass ISPRE

$\Theta \leftarrow 0.9$ * *maximum block count*
passCount $\leftarrow 0$
repeat
  ISPRE **(CFG)**
  $\Theta \leftarrow \Theta/2$
  passCount $\leftarrow$ *passCount* + 1
until *passCount* = number of desired iterations

3 Results

3.1 Experimental Setup

Our goal was to test two seperate things: 1.) How ISPRE performs with respect to GVN (LLVM's built in Partial Redundancy Elimination Pass) and 2.) How Multipass ISPRE performs as we increase the number of iterations we run ISPRE. In order to test the first part, we created four custom C files as test cases, listed in the appendix (Section 5.1). We compiled each once with just the LLVM Global Value Numbering (GVN) and Dead Code Elimination passes, and once with just our custom ISPRE pass and dead code elimination pass (link to code listed in Section 5.1). To determine how ISPRE performs against GVN, we measured runtime, compile time, and IR bitcode size for both optimizations. The data displayed is the average of 10 runs for each test case. The results are listed in Table 1. To test the second part, we created 3 slightly more complex C programs in which long dependency chains could potentially highlight the benefits of a multipass optimization. On each of the test programs, we ran GVN, Singlepass ISPRE, and then Multipass ISPRE with 2,3, and 4 iterations. We tested multiple number of iterations in order to asses the cost to reward ratio for increasing the number of passes in Multipass ISPRE. Similarly to the first test, we measured runtime, compile time, and bitcode size for all 5 passes. The results are listed in Tables 2, 3, 4. Please note that in the results, the number in parentheses next to Multipass is the number of iterations ISPRE was run using the algorithm.

3.2 Discussion

In the Singlepass ISPRE vs GVN test, we can see that ISPRE clearly outperforms GVN across the tests as GVN cannot perform optimizations on those programs without the knowledge of branch weights. Across the 4 programs, ISPRE saw an average runtime decrease of **41.3%** and a compile time decrease of **39.5%** at the slight cost of a **0.497%** increase in IR bitcode size. The slight increase in size is caused by the incurred overhead of adding fix up code during code motion between hot and cold regions. However, regardless, the significant gains in runtime and compile time show the merit of incorporating profiling information into the partial redundancy elimination optimization. The significant gains in compile time could be a result of the fact that the GVN pass performs a few other optimizations on top of PRE, such as load elimination, which could slow down compile time.

The underlying code motion is demonstrated by Figures 3 and 4 in the Appendix. Code motion of an expensive multiply operation has been achieved from hot node $if.else$ into cold node $if.then$.

In the Multipass ISPRE test, the results stay consistent with the findings of the original paper and the singlepass test. Both singlepass and multipass ISPRE outperform the GVN pass, however notably, not as drastically as in the first test. Across the three tests from GVN $\rightarrow$ Singlepass ISPRE, we see a **21.1%** decrease in runtime and a **33.7%** decrease in compile time with a negligible decrease in size. From Singlepass ISPRE $\rightarrow$ 2 iterations of Multipass ISPRE, there is a **3.63%** decrease in runtime and a **14.2%** increase in compile time with a **1.19%** increase in size. From 2 iterations of Multipass $\rightarrow$ 3 iterations, there is a **1.24%** decrease in runtime and a **16.4%** increase in compile time with a **1.51%** increase in size. The same trend of diminishing return can be observed with 3$\rightarrow$ 4 iterations as well. Considering, the added costs in compile time and size, we believe some form of heuristics in determining which programs would benefit from Multipass ISPRE could be beneficial.

