MC-PRE Implementation in LLVM

[EECS583 Final Project]

Qijun Jiang
University of Michigan
qijun@umich.edu

Ke Lin
University of Michigan
kelin@umich.edu

Yicong Zhang
University of Michigan
ianzyc@umich.edu

Zhizhong Zhang
University of Michigan
zhangzz@umich.edu

ABSTRACT

Partial redundancy frequently happens in compilers nowadays, and it is becoming important since it is difficult to optimize. We cannot simply replace all redundant computations. Partial redundancy elimination (PRE) is an optimization technique for removing partial redundancies and the key idea is to insert or delete computations in the control flow graph (CFG). The original formulations of PRE use lazy code motion (LCM) to insert expressions such that it became fully redundant. However it cannot guarantee the optimality on the number of insertions performed in CFG.

After studying several research papers, we found that speculation based PRE which uses edge profiling to remove partial redundancy according to execution frequencies is a good solution. And in the work of Qiong and Jingling [1], they present an efficient algorithm to apply Min Cut theorem to find all edges where performing insertions will be optimal. They provide detailed proof for correctness and optimality that the total number of computations for an expression is always minimized with respect to the edge profile given. Inspired by their work, we decided to implement MC-PRE algorithm in LLVM.

This paper presents the implementation of an efficient speculation-based partial redundancy elimination algorithm, called MC-PRE algorithm. Inspired by [1], we are implementing MC-PRE in LLVM where the original work was only for JAVA program and running on Intel Open Runtime Platform (ORP). We evaluated our algorithm on LLVM SingleSource benchmarks and the experiments show that our algorithm can achieve reasonable speedup compared to original program without optimization, and good optimization level compared to other frequently-used optimization passes.

Keywords

Partial Redundancy Elimination (PRE), Min Cut, LLVM

1. INTRODUCTION

In this course, we learned partial redundancy where some computations are repeated but only along some paths instead of all. It brings some difficulties to optimize partial redundancy since we cannot simply replace all redundant computations. After studying several research paper, we found partial redundancy elimination (PRE) technique is very popular and promissing. In the work of Qiong and Jingling [1], we learned that by applying Min Cut theorem on the reduced graph it will achieve better speedups. They orginally only experimented on JAVA benchmarks and it was running on Interl Open Runtime Platform (ORP). So for our final project, we decided to implement MC-PRE algorithm in LLVM and evaluate the performance.

1.1 Paper Organization

In Section 2, we review partial redundancy elimination (PRE), min cut algorithm and global value numbering (GVN). The system design and LLVM pass implementations are discussed in Section 3. Performance evaluations are discussed in Section 4, as well as some discussions in Section 5.

2. BACKGROUND & PREVIOUS WORK

In the work of Qiong and Jingling [1], they first introduced MC-PRE algorithm that analyzes edge insertion points based on an edge profile. The algorithm ensures the optimally that the total number of computations for an expression in the transformed code is always minimized with respect to the edge profile given. The key is about the removal of some non-essential edges/nodes from the flow graph, and the problem of finding an optimal code motion is reduced to finding a min cut in the reduced flow graph. In this project, we will use similar ideas as their original MC-PRE algorithms, and we will not mention proof for correctness and optimality here due to limited space. Please refer to their paper for the detailed proof of correctness and optimality.

2.1 PRE

Partial redundancy elimination (PRE) is an optimization technique for removing partial redundancies. Different from fully redundant, an expression is partial redundant if it is computed along some paths, but not all paths. Thus it brings some difficulties to eliminate partial redundancy. PRE is used to optimize partial redundancy by inserting or deleting computations in the control flow graph (CFG). The original formulations of PRE uses lazy code motion (LCM) to insert expressions at selected points such that the partial redundant expressions became fully redundant and could be eliminated. It uses availability and anticipability expression analysis to find the latest possible place for modification without changing the semantics and results of the original code. However, lazy code motion cannot guarantee the optimality, so we introduce speculation-based PRE which uses edge profiling to remove partial redundancies according to

execution frequencies. MC-PRE is one of the algorithms. It finds an optimal code motion by finding a minimum cut and guarantees the total number of expression evaluation is computationally optimal.

2.2 Min Cut

For a graph, a minimum cut is a partition that separates vertices into two disjoint subsets that are joined by at least one edge, where the partition will meet some minimal requirements, such as minimum number of edges or minimum weight on the edges. For a flow network, the min cut will separate the source and sink and minimize the total edge weights from the source side of the cut to the sink side of the cut. According to max-flow min-cut theorem, the weight of the min cut is equal to the maximum amount of flow from the source to sink.

