Optimizing Utilization of Memory Hierarchy Based on Code Motion

コード移動に基づくメモリ階層利用最適化

by
Yasunobu Sumikawa

Submitted to the Graduate School of Science and Technology
in partial fulfillment of the requirements for
the degree of Doctor of Science

Tokyo University of Science
2015

Abstract

Access speed to main memory is much slower than processor speed. In order to reduce the number of accesses to memory, most modern processors have some registers and cache memories that are much faster than main memory. To use such registers and cache memories effectively, many researchers have proposed several code optimization techniques that promote their utilization. However, because the gap between processor and main memory access speed is growing, and such gap is likely to increase in the future, the importance of solving this issue is also increasing. That is, we have to develop more sophisticated code optimization techniques for the effective use of registers and cache memories.

In this thesis, we propose four new techniques for decreasing the number of references from the processor to the cache memory, and from the cache memory to the main memory. One of these techniques is an extension of the traditional *partial redundancy elimination* (PRE) that removes redundant expressions, and the other techniques are types of optimization techniques based on PRE. A brief description follows. First, we propose *effective demand-driven PRE* (EDDPRE) that improves analysis efficiency without sacrificing the effectiveness of the traditional PRE. Second, we propose *PRE-based scalar replacement* (PRESR) that removes array references that are redundant over loop iterations. As EDDPRE and PRESR remove redundant array references by replacing them with temporary variables, these techniques increase the number of used registers, rather than cache/main memory, because the data referred by arrays are stored on the main memory, whereas the data from temporary variables can be stored in registers. Finally, we propose *global load instruction aggregation* (GLIA) that improves the spatial locality of memory by making accesses to the same array continuous, so that suppressing cache misses. Furthermore, we extend GLIA to manage multidimensional arrays that can be regarded as arrays of lower dimensional arrays, and we call the extended GLIA *multidimensional GLIA* (MDGLIA). MDGLIA continuously aggregates the array references with the most similar indexes in the highest dimensions, even if they are not identical. GLIA and MDGLIA decrease the number of cache misses, thus preventing access to the main memory.

We implement our techniques in a COINS compiler and evaluate them

with the standard performance evaluation corporation (SPEC) benchmark programs. The experiment results show that EDDPRE improves analysis efficiency by about 50% on average. PRESR improves execution efficiency by about 2% on average. GLIA and MDGLIA decrease cache misses in many programs.

The effective utilization of registers and cache memories achieved by these techniques result in the efficient execution of object code.

Keywords: memory hierarchy, compiler, code optimization, data-flow analysis, code motion, partial redundancy elimination, scalar replacement, cache optimization

Acknowledgements

To express my gratitude, I write acknowledgements in Japanese.

博士論文を執筆するにあたり，修士課程の 2 年間と博士後期課程の 3 年間の計 5 年間，多くの方々にお世話になりました．この 5 年間，思うように成果を出せずに苦しんだ時期もありましたが，とても多くの素晴らしい出会いに恵まれ，研究を行うための心構えや考え方について助言や励ましを多くの方にしていただきました．無事に博士論文を完成させることができたのは，皆様のお力添えのお蔭でございます．ここで深く感謝申し上げます．

まず修士課程からの 5 年間指導してくださった滝本宗宏先生に深く感謝致します．先生には，まだ私が数学科に所属していたときから自由に研究室やゼミに出入りして計算機科学の勉強と研究を行う環境を整えてくださり，研究での議論では毎回本質的なコメントをいただきました．また，研究を行うためには 1 つのことを追求するだけでなく幅広い知識を学ぶことが大事だからと，他大学で開催されている勉強会や研究会に私が参加できるようにご支援していただきました．その中で，非常勤講師や共同研究を行う機会にも恵まれ，専門外のことも学ぶことができ，自分に出来ることが着実に増えていることを実感することができました．院生として過ごした日々が本当に楽しくて刺激的であったのも先生の暖かいご支援のおかげです．私も先生のように寛大で的確な助言をできる研究者になれるように，これからも励んでいきたいと思います．本当にありがとうございました．

次に，お忙しい中，博士論文の審査を引き受けていただきました，武田正之先生，太原育夫先生，明石重男先生，大和田勇人先生，西山裕之先生に御礼申し上げます．先生方には，博士論文を作成している時は励ましてくださり，審査が始まると，博士論文への有益なコメントと共に「勉強になった」というとても嬉しいお言葉をいただきました．これから本研究を更に発展させられるように研究を進めていきます．ありがとうございました．

また，計算機科学に関係する名著を読む輪講に，私が学部生のときから参加することを歓迎してくださった筑波大学の久野靖先生と，久野ゼミのメンバー皆様に深く感謝致します．先生のゼミに参加することで辞書のような分厚い本を読み，型理論やプログラミング言語の意味論といったコンパイラの研究を行うための基礎知識となる数々の重要な理論や概念を学ぶことができました．これも，皆様の非常にわかりやすい解説によって，難しい内容でも挫けることなく本を読み進めることができたおかげです．また，私が担当していた非常勤講師の授業で，どうすればよりわかりやすく，学生がプログラミングを楽しいと思える授業をできるのかについて先生に相談させていただ

きました．これからは，学生がプログラミングを学ぶ手助けをする機会が増えますが，試行錯誤を続け，プログラミングや計算機科学の楽しさを伝えられるように頑張ります．本当にありがとうございました．

私が数学科の学生のときに卒業研究のご指導をしていただき，修士課程から情報科学専攻に進学するためのご支援をして下さった東京理科大学理工学部数学科の牛島健夫先生に感謝致します．最後のゼミ発表では，学部 3 年の後期からの約 1 年半の間にゼミで学んだ内容を踏まえ，各自の興味のある事を数学と関係させた内容を発表する事になっていましたが，私の修士課程での研究内容を考慮していただき，コード最適化について発表する事を薦めてもらいました．この発表が分野外の人に研究紹介を行う初めての機会でしたが，先生に「とてもわかりやすかった．興味深いですね.」と言ってもらえ，とても嬉しかったです．また，先生のゼミで学んだ微分方程式や不動点は，現在進めている研究と密に関係しており，これらの内容を扱う論文や本を読むときはいつも先生の指導を思い出しています．親切なご指導と暖かいご支援をしてくだり，ありがとうございました．

私を東京理科大学へ導いてくれた中学・高校時代の友人や先生方，歴代の滝本研究室のメンバーを含め，これまでに多くの素晴らしい友達に出会うことができました．大学院での生活は孤独であるとよく聞きますが，私がそのように感じることはありませんでした．これも皆様と一緒に遊んだり飲みに行く事でリフレッシュができたおかげです．どうもありがとうございました．これからもどうぞよろしくお願い致します．

最後に，頑固で負けず嫌いであり，特に家族からの助言には耳を貸そうともしない私を，時には厳しく，時には優しく支え続けてくださった両親と，小さい頃から切磋琢磨していた兄妹に感謝致します．大学を卒業したら高校で数学の教師になるとずっと言っていたのに，学部 3 年生の秋に大学院への進学に進路変更を決めたとき，無理を言って承諾してもらいましたが，大学院での生活は本当に楽しくて，この道を進めることができて良かったです．博士後期課程に進学してからはあまり顔を見せずにいましたが，これからは親孝行をしていきたいので，いつまでも元気に過ごしてください．

2015 年 1 月
澄川靖信

Contents

List of Figures **vii**

List of Tables **x**

1 Introduction **1**
1.1 Motivation 1
1.2 Memory Hierarchy 3
1.3 Removing Redundant Expressions 4
1.4 Contribution 7
1.5 Thesis Organization 13

2 Background **14**
2.1 Preliminaries 14
2.1.1 Program Representation 14
2.1.2 Control Flow Graph 14
2.1.3 Data-flow Analysis 16
2.1.4 Static Single Assignment Form 19
2.2 Fundamental Code Optimization Techniques 20
2.2.1 Global Value Numbering 21
2.2.2 Common Sub-expression Elimination 23
2.2.3 Loop Invariant Code Motion 23
2.2.4 Partial Redundancy Elimination 24

3 Effective Demand-driven PRE **31**
3.1 Motivation 31
3.2 PRE Based on Query Propagation 32
3.3 Effective Demand-driven PRE 37
3.3.1 Algorithm Overview 37
3.3.2 Optimistic Global Value Numbering 38
3.3.3 Query Propagation 44
3.4 Experimental Results 48
3.5 Related Work 51
3.6 Summary 52

4 Demand-driven Scalar Replacement **53**
4.1 Motivation . . . 53
4.2 Related Work . . . 56
4.2.1 Scalar Replacement . . . 56
4.2.2 Register Promotion . . . 57
4.3 Array Reference Representation . . . 58
4.4 Demand-driven Scalar Replacement . . . 59
4.4.1 Value Numbering for Array References . . . 60
4.4.2 Redundancy Removal over Iteration . . . 60
4.5 Experimental Results . . . 64
4.6 Summary . . . 66

5 Global Load Instruction Aggregation **67**
5.1 Motivation . . . 67
5.2 Related Work . . . 69
5.2.1 Cache Optimization . . . 69
5.2.2 Removing Redundant Expressions . . . 70
5.3 Background . . . 71
5.3.1 Program Representation . . . 71
5.3.2 Lazy Code Motion . . . 71
5.4 Array Reference Aggregation . . . 74
5.4.1 Local Properties . . . 74
5.4.2 Modified Global Properties . . . 75
5.4.3 Application to the Entire Program . . . 80
5.5 Experimental Results . . . 81
5.6 Summary . . . 87

6 Conclusion and Future Direction **88**
6.1 Summary of Contributions . . . 88
6.2 Future Direction . . . 90

List of Figures

1.1 Memory hierarchy. 3
1.2 Jacobi's stencil solver. 5
1.3 Example of removing redundant expressions over iteration. (a) Original code. (b) A result of scalar replacement. 5
1.4 Removing redundant expressions. (a) Original code. (b) Result of removing redundancy. 6
1.5 Using PRE to remove partially redundant expression. (a) Original code. (b) Result of PRE. 7
1.6 Propagating a query for `a[i]` by EDDPRE. (a) Original program. (b) Query propagation. 8
1.7 (a) Result of query propagation shown in Fig. 1.6. (b) Translating the original program. 9
1.8 Query propagation of PRESR. (a) Original program. (b) Query propagation. (c) Result of query propagation. (d) Result of translating the original program. 10
1.9 Motivation example of GLIA. This figure represents that execution `b[i]` expels the reference data of `a[i]` and `a[i+1]` from the cache memory because the execution results in cache miss. 12
1.10 Moving the array reference `a[i+1]` by GLIA. 12

2.1 (a) Original code. (b) An example CFG of the code. Each node corresponds to a basic block. 15
2.2 Removing critical edge. (a) Original code. (b) After removing the critical edge and applying PRE. 16
2.3 An example code for constant propagation. 17
2.4 An example of SSA form. (a) Normal form code. (b) SSA form code. 20
2.5 Assigning value numbers. (a) Original code. (b) After value numbering. The numbers with "[]" represent the value numbers of the corresponding expressions. 21
2.6 Effectiveness of GVN. (a) Original code represented in a CFG. (b) Result of applying GVN on a dominance tree. The labels of the tree correspond to the CFG. 22