Test Case	GVN Runtime	GVN Compile Time	GVN IR Size	ISPRE Runtime	ISPRE Compile Time	ISPRE IR Size
ISPRE_1	0.354s	7.5ms	3504b	0.288s	4.9ms	3524b
ISPRE_2	0.376s	13.5ms	3504b	0.185s	6.3ms	3524b
ISPRE_3	0.061s	8.5ms	3932b	0.022s	6.8ms	3984b
ISPRE_4	0.376s	8.0 ms	3524b	0.189s	4.7ms	3540b

Table 1. Test Results Against Custom Benchmarks

Test Case	GVN	Singlepass	Multipass(2)	Multipass(3)	Mutlipass(4)
MULTI_1	1.257s	1.201s	1.172s	1.162s	1.159s
MULTI_2	1.395s	1.384s	1.342s	1.314s	1.308s
MULTI_3	1.485s	0.678s	0.662s	0.661s	0.627s

Table 2. Execution Times of GVN, Singlepass ISPRE, and Multipass ISPRE

Test Case	GVN	Singlepass	Multipass(2)	Multipass(3)	Mutlipass(4)
MULTI_1	7.5ms	5.6ms	6.4ms	7.5ms	9.7ms
MULTI_2	7.9ms	5.1ms	6.0ms	7.3ms	10.2ms
MULTI_3	9.1ms	6.2ms	6.9ms	8.3ms	9.6ms

Table 3. Compile Times of GVN, Singlepass ISPRE, and Multipass ISPRE

Test Case	GVN	Singlepass	Multipass(2)	Multipass(3)	Mutlipass(4)
MULTI_1	3244b	3248b	3288b	3336b	3376b
MULTI_2	3360b	3344b	3392b	3428b	3468b
MULTI_3	3876b	3864b	3900b	3940b	3980b

Table 4. IR Bitcode Size of GVN, Singlepass ISPRE, and Multipass ISPRE

4 Future Work

Though our pass is not as computationally intensive as Speculative PRE, there are still a few improvements that we could make to our ISPRE pass. As mentioned before, the Multipass optimization could benefit with heuristics in terms of when it should be applied, for how many iterations, and how Θ should change between iterations.

Building upon this idea, another improvement that we suggest is exploring the importance of the hot/cold threshold θ. Further research can be done on tuning this hyper parameter to determine the optimal value across a broad spectrum of test cases as the split of hot and cold regions can have a drastic impact on the performance of ISPRE.

Other future work includes analysis of register pressure and lifetime issues, and incorporating unsafe expressions into the framework.

5 Appendix

5.1 Source Code

Full source code for the LLVM pass, benchmarks, and tests can be found on GitHub (https://github.com/bd-g/isothermal-speculative-pre/).

Listing 1. ISPRE_1 Test Case Source Code

```
#include <stdio.h>

// Optimized by ISPRE, but not
// by standard PRE

int main() {
    long long int a = 0;
    long long int sum = 0;
    for (int i = 0;
         i < 100000000; i++) {
        if (i % 200000 == 0) {
            a = i;
        } else {
            sum += a * a;
        }
    }

    printf("Result: %llu\n",
           sum);

    return 0;
}
```

Listing 2. ISPRE_2 Test Case Source Code

```
#include <stdio.h>

// Optimized by ISPRE, but not
// by standard PRE

int main() {
    long long int a = 0;
    long long int sum = 0;
    for (long long int i = 0;
         i < 100000000; i++) {
        if (i < 90) {
            a = a + 2;
        } else {
            sum += a * 3;
        }
    }

    printf("Result: %llu\n",
           sum);

    return 0;
}
```

Listing 3. ISPRE_3 Test Case Source Code

```
#include <stdio.h>

// Optimized by ISPRE, but not
// by standard PRE

int main() {
    long long int a = 0,
                  sum = 0,
                  b = 0,
                  c = 0,
                  sum2 = 0,
                  sum3 = 0;
    for (long long int i = 0;
         i < 10000000; i++) {
        if (i < 90) {
            a = a + 1;
            b = b + 2;
            c = c + 3;
        } else {
            sum += a * 3;
            sum2 += b * 2;
            sum3 += c * 4;
        }
    }

    printf("Result1: %llu\n",
           sum);
    printf("Result2: %llu\n",
           sum2);
    printf("Result3: %llu\n",
           sum3);

    return 0;
}
```

Listing 4. ISPRE_4 Test Case Source Code

```
#include <stdio.h>

// Optimized by ISPRE, but not
// by standard PRE

int main() {
    long long int count =
        100000000;
```

```
long long int sum = 0;
int value = 10;
while (count > 0) {
    if (count >
        98000000) {
        value += 1;
    } else {
        sum += value * 2;
    }
    count--;
}

printf("Result: %llu\n",
       sum);
    return 0;
}
```

5.2 Control Flow Graphs

See Figures 3 and 4.