In this project, we will have a weighted Single-Source-Single-Target reduced CFG where we will run Min Cut algorithm on. The edge weight represents the execution count and by finding the min cut we are able to perform minimum number of computations. For our project, we are using a greedy algorithm called Ford-Fulkerson algorithm to compute the maximum flow in a flow network, which is explained in Section 3.4.

2.3 GVN

Global value numbering (GVN) is a compiler optimization based on the static single assignment form (SSA) intermediate representation. GVN works by assigning a value number to variables and expressions. The same value number is assigned to those equivalent variables and expressions. There is a GVN pass in LLVM (release 3.3), and in our project we take use of GVN to find the equivalent expressions so that we can find all computation nodes for a given expression. Finding all computation nodes is essential in MC-PRE algorithm, and we will provide further disccusion about GVN in Section 5.

3. SYSTEM DESIGNS

In this section, we will discuss our system designs in detail. We implemented a LLVM pass, called **`mcpre`**, for evaluating MC-PRE algorithm. We used the similar idea presented in [1] as our guideline for the design. Like other LLVM passes, our system will take LLVM IR as input and output optimized IR where we eliminated partial redundancy and inserted minimum (optimal) computations.

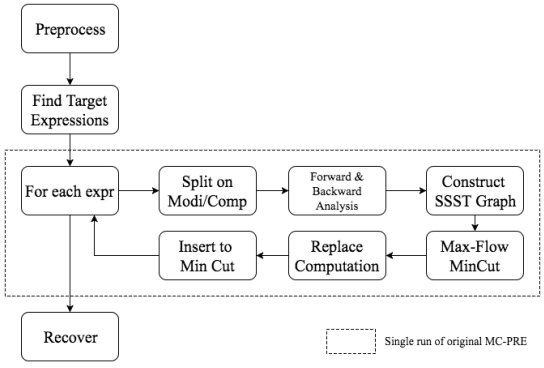

Figure 1: System Overview

3.1 Overview

The system flow is shown in Figure 1. In order to perform MC-PRE, we first need to find all possible target expressions that we can optimize. For each expression that we can optimize, we will split on modifications and computations, run forward partial availability analysis, run backward partial anticipability analysis, construct a Single-Source-Single-Target reduced graph based on analysis result, run Maxinum-Flow Min Cut algorithm to find the min cut, replace all computations and do insertions on all edges found by the min cut. This process is considered as one run of MC-PRE, which is marked in Figure 1.

One thing to notice is that performing MC-PRE requires to locate computation nodes and modification nodes in the beginning. However, LLVM is Static Single Assignment (SSA) based, and it loads variables to registers every time before they are used, making it difficult to locate computation nodes. Our solution is preprocessing the representation by replacing the register with its corresponding stack variable before elimination (Section 3.2). It requires addition recover opertions in order to make it executable. So we perform recovery after we run MC-PRE for all possible expressions in the end (Section 3.3). Also, we use a max-flow min-cut algorithm to find the min cut, which is introduced in Section 3.4.

```
int x = 0;
int y = 1;
a = x + y;
b = x + y;
```

(a) Simple Code Snippet

```
...
%0 = load i32* %x, align 4
%1 = load i32* %y, align 4
%add = add nsw i32 %0, %1
...
%2 = load i32* %x, align 4
%3 = load i32* %y, align 4
%add1 = add nsw i32 %2, %3
...
```

(b) Represention before Preprocess

Figure 2: Code Snippet and Representations

3.2 Preprocess

LLVM is using SSA-based representation, where an instruction is different from expression mentioned in [1]. A variable must be loaded to register every time before it is used, and one register can be only defined once. It means that one stack variable will be represented by multiple register variables. Even though several instructions are for computing same expression, they have different operands and cannot be considered as the same directly. For example, in Figure 2.b, the representations `%add` and `%add1` are for same expression `x+y`, but they have different operands. It brings difficulties to identify the computation nodes of an expression.

We perform preprocessing by replacing register variable with its corresponding stack variable as operands in an in-

struction. It finds the computing instructions (add, multiply, ...) and check the operands. If an operand represents a load instruction, the preprocess replaces the operand with the loaded stack variable. For the example in Figure 2, the operands (bold) in `%add` and `%add1` are replaced with `%x` and `%y` so that we have same operation type and operands for `%add` and `%add1`.