2.7 Effectiveness of CSE. (a) Original code. (b) Result of applying CSE. 23
2.8 Effectiveness of LICM. (a) Original code. (b) Result of applying LICM. 24
2.9 Effectiveness of PRE. (a) Original code. (b) Result of applying PRE. 25
2.10 Motivation example of LCM. (a) Original code. (b) down-safety of `a[i]` 28
2.11 Insertion candidates. (a) *Earliest*. (b) *Latest*. 29
2.12 Translated program. (a) *Isolate*. (b) Result of applying LCM. 30

3.1 Effect of PREQP's removing redundant expression and applying copy propagation. (a) Original program. (b) After removing $\mathtt{x}_1\mathtt{+1}$ at Node 3. 33
3.2 Effect of PREQP's query propagation. (a) Example of propagating a query. (b) Result program of applying PREQP . . 34
3.3 Proof of equality between traversals of topological sorted dominance tree in depth-first and left-first search and a topologically sorted CFG. We assume $2 < \mathtt{k} < \mathtt{r}$. 39
3.4 Effect of optimistic value numbering. (a) Original program. (b) A result of GVN that is carried out on the dominator tree. 40
3.5 Effect of EDDPRE's query propagation. (a) Original program. (b) A result program. 45

4.1 Effect of PRESR. (a) Original program. (b) After applying PRESR. 54
4.2 An example program after attaching versions for arrays. When PRESR propagates a query for $\mathtt{a}_3\mathtt{[i}_1\mathtt{]}$, its attached version is changed by using *arrayVersion*, as displayed in Table 4.1. . 59
4.3 Resulting program after applying PRESR to Fig. 4.2. 59
4.4 Effect of PRESR. (a) Original program. (b) After applying PRESR. 62
4.5 Ratio of analyzing costs, compared with exhaustive style of PRE. 66

5.1 An example of a cache miss that is a target of MDGLIA. . . 68
5.2 Effectiveness of MDGLIA. (a) Original code. (b) Result of applying MDGLIA. 69
5.3 Code motions of LCM. (a) Hoisting expressions. (b) Delaying. 72
5.4 Result of applying LCM to Fig. 5.3(a). 72
5.5 Speculative code motion. (a) Original code. (b) Moving array reference, not satisfying down-safety. 76

5.6 Effectiveness of extending *UpSafe*. (a) Result of computing data-flow equations to move `a[k][l]` at Node 5. (b) Result of moving the array reference. 77
5.7 Computing closeness of each addresses of `a[i][l]` at Node 4 and *Delayed*. 79
5.8 Preserving unnecessary code motion. (a) Original code. (b) Result of applying exhausive analysis version of MDGLIA to `a[i+1]`. (c) Result of applying demand-driven style MDGLIA to `a[i+1]`. 81
5.9 Decrease rate of cache misses. 82
5.10 Difference of an insertion point for `a[i][j+1]`. (a) Original code. (b) Result of applying LCM-MEM. (c) Result of applying GLIA. (d) Result of applying MDGLIA. 84
5.11 The decrease ratio of L2 cache miss for GLIA, MDGLIA, USGLIA, and USMDGLIA to the cache miss for LCM-MEM. 85

List of Tables

3.1 Execution time of objective code 49
3.2 Analysis time 49
3.3 The time of query propagation 50
3.4 The number of nodes which query propagated 50

4.1 *array Version*s from Fig. 4.2 59
4.2 Results of execution time 65
4.3 Results of the dynamic number of load statement 65

5.1 System parameters of cache memories 82
5.2 The number of DCache_Repl 83
5.3 The number of L2_Lines_Out 83
5.4 The number of register spills 84
5.5 The number of L2 cache miss of USGLIA and USMDGLIA, and the cache miss ratio of them to the cache miss for GLIA and MDGLIA 86
5.6 The number of aggregated array references under *keepOrder* and *keepDimension* in speculative/not speculative code motion 86

Chapter 1

Introduction

1.1 Motivation

When we execute a program described in a high-level programming language, the program has to be processed by the own processor of the language, such as an interpreter or a compiler. If programmers' main concern is execution efficiency, for example, the program is expected for product use, the compiler should be used for applying code optimizations to the program partially or totally to transform it into efficient one. In particular, because optimizing compilers apply several code optimizations to a program, they generate extremely efficient code for the machine where the program is executed.

Initial compilers could only apply limited code optimizations to a program; therefore, in many cases, the code directly described in the machine code by expert programmers was more efficient than generated by compilers. However, as the size of each program increases, it has been difficult for programmers to manually create machine code considering the entire program, whereas improving a lot of code optimization techniques based on global information has made compilers requisite items for programmers.

Since the 1960s, many researchers have proposed several optimization techniques for optimizing compilers. These researchers' efforts primarily focused on solving the issue of the main memory's access speed functioning much slower than the processor's speed. In order to reduce the number of accesses to the main memory, modern processors have a set of registers that work as fast as processors. Modern processors also have cache memories that are much faster than the main memory though they are slower than the registers.

Most compilers apply two optimization techniques for the effective use of registers, register promotion and register allocation. Register promotion replaces variables declared by users with virtual registers. On the other hand, register allocation tries to allocate the virtual registers to as many physi-

cal registers as possible. Notice here that virtual registers have the same properties as physical registers except that the number of virtual registers is infinite, corresponding to temporary variables generated inside a compiler. In general, unlike virtual registers, user-declared variables may be modified through side effects; therefore, they should be allocated in the main memory. A simple way to replace variables with virtual registers is to pick up variables with no side effects, and then to replace them. This way enables totally replacing the variables with the virtual register, but the number of them is restricted. In order to handle variables with side effects, there is the way to replace redundant load instructions with temporary variables holding a result of preceding load instructions that access the same memory locations and load the same value.

For the effective use of cache memory, it is important to continuously access memory locations whose copies exist in cache memory simultaneously. Therefore, to continuously access contiguous memory locations, it is popular to transform index of array references inside a loop.

Although many researchers have tried to solve the issue, the gap between processor and main memory access speed is, unfortunately, growing. That is, the percentage of the time required to access the main memory in the execution time of program become increasing year by year; therefore, solving the issue is becoming more important.

This thesis presents new techniques that utilize code motion to more effectively handle the use of the registers and cache memory. Code motion is a technique for program transformation that moves expressions or statements to suitable points in a program. Techniques based on code motion can be categorized into heuristic approaches and data-flow analysis approaches. Heuristic approaches are used in cases where code motion simply leads to the improvement of the performance of a program (for example, parallelizing instructions), whereas data-flow analysis approaches are used in cases in which code motion is utilized depending on the global conditions of a program such as removing redundant expressions. To remove partially redundant expressions that are redundant on some execution paths, many code motion techniques based on the data-flow analysis have been proposed since 1980s. Our approach also uses a data-flow analysis-based code motion in order to remove redundant load instructions or aggregate array references to the same array or the same higher-dimensions of the same array. These techniques contribute to promoting user declared variables to virtual registers and decreasing cache misses.

In the remainder of this chapter, we summarize the memory hierarchy of which we improve utilization and the approach adopted for the improvement.

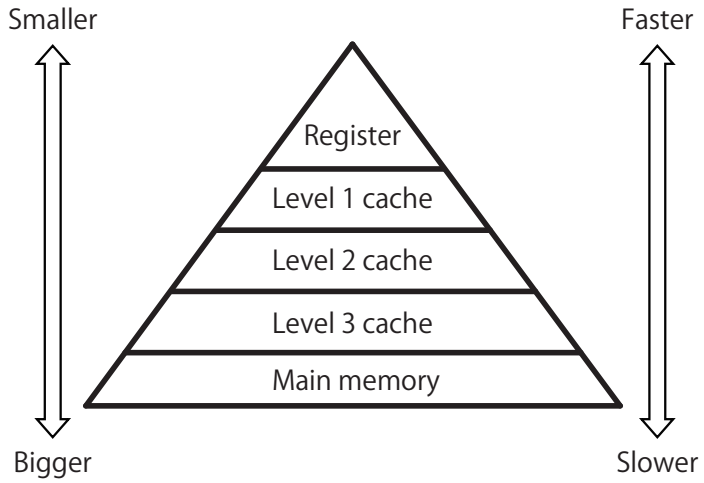


Figure 1.1: Memory hierarchy.

1.2 Memory Hierarchy

A computer needs memory to retain a program and the data for the execution of the program. In general, a computer uses three kinds of memory: register, cache memory, and main memory. These three kinds of memories are used hierarchically to make the memory system fast and scalable. This structure of several memories is called the *memory hierarchy*. As shown in Fig. 1.1, there is a trade-off between speed and size in the memory hierarchy. The feature of each type of memory is summarized as follows:

Register is the fastest type of memory, but the bigger it is, the more expensive its production costs. Therefore, most CPUs have just a small set of register.

Cache memory is used to store copies of data that are frequently accessed from the main memory. It works as fast as the register if accessed data is included in it; otherwise, it is necessary to access main memory to copy the data. In terms of efficiency and size, it is medium. The middle layer for cache memory can consist of several caches as sublayers. The lower level caches are relatively slower, although they are allowed to be larger in size.

Main memory is the slowest kind of memory in the memory hierarchy although it is the biggest. Once the memory is accesses, the CPU stalls; therefore, memory access remarkably decreases the execution efficiency of a program.

The memory hierarchy enhances the efficient utilization of different types of memory based on the *principle of locality* [44]. Locality is categorized into two basic varieties: *temporal locality* and *spatial locality*.

Temporal locality. If a memory location is accessed, it tends to be accessed again in the near future.

Spatial locality. If a memory location is accessed, other data around it tends to be used in several parts of the program.

Whenever the processor needs data at a location x in the main memory, the cache memory is checked first to determine whether it stores a copy of the data. If the data is found in the cache memory, it can be obtained without any main memory access; otherwise, the processor fetches the data around x from the main memory and places them in the cache memory so that it can be available for subsequent accesses. The former case is called a *cache hit*; and in contrast, the latter case is called *cache miss*. If a cache miss occurs, the access to x not only causes a significant delay to fetch the data around x in the main memory, but also to remove the old data from the cache memory. Thus, the key points for improving execution efficiency are to 1) increase the use of registers, and 2) decrease the number of cache misses.

1.3 Removing Redundant Expressions

Redundant expressions are executed in various kinds of practical programs, such as computations of the offsets of array references, calculating Fibonacci numbers, multiplying matrices, and using the stencil solver. Further, about half of the load instructions in SPEC's Benchmark, which is one of the most popular benchmarks to confirm improvement of techniques in the code motion research field, can be removed as they are redundant [7].

Example.

Figure 1.2 shows the pseudo code for Jacobi's stencil solver. Consider the two array references `a[i][j-1]` and `a[i][j]`. For the first iteration of the inner-loop (`j=1`), these array references execute `a[i][0]` and `a[i][1]`, respectively. For the second iteration (`j=2`), they execute `a[i][1]` and `a[i][2]`, respectively. That is, `a[i][j-1]` references the data that is referred to by `a[i][j]` in the previous iteration; therefore, `a[i][j-1]` is redundant. In addition, `a[i][j]` and `a[i-1][j]` are redundant because these referenced data are already referred in the previous iteration of the outer loop by `a[i+1][j]` and `a[i][j]`, respectively.