References

[1] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 159–170, New York, NY, USA, 1994. Association for Computing Machinery.

[2] R. Nigel Horspool, David J. Pereira, and Bernhard Scholz. Fast profile-based partial redundancy elimination. In *Proceedings of the 7th Joint Conference on Modular Programming Languages*, JMLC'06, page 362–376, Berlin, Heidelberg, 2006. Springer-Verlag.

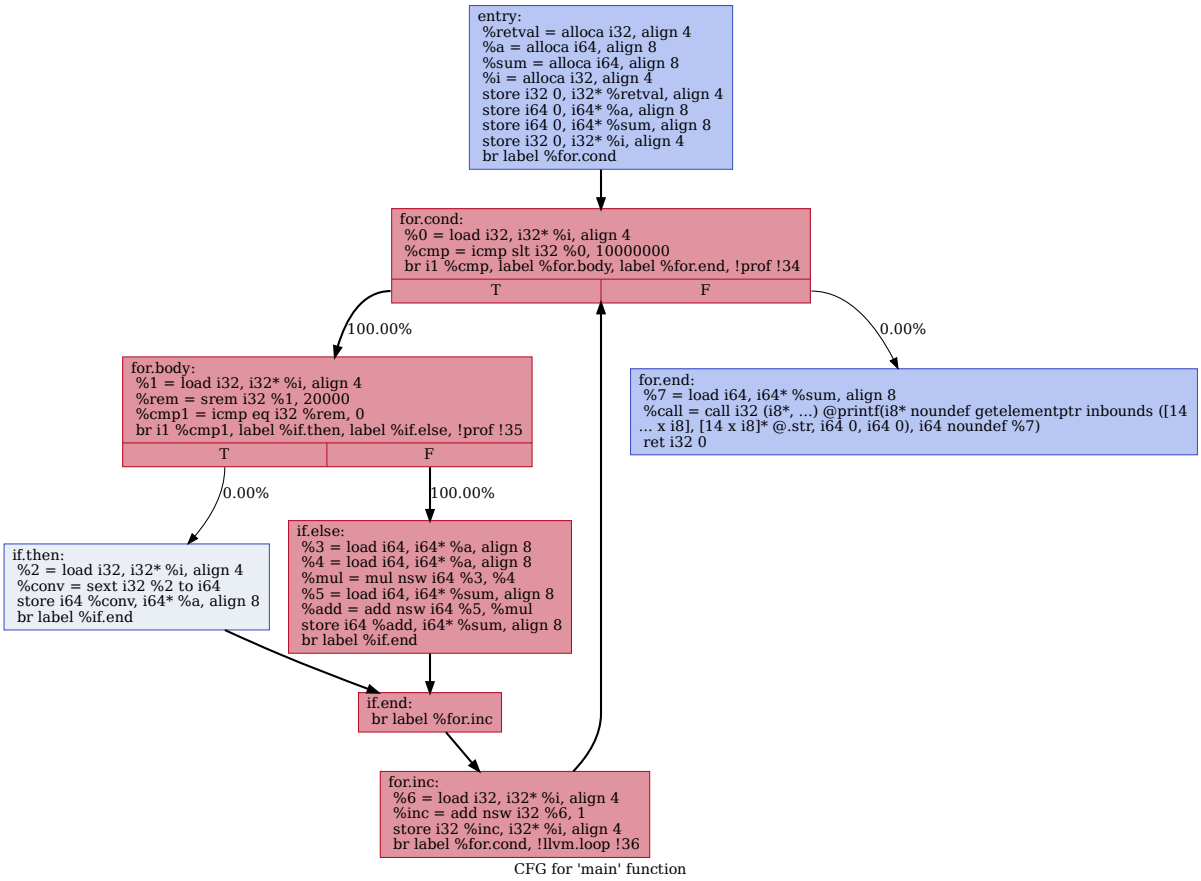

Figure 3. CFG of Test Case 1 Before Custom ISPRE Pass

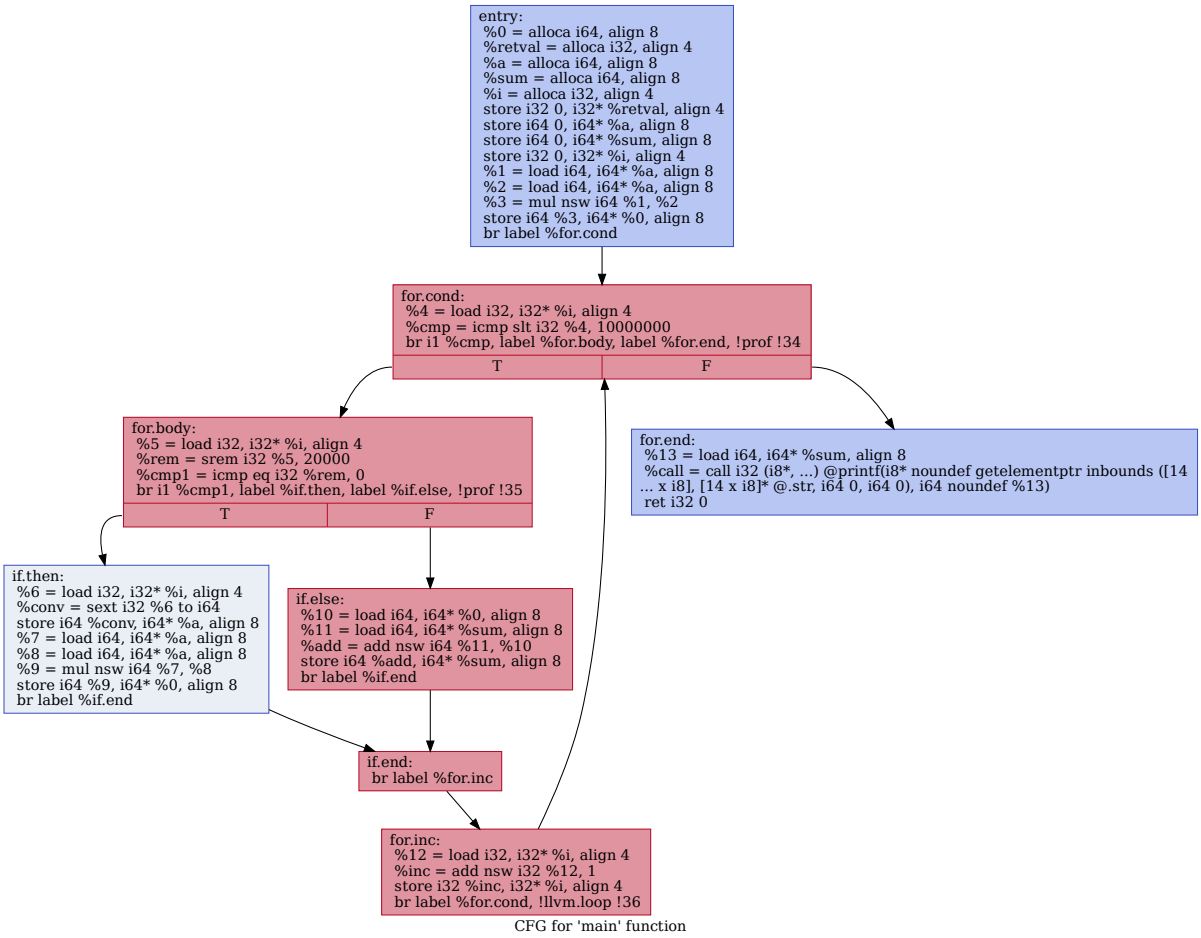

Figure 4. CFG of Test Case 1 After Custom ISPRE Pass