After preprocessing, we are able to locate the computation nodes under the help of Global Value Numbering (GVN). GVN provides us a mapping from expression to an unique number, then we consider all expressions with the same value number to be the computation nodes. Then for an preprocessed instruction, the modification node of a register operand is its definition expression, and that of a stack variable is the corresponding store instructions. Therefore, we can follow the ideas in [1] to find insertion edges for eliminating the partial redundancies now.

3.3 Recover

For the example shown above, although we can replace the operands in computing instructions with stack variables in preprocessing for performing MC-PRE, the modified instructions are not executable. The solution is to load the stack variable again before it is used, and replace it with the new register. As shown in Figure 3, `%x` and `%y` are loaded to `%ld1` and `%ld2`, then the instruction `%t` is modified. After that, the representation holds the properties of SSA and is executable again.

```
...
%t = add i32* %x, %y
store i32 %t, i32* %stackVar
...
```

(a) Represention before Recover

```
...
%ld1 = load i32* %x
%ld2 = load i32* %y
%t = add i32* %x, %y
store i32 %t, i32* %stackVar
...
```

(b) Represention after Recover

Figure 3: Recover for Previous Example

3.4 Ford-Fulkerson Min Cut Algorithm

The greed thought behind Ford-Fulkerson algorithm is as follows: as long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of the paths. Then we find another path, and so on. A path with available capacity is called an augmenting path. The algorithm in shown above, the path p in line 4 can be found, for example, using a breadth-first search or a depth-first search. And when the while loop breaks, s will not be able to reach t in the residual network. If S is the set of nodes reachable by s in the residual network, then the total capacity in the original network of edges from S to the remainder of V a) is equal to the total flow we found from s to t; and b) serves as an upper bound for all such flows. This proves that the flow we found is maximal.

Algorithm 1 Ford-Fulkerson

1: Input: network $G = (s, t, c, V, E)$ where c is capacity
2: Output: flow f from s to t of maxinum value
3: Initialization: $f(u, v) = 0$ for all adges (u, v)
4: **while** there is a path p from s to t in G_f, such that $c_f(u, v) > 0$ for all edges $\in p$ **do**
5: find $c_f(p) = \min\{c_f(u, v) : (u, v) \in p\}$
6: **for** each edge $(u, v) \in p$ **do**
7: $f(u, v) = f(u, v) + c_f(p)$ (send flow along path)
8: $f(v, u) = f(v, u) - c_f(p)$ (might be "returned")
9: **end for**
10: **end while**

4. EVALUATION

We conducted all experiments on CAEN machine which is running on 64-bit Red Hat Enterprise Linux Workstation 7.2, and we are using LLVM 3.3 release. We analyze the correctness and performance of MC-PRE for a number of benchmarks from LLVM SingleSource test suites. We compared the output of the binary optimized with MC-PRE pass and the reference output, all benchmarks passed the correctness test. For the performance test, we compared the execution time of binaries optimized by MC-PRE (referred as `mcpre`) and binaries optimized by several frequently-used optimization passes (referred as `baseline`) [2]. The execution time percentage is calculated based on the execution time of binaries without any optimization.

Table 1: Optimization Passes

pass name	-opt switches
`mcpre`	-mcpre -mem2reg
`baseline`	-mem2reg -loop-rotate -reassociate
	-mem2reg -simplifycfg

Figure 4 shows that our pass `mcpre` outperforms `baseline` for multiple programs in LLVM SingleSource benchmarks. The execution time is about 40% - 60% of original program without optimization. Our algorithm reduced the number of computations by inserting evaluation of the expressions on edges found by Min-Cut and replacing computation nodes. The results meet our expectations and we find that the speedup is around 5% compared to `baseline`.

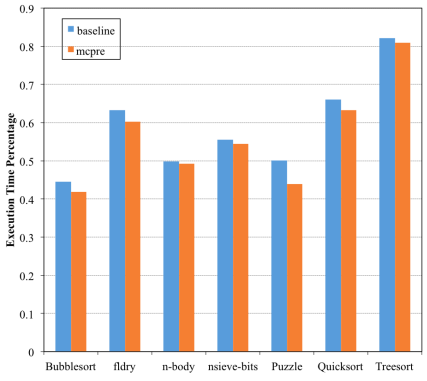

Figure 4: Execution Time Percentage (Benchmark Set 1)

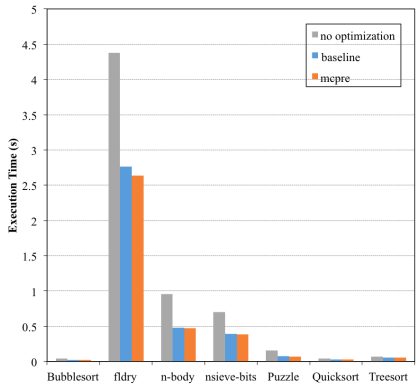

Figure 5: Execution Time (Benchmark Set 1)