End of Example.

```
for(i=1;i<N;i++){
 for(j=1;j<N;j++){
  T[i][j] = 0.2*(a[i][j]+a[i-1][j]+
            a[i+1][j]+a[i][j-1]+a[i][j+1]);
 }
}
```

Figure 1.2: Jacobi's stencil solver.

```
while(i<N){
  sum+=a[i]+a[i-1];
  i++;
}
```

(a)

```
t'=a[0];
while(i<N){
  t=a[i];
  sum+=t+t';
  i++;
  t'=t;
}
```

(b)

Figure 1.3: Example of removing redundant expressions over iteration. (a) Original code. (b) A result of scalar replacement.

Removing array references that are redundant over iterations is called *scalar replacement*. We show how traditional scalar replacement removes the redundant array reference with a simple program shown in Fig. 1.3.

Example.

The array reference, `a[i-1]` in Fig. 1.3 (a), is redundant because the reference data is referred by `a[i]` in the previous iteration. To remove the redundant array reference, traditional scalar replacement introduces a new temporary variable `t` for holding the value of `a[i]`, and inserts `t'=a[0]` before the loop and `t'=t` at the exit of the loop body. Finally, `a[i-1]` can be replaced with `t'` because `t'` has an initial value of `a[i-1]`, otherwise, the temporary variable holds the value of `t` for the next iteration. The result of this translation is shown in Fig. 1.3 (b).

End of Example.

```
x=a[i];   x=a[i];
y=a[i];   y=x;
```

(a) (b)

Figure 1.4: Removing redundant expressions. (a) Original code. (b) Result of removing redundancy.

Removing redundant expressions is an important technique for improving execution efficiency because it can shorten the critical path or reduce the necessity of using hardware. Furthermore, removing redundant memory references is a critical technique because it reduces the number of references for cache and main memory. This thesis presents a demand-driven scalar replacement based on *partial redundancy elimination* (PRE) that is one technique for removing redundancy. In addition, we extend PRE to decrease the number of cache misses. In the remainder of this section, we show how redundant expressions are removed by PRE.

Example.

Consider the two statements that contain `a[i]` in Fig. 1.4 (a). The values of the operand variable `i` and `a[i]` are the same because there is no variable definition or store statement between the statements; therefore, the right-hand side of the statement `y=a[i]` is redundant. This expression can be removed by substituting the value referred to by the preceding expression; that is, the redundant expression can be removed by substituting `x` in the left-hand side of the preceding statement as shown in Fig. 1.4 (b).

End of Example.

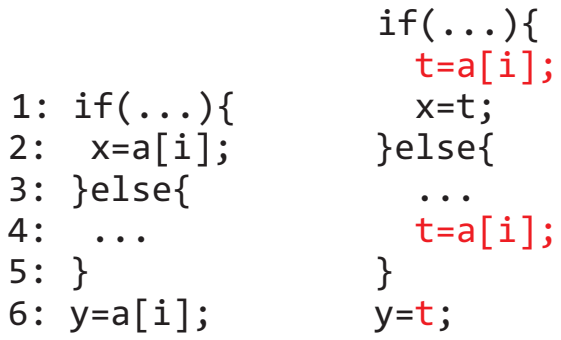
If a program contains some branch instructions, the preceding instructions are not always executed. In this case, even if some expressions are redundant on specific execution paths, they may not be removed as in the case of previous example.

Example.

Consider an expression `a[i]` in line 6 in Fig. 1.5 (a). If the condition of the *if* statement is *true*, the expression is redundant; otherwise, it is not. An expression that is redundant on some, but not all, execution paths is called *partially* redundant.

End of Example.

PRE makes partially redundant expression *fully* redundant by inserting expressions.



(a) (b)

Figure 1.5: Using PRE to remove partially redundant expression. (a) Original code. (b) Result of PRE.

Example.

In Fig. 1.5 (a), PRE inserts statement `t=a[i]` into *then* and *else* parts as shown in Fig. 1.5 (b); therefore, the partially redundant `a[i]` becomes fully redundant. Finally, all the fully redundant expressions can be removed by replacing them with `t`, which results in a program as shown in Fig. 1.5 (b).

End of Example.

1.4 Contribution

To improve the efficient utilization of the memory hierarchy, we propose three new optimization techniques based on PRE, *PRE-based scalar replacement* (PRESR) [76], *global load instruction aggregation* (GLIA) [77], and *multidimensional GLIA* (MDGLIA) [79]. Furthermore, in order to enhance effectiveness of these PRE based techniques, we propose a new PRE technique *effective demand-driven PRE* (EDDPRE) [78] that is based on *demand-driven* data-flow analysis.

The contribution details of this thesis are as follows:

- **Development of effective query propagation for removing redundant array references.**

  Traditional PRE and scalar replacement require the iterative applications of their entire algorithm for removing all redundant expressions. Regarding PRE, because it detects redundant expressions based on their lexical equality, removing lexically different expressions with the

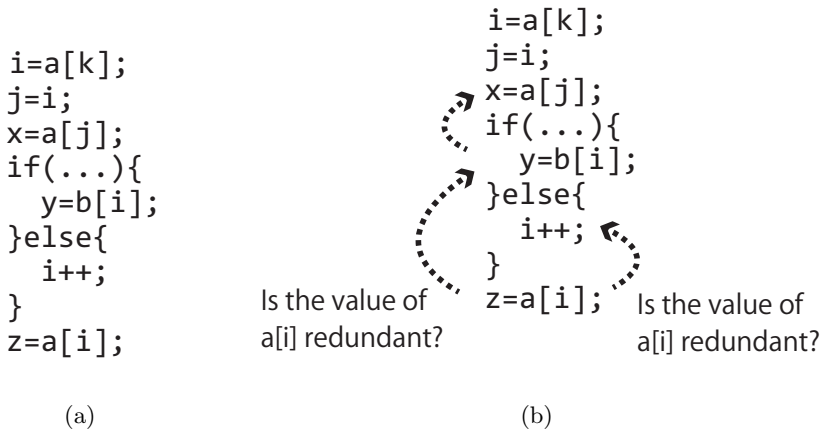


Figure 1.6: Propagating a query for `a[i]` by EDDPRE. (a) Original program. (b) Query propagation.

same value requires the iterative applications of PRE and copy propagation, as explained detail in Chapter 3. Regarding scalar replacement, traditional techniques are applied to each nest-level in a nested loop; therefore, if an array reference can be found to be redundant by analyzing some nest-level at a time, traditional techniques need to be applied iteratively. In contrast, EDDPRE and PRESR can remove redundant array references by only once application. For each array reference, these techniques backwardly propagate a query that can detect redundant array references by checking whether the value of the array reference is redundant or not even if they are lexically different. This is achieved by applying *global value numbering* (GVN) that assigns value numbers to all array reference. If some array references generate the same value, GVN assigns the same value number to the array references; therefore, GVN finds redundant array references, leading to the disuse of copy propagation. Our GVN creates two kinds of occurred value numbers table at each control flow graph (CFG) node in order to suppress unnecessary query propagation. EDDPRE and PRESR propagate queries that check occurrence of the value number of inquired array references on CFG of the program independent of its structure.

Example.

Consider the array reference `a[i]` in Fig. 1.6 (a). Because the values of `i` and `j` are same on a path through the *then* part, the array reference is redundant on the path. On the other hand, these values are different from each other on another path through the *else* part; therefore, the

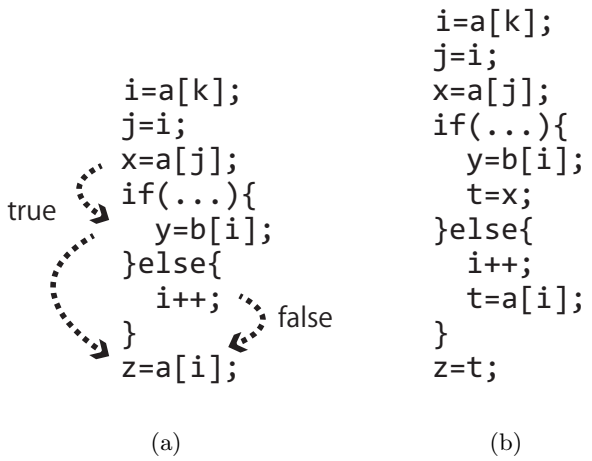


Figure 1.7: (a) Result of query propagation shown in Fig. 1.6. (b) Translating the original program.

array reference is not redundant on the path. That is, `a[i]` is partially redundant. EDDPRE backwardly propagates a query "Is the value of `a[i]` redundant?", as shown in Fig. 1.6 (b). Considering a query that is propagated to the *then* part, it is further propagated before the *if* statement, and then, it gets an answer *true* because it can find that the value of `a[j]` is same as the inquired array reference's value. Considering a query propagated to the *else* part, the query can get an answer *false* without propagating it before the *if* statement because there is no occurrence of value until before the `a[i]`. These results are returned backed to the `a[i]`, as shown in Fig. 1.7 (a). Using these results, EDDPRE finds that `a[i]` is partially redundant; therefore, it inserts `a[i]` into the *else* part. Finally, EDDPRE replaces the inquired `a[i]` with newly introduced an introduced temporary variable, as shown in Fig. 1.7 (b).

End of Example.

- **Decreasing the number of references from processor to cache memory based on effective demand-driven analysis.**

  Removing redundant array references promotes memory references to register references, which means that it promotes even cache references to register references. This is achieved by EDDPRE and PRESR. These techniques remove redundant array references included in an iteration of a loop. In addition, PRESR can also remove array references that are redundant over iterations.

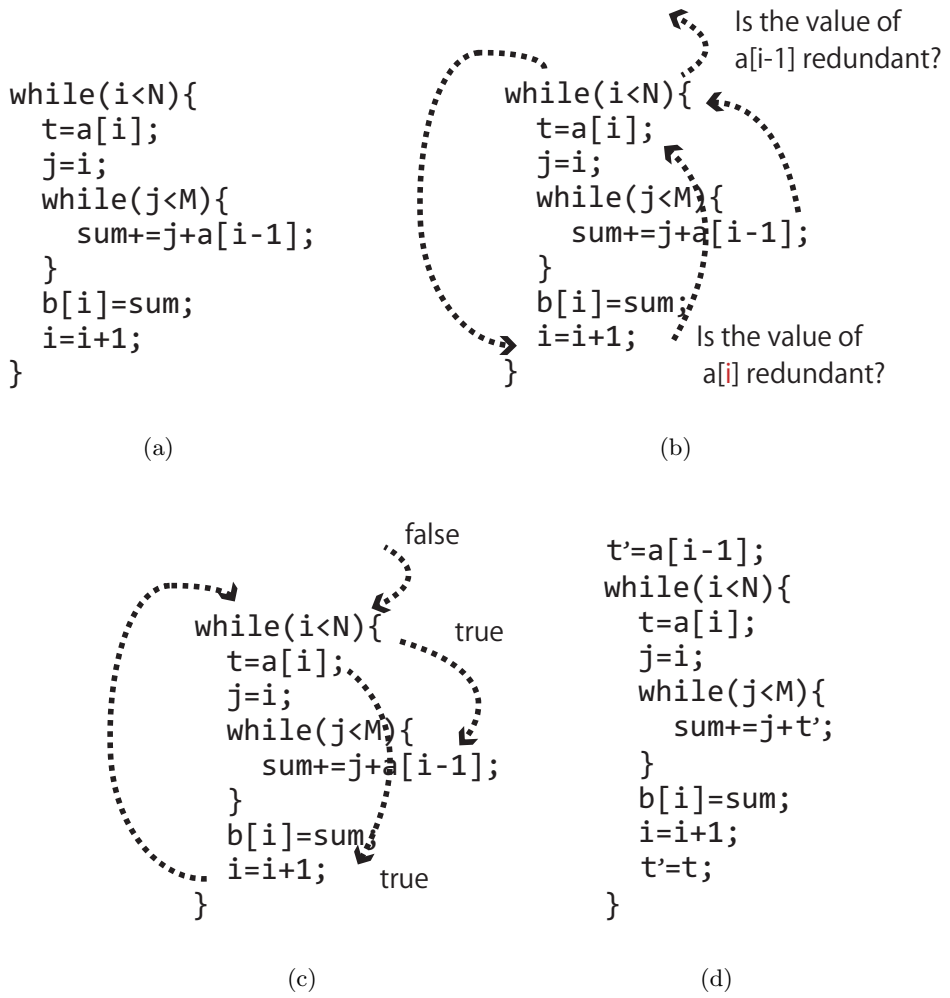


Figure 1.8: Query propagation of PRESR. (a) Original program. (b) Query propagation. (c) Result of query propagation. (d) Result of translating the original program.

Example.

Consider the array reference `a[i-1]` in Fig. 1.8 (a). This array reference is redundant because the referenced data is already loaded to by `a[i]` in the previous iteration. To remove this redundancy, PRESR backwardly propagates a query "Is the value of `a[i-1]` redundant?" as shown in Fig. 1.8 (b). Consider a query that is propagated through the back edge to the exit of the loop body. This query is propagated to the definition `i=i+1` of an induction variable included in the inquired

array reference. To check redundant array references in the previous iteration, the query is changed to check the value of `a[i]` by replacing `i` with the right-hand side of the statement defining the `i`, and then, the query is propagated further. This query will be propagated to `a[i]` below the *while* statement; therefore, it gets *true*. Because a query propagated to outside of the loop gets *false*, this array reference is partially redundant. Similar to EDDPRE, PRESR inserts `a[i-1]` at the entry of the loop, and then it returns *true*, as shown in Fig. 1.8 (c). Finally, PRESR replaces the inquired `a[i-1]` with `t'`, as shown in Fig. 1.8 (d).

End of Example.

- **Decreasing the number of references from cache memory to main memory based on PRE framework.**

  GLIA makes references to the same array continuous by moving the references immediately after preceding other references to the same array. This movement is achieved based on PRE framework. By this movement, some cache misses can be suppressed because data in an array are continuously stored in the main memory. To explain the motivation for GLIA, we show how accessed data are copied into cache memory with an example program in Fig. 1.9. In this thesis, for ease of explanation, we assume that cache memory is directly mapped without loss of generality. That is, when the data are transferred from the main memory to the cache memory, the cache line is determined by the memory address modulo of the number of lines in the cache memory.

  Example.

  Consider the array reference `a[i+1]` in the C program in Fig. 1.9. After execution of the array reference `a[i]`, the data from `a[i]` and `a[i+1]` are copied to a cache line whose index is `10`. Then, because execution of the array reference `b[i]` results in a cache miss, the data contained in `b[i]` and `b[i+1]` are also copied into cache memory. In this case, these data are copied to the cache line at the index `10` because the address of `b[i]` is `1010`; therefore, the data from `a[i]` and `a[i+1]` are removed from cache memory, which may result in a cache miss for the subsequent access of `a[i+1]`.

  As shown in the example, once data at a specific array index is loaded from main memory, it is stored in cache memory along with other data belonging to the same array. That is, continuously accessing the same array may result in cache hits. Continuously accessing the same array can be promoted by moving references to an array around other

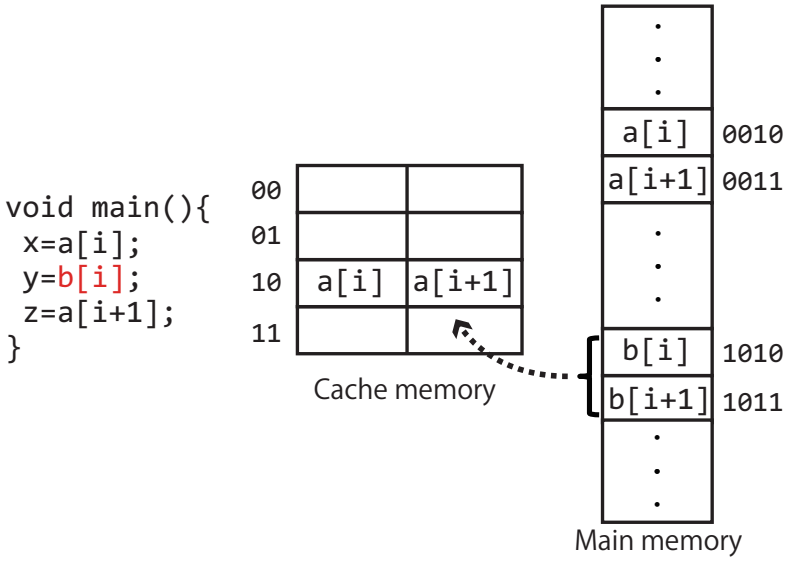


Figure 1.9: Motivation example of GLIA. This figure represents that execution `b[i]` expels the reference data of `a[i]` and `a[i+1]` from the cache memory because the execution results in cache miss.

```
void main(){
 x=a[i];
 z=a[i+1];
 y=b[i];
}
```

Figure 1.10: Moving the array reference `a[i+1]` by GLIA.

reference to the same array. GLIA moves `a[i+1]` immediately before `b[i]` so as to follow `a[i]`, as shown in Fig. 1.10.

End of Example.

In addition, a multidimensional array can be regarded as an array of lower dimensional arrays, which means that it is more effective to continuously aggregate the array references with the most similar indexes in higher dimensions. We extend GLIA to handle multidimensional arrays for moving a reference immediately after the preceding references to the same array with the largest number of similar indexes. We call this extended GLIA MDGLIA. MDGLIA computes the number of indexes of each array reference, preceding a candidate that was moved, which are the same as the indexes of the candidate; then,

MDGLIA moves the candidate to the points in the program close to the references holding the number of the same indexes.

- **Demonstrating the effectiveness of our techniques through experiments employing popular benchmarks.**

  We have implemented EDDPRE, PRESR, GLIA, and MDGLIA as low-level intermediate representation converters in a COINS compiler [24]. We evaluated the effects of our techniques using programs from CFP2000 and CINT2000 in SPEC's Benchmarks. On average, the analyzing cost of EDDPRE was about half of the demand-driven PRE. PRESR improved the performance of all the programs about 2% on average. GLIA and MDGLIA decreased the number of cache misses of many programs.

1.5 Thesis Organization

Chapter 2 presents preliminaries and details the algorithms of two basic optimization techniques, GVN and PRE. Chapter 3 describes the algorithm for EDDPRE, and demonstrates the analytical improvement and efficiency of execution of objective code. Chapter 4 explains the algorithm for PRESR where EDDPRE is extended to remove redundant array references across loop iterations and shows the experimental result. Chapter 5 outlines the algorithm for GLIA and MDGLIA and the experimental results from suppressing the number of cache misses. Chapter 6 concludes this thesis and indicates future directions of this thesis.

Chapter 2

Background

In this chapter, we define the program representation and control flow graph, and then provide the details of the data-flow analysis. Finally, we describe the algorithms of GVN and PRE that are the basis of our technique.

2.1 Preliminaries

2.1.1 Program Representation

Many modern compilers consist of five stages: lexical analysis, syntax analysis, semantic analysis, code optimization, and code generation. Lexical analysis creates a sequence of tokens from a source program, and then syntax analysis builds abstract syntax trees (ASTs) from those tokens. The ASTs are checked for their consistency with declarations by semantic analysis such as type checking. After that, the ASTs are transformed into an *intermediate representation* (IR). The IR code is used as both input and output for optimizations that are independent of programming languages and machines.

We assume that the IR code is a sequence of statements, that have at most one operator, known as *three-address code* [2], as the following:

$$x \leftarrow y \ op \ z$$

2.1.2 Control Flow Graph

We assume that a *control flow graph* (CFG) has been built for each program. The CFG is a graph structure, which is represented as a quadruple (**N**, **E**, **start**, **end**), where **N** is a set of node, **E** denotes a set of edges $\mathbf{N} \times \mathbf{N}$, **start** is a start node with an empty statement, and **end** is an end node with

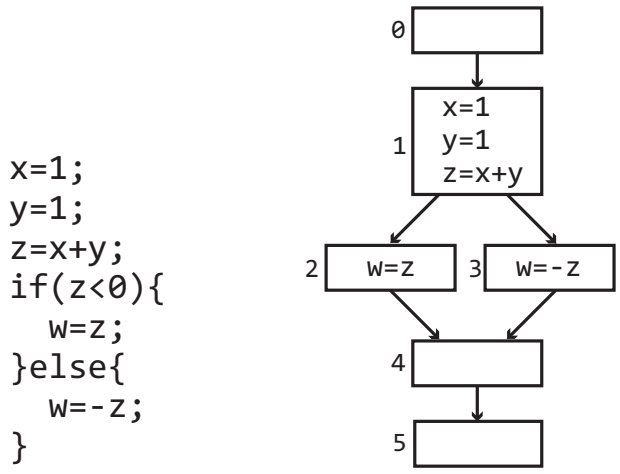


Figure 2.1: (a) Original code. (b) An example CFG of the code. Each node corresponds to a basic block.

an empty statement. [1]

Example.

Figure 2.1 (b) shows a CFG of Fig. 2.1 (a). As the left side code has one *if-else* statement, it has four basic blocks. These basic blocks correspond to Nodes 1, 2, 3, and 4 in the CFG. In particular, Nodes 0 and 5 are called the start and end node, respectively.

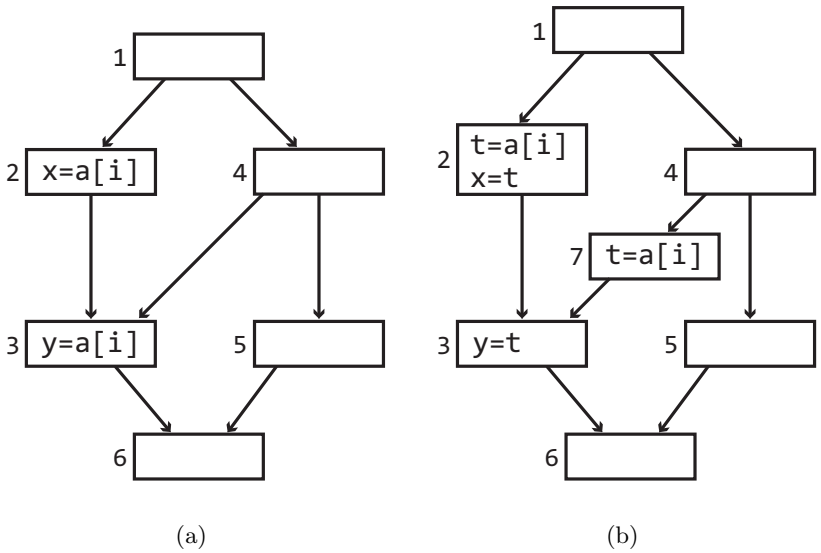
End of Example.