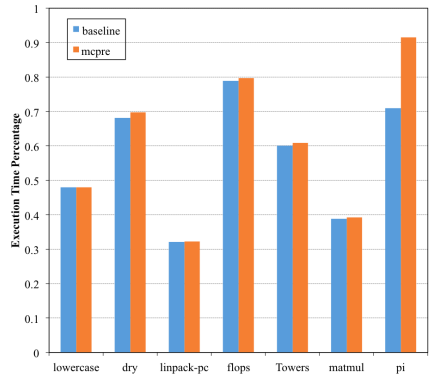

Figure 6: Execution Time Percentage (Benchmark Set 2)

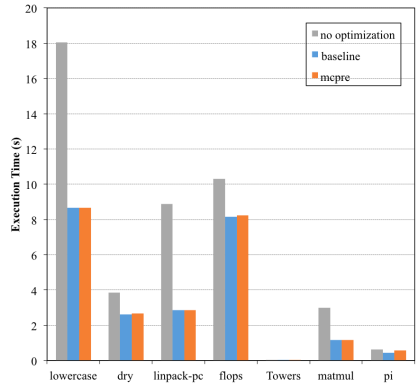

Figure 7: Execution Time (Benchmark Set 2)

Figure 6 shows the benchmarks where the `mcpre` is slightly slower than the `baseline`. One possible reason is that there is little partial redundancy in these benchmarks. Even the expression cannot be optimized by our algorithm, we replace the computation by loading the result from a stack variable where we store the value of the expression in the preprocess. The extra store/load operations result in the increment of execution time. It is also worth mentioning that, we only process the basic instructions in our design. There is a great possibility that the performance of our algorithm could be improved if we could also support complicated instructions. Figure 7 shows that although the `mcpre` is slightly slower than the `baseline`, our algorithm is lightweight and efficient as the execution time is at the same level as the baseline.

5. DISCUSSION & FUTURE WORK

As GVN is essential in finding the computation nodes of an expression, it largely domains the efficiency of MC-PRE algorithm. In the process of reducing the graph, we need to find all the computation nodes and connect the virtual source with them. We currently borrowed the value table from the LLVM/GVN.cpp (release 3.3), and our experiments show that it can only recognize well on those simple expressions like $r_1 = a + b - c$. For more complex expressions, such as $r_2 = ((b * d) - (a/c))/(k + j)$, it does not perform well. During the development of our project, we have several important findings that might help for future explorations:

1. Up to now, due to the limited time for development, we only support basic instructions (add, mul, sub, div...) for our algorithm. According to the experiment result, we can see that there is some potential for further speedup if more instructions are included. It also might decrease the performance since we will have more time for compilation and that might not cover what we improved on execution. We highly recommend to add more instructions to experiment on.

2. For the benchmark evaluation, we tried SPEC2006 first but currently our algorithm does not handle well on getting edge profiles from multi-source programs. It is worthwhile to find a way for handling multi-source programs and experiment on SPEC2006 to evaluate the performance.

3. As mentioned above, for this project we are using GVN code from LLVM 3.3 release, however, the performance is not very good. LLVM GVN could not handle complex expressions so we could not find equivalent expressions when generating the reduced graph and performing MC-PRE algorithm. We recommend to better understand GVN or rewrite one if necessery.

6. CONCLUSIONS

In this project, we understand and implement an efficient speculation-based partial redundancy elimination algorithm, called MC-PRE algorithm. To apply the algorithm in LLVM, we explore and implement a LLVM pass **`mcpre`**. To evaluate the performance of our algorithm, we experiment on a number of benchmarks from LLVM SingleSource test suites and compared the results with other frequently-used optimization passes. The experiment result shows that our algorithm can achieve reasonable speedup compared to original program without optimization, and good optimization level compared to other frequently-used optimization passes.

7. ACKNOWLEDGMENTS

We wish to thank Professor Lingjia Tang at University of Michigan for her insightful comments and suggestions, GSI Animesh Jain for his assistance in project design process, and multiple authors/contributors of the original MC-PRE algorithm.

8. REFERENCES

[1] Q. Cai and J. Xue. Optimal and efficient speculation-based partial redundancy elimination. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 91–102, Washington, DC, USA, 2003. IEEE Computer Society.

[2] S. Dasgupta and T. Gangwani. Partial redundancy elimination using lazy code motion. 2014.