A given edge is expressed as $(m, n) \in \mathbf{E}$, where m is referred to as a predecessor of n and n is known as a successor of m. In general, a node has several predecessors and successors because of the nondeterministic branching structure of a CFG. Here, the sets of predecessors and successors of node n are denoted as $pred(n)$ and $succ(n)$, respectively.

When all the paths from **start** to node n include node m, it is said that m *dominates* n [4]. If m dominates n and m is not n, it is said that m *strictly dominates* n. An edge $a \rightarrow b$ where the destination b dominates the source a is called a *back edge* [2].

In the CFG, it is assumed that the *critical edges*, which are edges leading from a node with more than one successor to a node with more than one predecessor, have been removed by inserting synthesized nodes, because the

[1] In this thesis, we sometime omit the start node and the end node to simplify explanation.



(a) (b)

Figure 2.2: Removing critical edge. (a) Original code. (b) After removing the critical edge and applying PRE.

critical edges can block effective code motion.

Example.

In Fig. 2.2 (a), there is a critical edge from Node 4 to Node 3. As `a[i]` at Node 3 is partially redundant, it can be removed by inserting the same expression at Node 4, but PRE does not allow such an insertion because the insertion introduces a new computation at the path through Nodes 1, 4, 5, and 6. To remove the partially redundant expressions, a new node, Node 7, is inserted on the critical edge as shown in Fig. 2.2 (b).

End of Example.

2.1.3 Data-flow Analysis

Data-flow analysis is performed to collect information for code optimizations, such as constant propagation, copy propagation, common sub-expression elimination, GVN, dead code elimination [6, 53], register promotion, and so on. To present formal definitions, we use Fig. 2.3 to show how the data-flow analysis for a constant propagation collects information.

Example.

Consider applying constant propagation to the code in Fig. 2.3. A variable `a` is initialized to `1` at Node 1, and then it is used at Node 2. To propagate the value of `a` from Node 1 to Node 2, the initial value `1` is

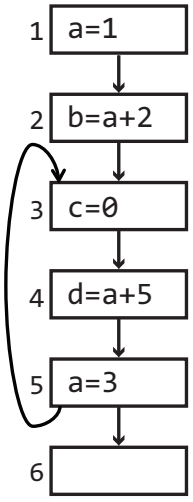


Figure 2.3: An example code for constant propagation.

recorded at the exit of Node 1.

End of Example.

The information collected at the exit of a node n, OUT[n], can be formally defined by a transfer function f, and information collected at the entry of n, IN[n], as follows:

$$\text{OUT}[n] = f(\text{IN}[n]) \tag{2.1}$$

Example.

Consider the statement `a=3` in Fig. 2.3. Once the statement has been executed, the value of `a` is modified to `3`. In other words, the statement *generates* a definition of `a`, `3`, and *kills* all definitions of the variable, `1`, reaching to the node.

End of Example.

The transfer function of a program point x can be formally defined as follows:

$$f(x) = gen \vee (x - kill) \tag{2.2}$$

Consider the statement `a=3` in Fig. 2.3, in which *gen* and *kill* correspond to `3` and `1`, respectively. Here, if a transfer function uses information at the entry point of a node to determine information about the exit point as well as constant propagation, its direction is *forwards*. In contrast, if the function uses information at the exit point of a node to determine information about the entry point, its direction is *backwards*.

To present the formal definition of IN[n], we continue to explain the example of constant propagation in Fig. 2.3.

Example.

We consider the value of `a` that can reach the expression `a+5` at Node 4 in Fig. 2.3. `a` is defined at `1` at Node 1, whereas it is defined to as `3` at Node 5. These values can reach the `a+5` through Node 3 because the former value propagates on a path that consists of Nodes 1, 2, 3, and 4 while the latter value propagates through a back edge and Nodes 3 and 4. This result indicates that it is impossible to statically determine which value is used at the `a+5`. This observation shows that the analysis has to calculate the intersection of values propagated on all paths from **start** to Node 3 that is a join point.

End of Example.

IN[n] is formally defined as follows:

$$\text{IN}[n] \quad = \quad \prod_{\forall_i} F_i(\text{IN}[\textbf{start}]) \tag{2.3}$$

where F_i is the functional composition of the transfer function over the execution path P_i from the **start** node to n_i.

To collect accurate information, the optimizer should calculate equation (2.3); however, in general, it is impossible to do so because of existing loops.

Example.

Consider the number of execution paths from **start** to Node 4 in Fig. 2.3. Let P_4^1 be an execution path that consists of Nodes 1, 2, 3, and 4. Let P_4^2 be an execution path that consists of Nodes 1, 2, 3, 4, 5, 3, and 4. Let P_4^3 be an execution path that consists of Nodes 1, 2, 3, 4, 5, 3, 4, 5, 3, and 4. Furthermore, we can define P_4^4, P_4^5, ..., P_4^n as well as the aforementioned paths. Thus, an infinite number of execution paths can be assumed for a program including the paths within loops.

End of Example.

Therefore, to calculate the data-flow information for any program, equation (2.4) presents an approximate solution as follows:

$$\text{IN}[n] = \prod_{p \in pred(n)} \text{OUT}[p] \tag{2.4}$$

The result of solving equation (2.3) is called the *meet-over-path* (MOP) solution. On the other hand, the approximate solution is called the *maximum fixed point* (MFP), which is determined by repeatedly solving equations (2.1) and (2.4) until the values of IN and OUT are fixed [2]. Notice that,

if the transfer function f of equation (2.2) satisfies distributivity, the MFP solution is the same as the MOP solution [50].

To define formal equations, we used constant propagation whose the transfer function's direction is forwards. In contrast, for an optimizer whose the transfer function's direction is backwards, IN[n] and OUT[n] are formally defined as follows:

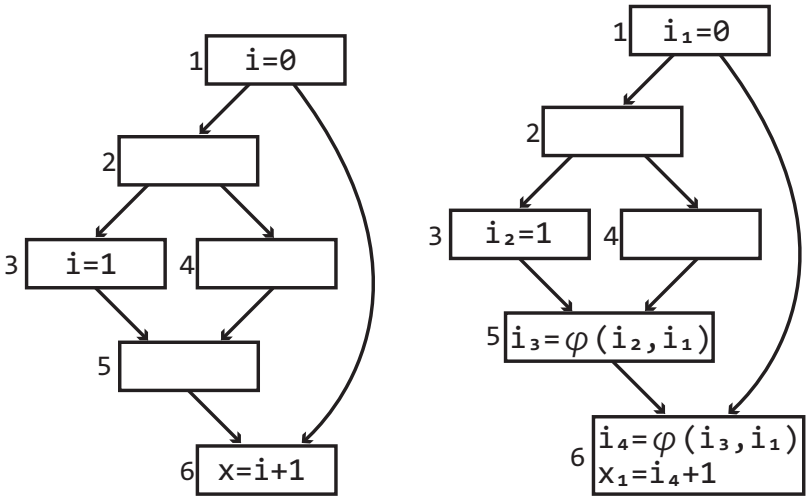
$$\mathrm{OUT}[n] = \prod_{s \in succ(n)} \mathrm{IN}[s]$$
$$\mathrm{IN}[n] = f(\mathrm{OUT}[n])$$

2.1.4 Static Single Assignment Form

Many programming languages support mutable variables. Such variables are useful for writing programs along control flows in imperative programming languages, but at times, this result in complex data-flow analyses because these variables can be modified at any point. If each variable can have only one value without modification, it is possible to simplify data-flow analysis by defining the transfer function without any *kill* term.

Static single assignment (SSA) form is one variety of program representation that allows every variable to be defined only once [4, 9]. In SSA form, the representation with only one definition for each variable is achieved through attaching a unique version number to an original variable name per a definition. Furthermore, in SSA form, a special function ϕ that alternatively returns one of the arguments depending on a control flow is inserted to merge several definitions that reach the same uses. The ϕ function gives the property where each definition dominates all of its uses, so that code optimizers can be more simply described for programs in SSA form. Transformation into SSA form is achieved by the following two steps: 1) inserting ϕ functions, and 2) renaming variables. The ϕ functions can be efficiently inserted at the dominance frontiers of all the definitions of each variable. The dominance frontier of a node n is a set of nodes that n cannot dominate. Notice that the insertions of ϕ functions result in the insertions of new assignments, that have to be considered for additional dominance frontiers. Thus, dominance frontiers can be recursively defined; these are referred to as *iterated dominance frontiers*. The ϕ functions have to be inserted at all the iterated dominance frontiers.

Once ϕ functions are inserted, the variables can be renamed. Remember that each definition dominates all uses. That is, the new version of a variable name that is generated at each definition is only used at nodes dominated by the definition. On the other hand, there are ϕ functions using the definition at immediate successors of nodes dominated by the definition. Therefore,



(a) (b)

Figure 2.4: An example of SSA form. (a) Normal form code. (b) SSA form code.

renaming the definitions and uses can be efficiently achieved by traversing a dominator relation.

Example.

Consider a variable `i` in Fig. 2.4 (a). The variable is defined twice at Nodes 1 and 3. In normal form, they are shared through a common variable `i` which is used at Node 6. In contrast, in SSA form, variable $\mathtt{i}_2$ does not dominate Node 6, but its value may be used at that node. To carry the value to its use, SSA form inserts ϕ functions at the entry of Nodes 5 and 6, and then variable $\mathtt{i}_4$ is defined by Node 6 where it is used as $\mathtt{i}_4$`+1`, as shown in Fig. 2.4 (b).

End of Example.

Three algorithms are known for efficiently transforming code from normal form into SSA form [11, 18, 27]. As the ϕ function is not a real statement that some processors can execute, it is necessary to transform its SSA form into normal form [8, 11, 64, 74].

2.2 Fundamental Code Optimization Techniques

In this section, we explain four compiler optimizations that are the basis of our technique.

```
a=read()          a=read() [1]
b=a+1             b=a+1    [3]
c=a+1             c=a+1    [3]
x=b+5             x=b+5    [5]
y=c+5             y=c+5    [5]
z=x*5             z=x*5    [6]
w=y*5             w=y*5    [6]
```

(a) (b)

Figure 2.5: Assigning value numbers. (a) Original code. (b) After value numbering. The numbers with "[]" represent the value numbers of the corresponding expressions.

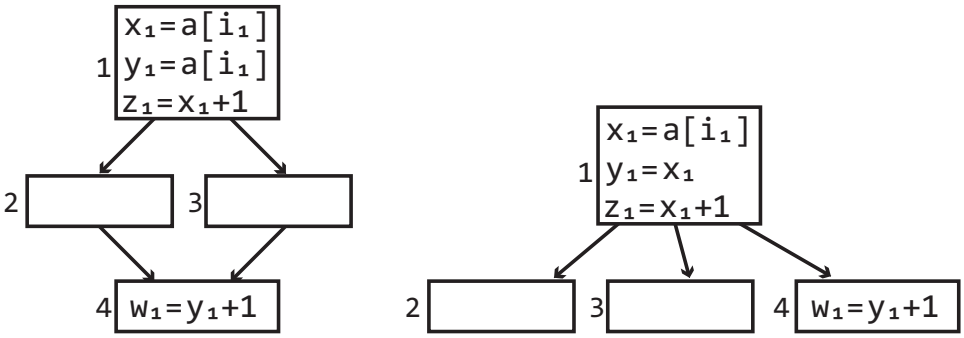
2.2.1 Global Value Numbering

Value numbering is a technique that removes redundant computations that generate the same value without depending on their lexical forms in a basic block. Cocke and Schwarts described a local technique that assigns a unique number, called a *value number*, to expressions with the same value numbers for operands and the same operator [23]. In addition, the variables with the value generated by the expressions are assigned the value number. The process of assigning value numbers to expressions and variables is called value numbering. Value numbering is often implemented by recording each value number in a hash-table where it has a tuple of the value numbers of the operands and the operator as the key. If expressions with a value number equal to e are found through their tuple in the hash-table for a basic block, they are congruent with e.

Example.

In Fig. 2.5 (a), two variables `b` and `c` have the same value, `a+1`; therefore, both `b+5` and `c+5` must compute to the same value. The congruence of these expressions results in the same value number for variables `x` and `y`. Similarly, two variables `z` and `w` also have the same value number. Notice here that the value numbers are assigned to not only variables but also constant values. In this example, the constant values `1` and `5` are assigned to value numbers `[2]` and `[4]`, respectively. As a result, the expressions are assigned to value numbers as shown in Fig. 2.5 (b).

End of Example.



(a) (b)

Figure 2.6: Effectiveness of GVN. (a) Original code represented in a CFG. (b) Result of applying GVN on a dominance tree. The labels of the tree correspond to the CFG.

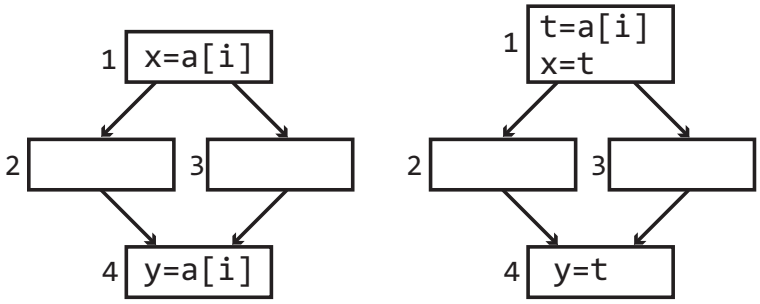
To replace the redundant expression with the variable to which the preceding congruent expression is assigned, each expression in a basic block is processed from top to bottom.

Value numbering locally detects congruent expressions within a basic block, whereas a technique globally detecting congruent expressions in an entire program is called global value numbering (GVN) [3, 21, 35, 37, 59, 65, 67, 69], which is based on a program converted into SSA form. Typical GVN techniques traverse a dominator tree using a depth-first and left-first search, in the process of which once an expression is visited, GVN assigns a value number to the expression. At this time, it records the assigned value number to a hash-table. After value numbering all expressions in a node, GVN propagates the hash-table to the children in the dominance tree.

Example.

Consider two variables $\texttt{x}_1$ and $\texttt{y}_1$ at Node 1 in Fig. 2.6 (a). They are each assigned the expression $\texttt{a[i}_1\texttt{]}$, that is, the same value number is assigned to them, which means that the expression assigned to $\texttt{y}_1$ is redundant; therefore, the expression can be replaced with $\texttt{x}_1$. Then, the value numbers of the three variables are propagated to Nodes 2, 3, and 4, which are children of Node 1 on a dominator tree that is shown in Fig. 2.6 (b). As result of this propagation, $\texttt{y}_1\texttt{+1}$ can be replaced with $\texttt{z}_1$ because $\texttt{z}_1$ is assigned $\texttt{x}_1\texttt{+1}$ that has the same value number as $\texttt{y}_1\texttt{+1}$. The propagation way of the value numbers on the dominator tree guarantees that the value numbers efficiently reach their uses, based on the dominance property of SSA.

End of Example.



(a) (b)

Figure 2.7: Effectiveness of CSE. (a) Original code. (b) Result of applying CSE.

2.2.2 Common Sub-expression Elimination

If more than one expression is executed on a path, and the value would not be changed between them; these expressions, excluding the first one, are said to be redundant on the path. If an expression is redundant on all the paths from *start* to the expression, the expression, which is called a common sub-expression, can be removed by replacing it with a variable with the same value. The optimization is called a common sub-expression elimination (CSE) [22], which contributes to decreasing the number of expressions to be executed in run time. In order to replace a redundant expression with a variable, a temporary variable assigned the preceding expression is introduced as a variable.

Example.

Consider `a[i]` at Node 4 in Fig. 2.7 (a). As the same expression has already been computed at Node 1, the expression is redundant at Node 4. CSE inserts a statement `t=a[i]` before the first occurrence, and then replaces all redundant `a[i]` with `t` as shown in Fig. 2.7 (b).

End of Example.

2.2.3 Loop Invariant Code Motion

If an expression is computed in a loop without any change in its value, it is called a loop invariant expression. In general, executing loop invariant expressions outside the loop remarkably decreases the execution cost because the code inside the loop can be repeatedly executed a lot of times. Loop invariant code motion (LICM) moves the loop invariant expressions out of loops.

```
i=1;
while(i<1000){
  x=a[0];
  i+=1;
}
print(x);
```

(a)

```
i=1;
x=a[0];
while(i<1000){
  i+=1;
}
print(x);
```

(b)

Figure 2.8: Effectiveness of LICM. (a) Original code. (b) Result of applying LICM.

Example.

Consider the `a[0]` in the while loop in Fig. 2.8 (a). As the expression is an array reference with a constant as the index, and without any modification within the loop, the expression is loop invariant. If the expression is moved outside the loop, as shown in Fig. 2.8 (b), the number of executions of the expression is reduced from 1,000 to 1.

End of Example.

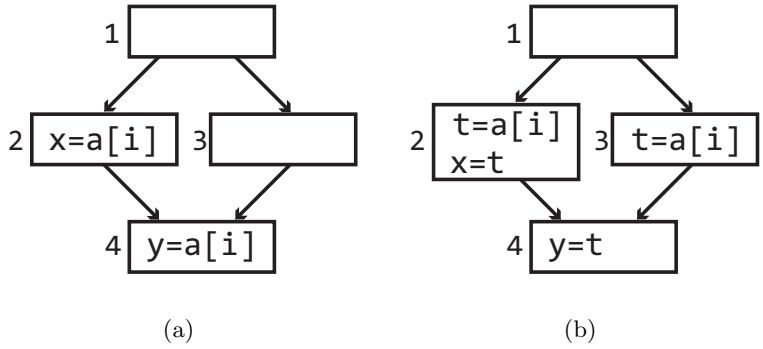
2.2.4 Partial Redundancy Elimination

If an expression e is redundant at n on all paths from **start** to n, it can be removed by CSE. Such an expression is said to be *fully* redundant. That is, CSE removes only fully redundant expressions. On the other hand, CSE cannot remove an expression that is redundant at n on some (but not all) paths from **start** to n. Such an expression is said to be *partially* redundant.

PRE makes a partially redundant expression fully redundant by inserting some expressions at preceding points and then removing the original expression. In PRE, a loop invariant expression can also be regarded as a partially redundant expression because it is not redundant on a path from the entry of loop, but it is redundant on a path from the exit of loop body to the entry. Once PRE is applied to the loop invariant expression, some expressions are inserted outside the loop, and the original expression inside the loop is removed. This process corresponds to LICM. Thus, PRE includes the effectiveness of CSE and LICM.

Example.

Consider `a[i]` at Node 4 in Fig. 2.9 (a). The expression is redundant on a path through Node 2 whereas it is not redundant on another path. To remove the redundant expression, PRE inserts statements `t=a[i]` at Nodes



(a) (b)

Figure 2.9: Effectiveness of PRE. (a) Original code. (b) Result of applying PRE.

2 and 3. As the insertions make the original expression fully redundant, it can be removed as shown in Fig. 2.9 (b).

End of Example.

The first PRE algorithm was proposed by Morel and Renvoise in 1979 [57] as a bi-directional data-flow analysis. Although Morel and Renvoise's algorithm is a powerful technique, it has two issues: 1) some redundant expressions cannot be removed, and 2) some insertions are ineffective, where ineffective insertions lengthen the lifetime of temporary variables, leading to an increase of register pressure. To improve the effect of PRE, many techniques have been proposed to achieve optimal completeness, computation, lifetime, or cost [5, 40, 51, 52, 68, 70], SSA-based sparse analysis algorithm [19, 49, 60, 81], speculative inserting based on profile [13, 38, 46, 72, 84, 87], removing redundant memory reference instructions [32, 48, 54, 55], and so on [10, 28, 29, 30].

In the rest of this section, we define the basic properties of PRE, *availability* and *anticipability*, and then we explain *lazy code motion* (LCM) [51, 52] that is one of the most popular technique of PRE and the basis of our technique.

Basic Properties

Expression e is *available* at node n iff e is computed on any path p from **start** to n, and there is no definition of e's operands since the most recent occurrence of e on p [5]. When e is available at n, n is *up-safe* with respect to e, and e is *partially available* at node n iff there is at least one path from **start** to n in which e is computed without subsequent redefinition of its operands. When e is available at n, e is fully redundant and can be replaced with the variable that has the preceding execution result. When e

is partially available at n, e is partially redundant. The partially redundant expression can be eliminated after inserting expressions to make the original expression fully redundant.

Expression e is *anticipable* at node n iff e is computed along any path r from n to **end**, and the operands of e are undefined before the first computation of e on r [5]. When e is anticipable at n, n is *down-safe* with respect to e. PRE inserts expressions at the down-safe nodes without extending the lengths of any path. If some expressions are inserted at non down-safe nodes, the insertions are *speculative*. The speculative insertions allow loop invariant expressions inside a zero-trip loop such as a while loop to be moved out of the loop.

Lazy Code Motion

PRE removes redundant expressions by inserting some expressions. However, the insertions and removals process tends to lengthen the live-ranges of variables carrying preceding expression values to their uses, so that the variables may be spilled in the register allocation phase [12, 17, 20, 36, 61, 63, 83]. To address the problem, LCM consists of the first code motion hoisting expressions as early as possible and the second code motion delaying them as late as possible. The first code motion contributes to eliminating all removable expressions, and the second one contributes to minimizing the live-ranges of variables. These code motions have to satisfy down-safety and up-safety.

Down-safety is used to ensure that LCM does not introduce a new occurrence of the inserted expression on any execution path. Down-safety is represented by predicate *DownSafe*. In addition, up-safety is used to ensure that there are some paths where the number of expressions is decreased by the insertions and removals. Up-safety is represented by predicate *UpSafe*. These safeties are defined based on the local properties $Comp(n)$ and $Transp(n)$. These predicates denote that n contains an occurrence of e, and no operands are defined or modified in n, respectively. In particular, the property represented by $Transp(n)$ is called the *transparency* at node n. To explain the algorithm, in the rest of section, we assume that each node has only one statement, and e represents the concerned expression to help understand the algorithm. The local properties are defined as follows:

Definition 2.2.1 (Local properties of LCM)**.**

$$\begin{aligned} Comp(n) &\overset{def}{\Leftrightarrow} rhs(n) = e \\ Transp(n) &\overset{def}{\Leftrightarrow} Def(n) \notin Var(e) \end{aligned}$$

In the equations, functions $rhs(n)$, $Def(n)$, and $Var(ar)$ return the right-hand side of statement at n, a variable defined at node n, and a set of

variables which are used in ar, respectively. Using these local properties, predicates *DownSafe* and *UpSafe* are defined as follows:

Definition 2.2.2 (Safety)**.**

$$\begin{aligned} DownSafe(n) &\overset{def}{\Leftrightarrow} (n \neq \textbf{\textit{end}}) \ \wedge \\ &\quad (Comp(n) \ \vee \ Transp(n) \ \wedge \prod_{s \in succ(n)} DownSafe(s)) \\ UpSafe(n) &\overset{def}{\Leftrightarrow} (n \neq \textbf{\textit{start}}) \ \wedge \\ &\quad \prod_{p \in pred(n)} Comp(p) \ \vee (Transp(p) \ \wedge \ UpSafe(p)) \\ Safe(n) &\overset{def}{\Leftrightarrow} DownSafe(n) \ \vee \ UpSafe(n) \end{aligned}$$

Example.

Consider partially redundant expression `a[i]` in Fig. 2.10 (a). Once a statement `t=a[i]` is inserted at Node 1, the original expressions at Nodes 4 and 6 can be removed by replacing them with `t`. However, this insertion introduces a new computation on a path through Node 2. If the control flows along the path, the execution time may be increased. To prevent such insertions, LCM checks down-safety at each node. As a result, LCM determines that Nodes 3, 4, 5, and 6 satisfy the down-safety as shown in Fig. 2.10 (b).

End of Example.

After determining the down-safe nodes, LCM determines optimal node where expressions can be inserted. The optimal nodes are determined by the predicates *Earliest* and *Latest*. The predicate $Earliest(n)$ denotes that node n is the closest to **start** of the nodes m satisfying $DownSafe(m)$ or $UpSafe(m)$. The predicate $Latest(n)$ denotes that node n is the closest to the node c that satisfies $Comp(c)$ on each path from *Earliest* to **end**, and there is no node that satisfies the *Comp* on the path from *Earliest* to c. *Earliest* is determined as a maximal fixed point of its data-flow equation. *Latest* is determined based on a maximal fixed point of the data-flow equation of predicate $Delayed(n)$, which denotes that the expression can be delayed until the exit of node n. Notice that *Delayed* is based on the result of *Earliest*; therefore, they have to be computed in proper sequence. These predicates are defined as follows:

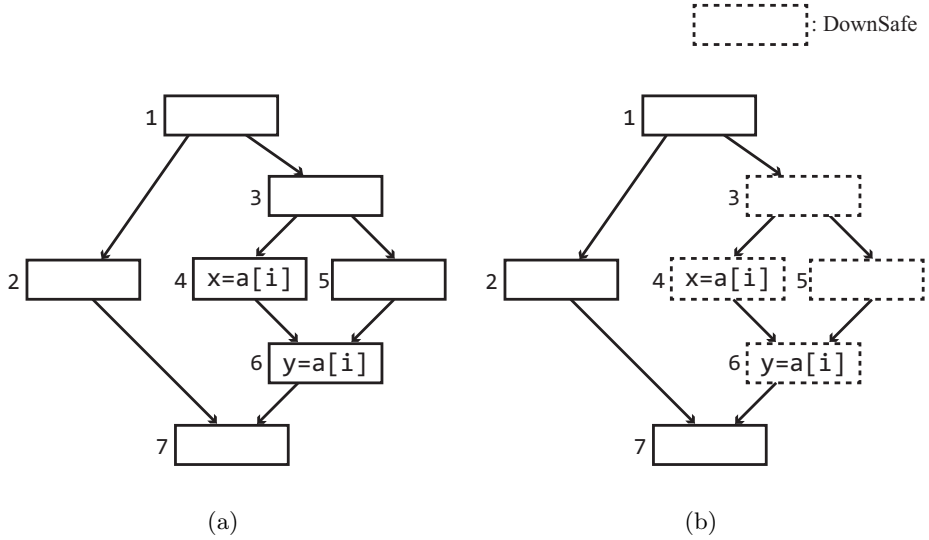


Figure 2.10: Motivation example of LCM. (a) Original code. (b) down-safety of `a[i]`

Definition 2.2.3 (Determine insertion candidate nodes)**.**

$$
\begin{aligned}
\mathit{Earliest}(n) &\overset{def}{\Leftrightarrow} \mathit{Safe}(n) \ \wedge \\
&\quad ((n = \textbf{\textit{start}}) \ \vee \sum_{p \in \mathit{pred}(n)} \neg \mathit{Transp}(p) \vee \neg \mathit{Safe}(p)) \\
\mathit{Delayed}(n) &\overset{def}{\Leftrightarrow} \mathit{Earliest}(n) \ \vee \\
&\quad (n \neq \textbf{\textit{start}}) \ \wedge \prod_{p \in \mathit{pred}(n)} \neg \mathit{Comp}(p) \ \wedge \ \mathit{Delayed}(p) \\
\mathit{Latest}(n) &\overset{def}{\Leftrightarrow} \mathit{Delayed}\,(n) \ \wedge \ (\mathit{Comp}(n) \ \vee \sum_{s \in \mathit{succ}(n)} \neg \mathit{Delayed}(s))
\end{aligned}
$$

Example.

Consider the four nodes with dotted borders that satisfy down-safety in Fig. 2.10 (b). Only Node 3 has a predecessor that does not satisfy the down-safety or up-safety of them. As a result, *Earliest*(3) is *true* whereas others are *false*, as shown in Fig. 2.11 (a). From Node 3, LCM checks whether each node satisfies *Delayed*. As Node 4 satisfies *Comp*, Node 6 does not satisfy *Delayed*. As a result, *Delayed*(3), *Delayed*(4), and *Delayed*(5) are *true*. Finally, the nodes satisfying *Latest* are Nodes 4 and 5, as shown in Fig. 2.11 (b).

End of Example.

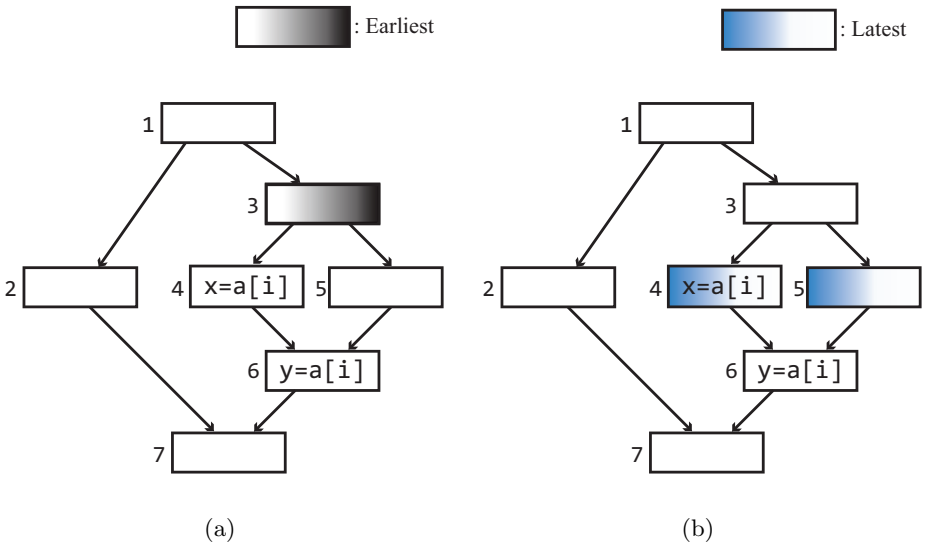


Figure 2.11: Insertion candidates. (a) *Earliest*. (b) *Latest*.

LCM inserts expressions at the entry of nodes n satisfying the predicate $Insert(n)$, where node n also satisfies $Latest(n)$. Notice that LCM does not insert expressions without decreasing the number of executed expressions on some paths because such insertions are unnecessary. In order to exclude the unnecessary insertions, LCM specifies them based on the predicate $Isolated(n)$ that denotes that the insertion at n does not result in removing any expression other than original one; therefore, $Insert(n)$ is defined as $Latest(n) \wedge \neg Isolated(n)$. After insertions at the entry points of nodes n satisfying $Insert(n)$ are performed, the original expressions at nodes n satisfying the predicate $Replace(n)$ that denotes that n contains an e are replaced with the temporary variable holding the value of the expression. These predicates are defined as follows:

Definition 2.2.4 (Translation of the Program)**.**

$$
\begin{aligned}
Isolated(n) &\overset{def}{\Leftrightarrow} \prod_{s \in succ(n)} Latest(s) \vee \neg Comp(s) \wedge Isolated(s) \\
Insert(n) &\overset{def}{\Leftrightarrow} Latest(n) \wedge \neg Isolated(n) \\
Replace(n) &\overset{def}{\Leftrightarrow} Comp(n) \wedge \neg(Latest(n) \wedge Isolated(n))
\end{aligned}
$$

Example.

As shown Fig. 2.12 (a), $Isolated(4)$ and $Isolated(5)$ are *false*. This result indicates that insertions to these nodes are useful for removing redundancy.

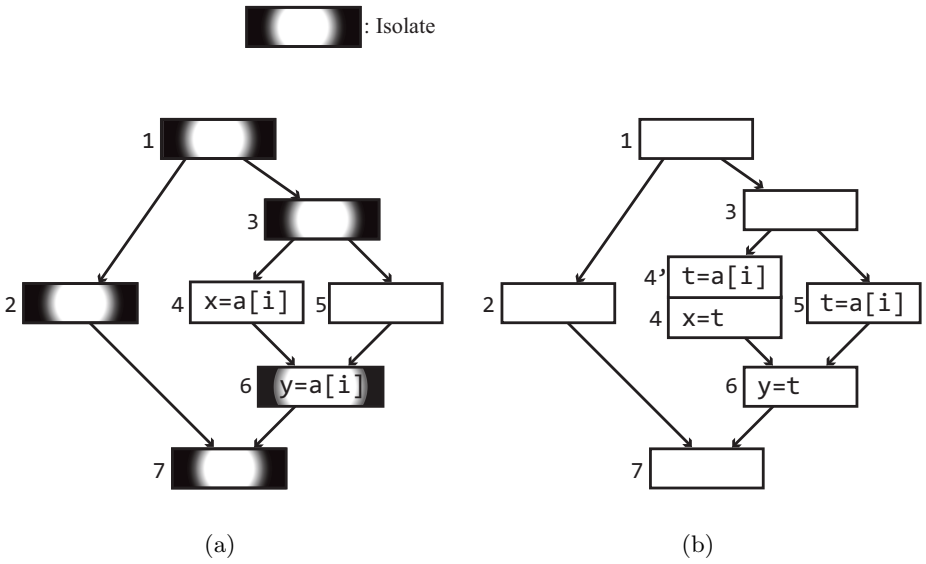


Figure 2.12: Translated program. (a) *Isolate*. (b) Result of applying LCM.

Finally, LCM inserts a statement `t=a[i]` with the temporary variable `t` that is unique for the expression at Nodes 4 and 5, and then replaces the original `a[i]` with `t`, as shown in Fig. 2.12 (b). Here, LCM assumes that each CFG node has only one instruction, and LCM makes a new CFG node 4' for the inserted `t=a[i]` at Node 4.

End of Example.

Chapter 3

Effective Demand-driven PRE

In this chapter, we describe the algorithm of effective demand-driven PRE (EDDPRE). Section 3.1 explains the motivation of EDDPRE. Section 3.2 describes a previous demand-driven PRE. Section 3.3 gives the detail of EDDPRE. Section 3.4 shows experimental results to demonstrate the effectiveness of EDDPRE. Section 3.5 and Section 3.6 summarize, respectively, the related works and EDDPRE.

3.1 Motivation

Traditional PRE exhaustively analyzes the entire program based on dataflow equations, before transforming the program to remove all of the redundant expressions found during the analysis. The removal of the expressions leads to some copy assignments; therefore, the application of copy propagation has the effect of exposing new redundancies, which are known as *second-order effects*. To remove more redundant expressions by capturing these effects, it is necessary to repeatedly apply PRE and copy propagation.

To address the issue, a demand-driven PRE, PRE based on query propagation (PREQP) [80], is proposed. For each expression e, PREQP backwardly propagates a query to determine if e is available. An answer *true* means that e is available on the path on which the query was propagated whereas *false* means that it is not. The query generated at the origin of e is duplicated at a join point that is a program point with several predecessors. Consequently, in case where the answers at the join point are both *true* and *false*, PREQP inserts expressions at nodes where *false* were obtained in order to make e available. After the insertion, PREQP replaces e with an introduced temporary variable, and then PREQP applies demand-driven copy propagation. Although this copy propagation can reveal some new redundant expressions, applying it for all expressions sometimes incur greater

costs than exhaustive PRE. In addition, the query may sometimes be propagated unnecessarily in order to check the redundancy of an expression, even if the expression is not redundant on some execution paths, as explained in the next section.

In this chapter, we propose EDDPRE to suppress unnecessary copy propagation and query propagation. First, EDDPRE applies global value numbering (GVN) in order to statically detect redundant expressions that are lexically different expressions for disuse of copy propagation. During GVN, EDDPRE records occurrences of the value numbers into tables at each CFG node. For each expression, EDDPRE propagates a query that checks whether the value number of the inquired expressions is available or not. This query is propagated to each node while a value number of the inquired expression is recorded in the value number occurrence table.

The advantages of EDDPRE are summarized as follows:

- EDDPRE can capture many second-order effects without copy propagation and iterative applications of whole algorithm.
- EDDPRE suppresses unnecessary query propagations by recording the occurrence of value numbers on paths from the start node to the exit of each node.

3.2 PRE Based on Query Propagation

PREQP is applied to programs translated into SSA form. PREQP visits each CFG node that represents a basic block in topological sort order, and performs query propagation [67] to check whether each expression e can be eliminated at the node. Before explaining the detail, we present an example how PREQP removes the redundant expressions.

Example.

Consider the statement z_1=x_1+1 at Node 3 in Fig. 3.1 (a). The x_1+1 is redundant because there is the statement y_1=x_1+1 that has the same expression in the right-hand side immediately before it. PREQP replaces it with y_1, and then PREQP applies copy propagation in order to capture the second-order effect through replacing z_1 with y_1. The copy propagation traverses Nodes 3, 4, 5, and 6 dominated by Node 3 while replacing z_1 with y_1. Remember here that the uses of each variable are dominated by the definition of it in SSA form. The copy propagation results in the program shown in Fig. 3.1 (b), where it is found that the right-hand side of a statement j_1=y_1+1 is redundant. Then, PREQP is applied to the expression, so that it is replaced it with i_1. After that, the copy propagation is applied again.

Consider the loop invariant expression x_1+2 at Node 5 shown in Fig. 3.2 (a). PREQP propagates a query "is x_1+2 available?" to the predecessor

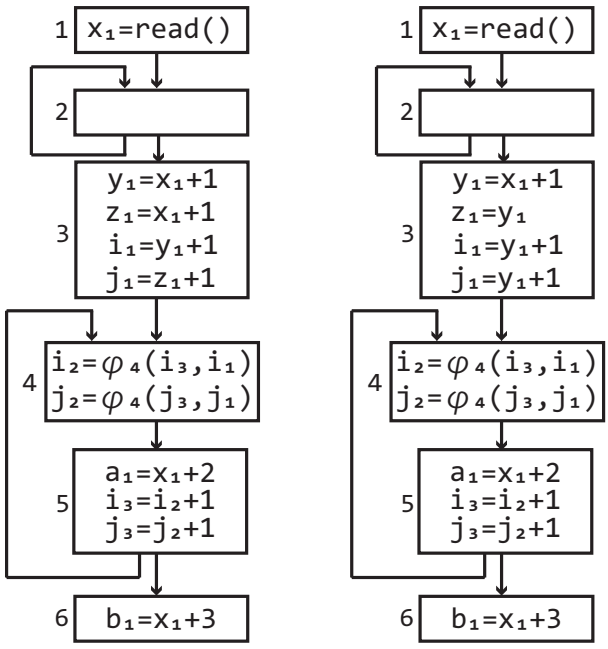


Figure 3.1: Effect of PREQP's removing redundant expression and applying copy propagation. (a) Original program. (b) After removing x_1+1 at Node 3.

4. There is no expression in Node 4; therefore, the query is further propagated to two predecessors 3 and 5. Following this, the query propagated to Node 3 is further propagated to Nodes 2 and 1 in this order. However, the expression is not found, so that the answer *false* is returned to Node 4. By contrast, the query propagated to Node 5 obtains the answer *true* because x_1+2 itself is found at Node 5. The two answers *true* and *false* mean that x_1+2 is partially available at Node 4. In this case, PREQP makes it available by inserting a statement $t_1=x_1+2$ into Node 3, where traditional PREs check down-safety. However, PREQP ignores the down-safety in the case where the query is propagated to the inquired expression itself, so that loop invariant expressions are speculatively moved out of the loop, even if the down-safety is not satisfied. Consequently, the answer *true* is returned to Node 5; therefore, x_1+2 is determined to be redundant. Finally, x_1+2 is replaced with t_1, as shown in Fig. 3.2 (b).

When a query is propagated over a ϕ function that defines some operands of the inquired expression, the operands are replaced with arguments corresponding to predecessors where the query is propagated. For example,

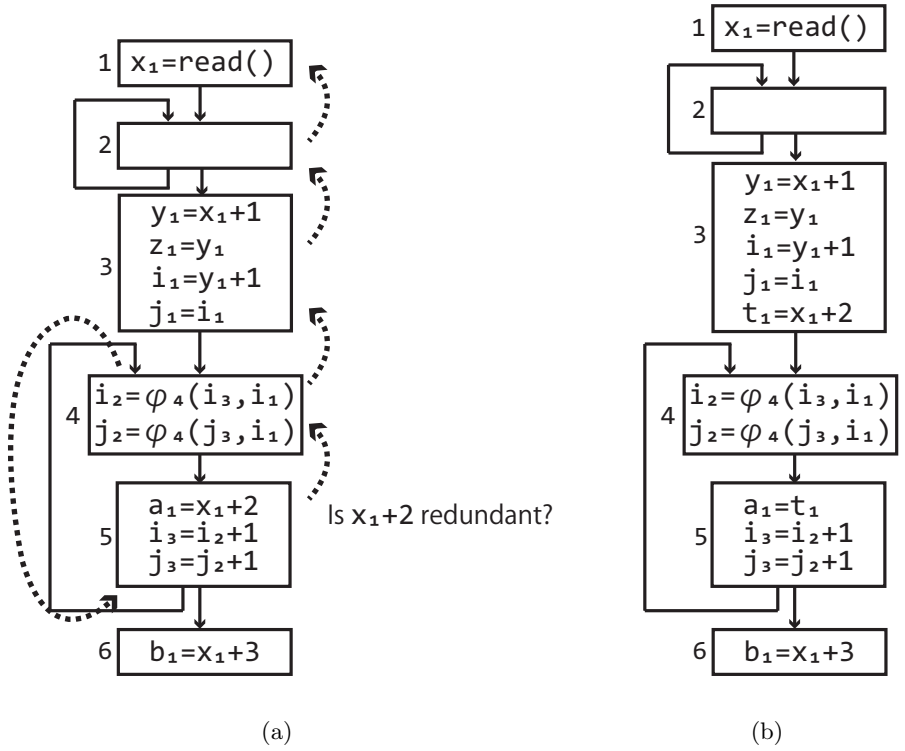


Figure 3.2: Effect of PREQP's query propagation. (a) Example of propagating a query. (b) Result program of applying PREQP

consider a query regarding i_2+1 at Node 5. The query is propagated to Nodes 3 and 5 over a ϕ function $i_2=\phi_4(i_3,i_1)$ after Node 4. In this situation, two new queries "Is i_1+1/i_3+1 available?" are generated for Nodes 3 and 5, respectively. We call this replacement ϕ function replacement.

If a query is propagated twice to a node without ϕ function replacement in a loop, the answer *true* should be returned for the maximum fixed point of the data-flow equations. However, this rule may lead to unnecessary code motion passing some empty loops. For example, in Fig. 3.2 (a), consider that a query regarding x_1+2 is propagated from Node 2 to Node 2 through the back edge. If it is simply assumed that the second visit to Node 2 returns an answer *true*, an expression would be inserted into Node 1 because an answer *false* is returned from Node 1. However, as mentioned above, the expression should be inserted into Node 3; therefore, inserting it into Node 1 unnecessarily extends the live-range of an introduced temporary variable. To suppress the unnecessary insertion, PREQP defines a predicate *isReal* to indicate that the answer *true* is derived from the fact that the query has

reached a real expression. Insertions are allowed only if *isReal* is *true*. Thus, the insertion into Node 1 is suppressed because *isReal*s at Nodes 1 and 2 are *false*, so that, *false* is returned from Node 2.

End of Example.

These observations lead to formal definitions of the answer of a query in PREQP. In this definition, e represents an inquired expression. Once a query is propagated to a node, PREQP determines the answer at the entry of a node under the predicate *NAnswer*. As mentioned above, *NAnswer* is *true* if one of the following two conditions is satisfied: 1) all answers are obtained as *true* from predecessors, or 2) expressions are inserted into predecessors, where the second condition is denoted by *Insertable*. In addition, PREQP represents the answer at the exit of a node as *XAnswer*, which is defined using *Local* that denotes some occurrences the expression in the node. The *N/XAnswer* and *Local* are formally described as follows:

Definition 3.2.1. *When a query is propagated to a node n, the answer is defined as follows:*

$$\begin{aligned} XAnswer(e,n) &\overset{def}{\Leftrightarrow} (n \neq \textbf{\textit{start}}) \wedge \\ &\quad (Local(e,n) \vee Transp(e,n) \wedge NAnswer(e,n)) \quad (3.1) \\ NAnswer(e,n) &\overset{def}{\Leftrightarrow} Insertable(e,n) \vee \\ &\quad \prod_{p \in pred(n)} XAnswer(transPhi(e,p,n),p) \quad (3.2) \end{aligned}$$

where $Transp(e,n)$ indicates transparency that means no operands of e is defined at n, and *transPhi* performs ϕ function replacement. $transPhi(e,p,n)$ replaces each operand of e with a ϕ function's relevant argument that corresponds to predecessor p if n contains the ϕ function.

Definition 3.2.2 (Rules of the local answer for a query)**.** $Local(e,n)$ *is defined by the following rules, which are checked when a query is propagated to node n:*

(1) If n is a node where the query has already been propagated, and the current query is same as the previous one, the answer is *true*.

(2) If n is a node where the query has already been propagated, and the current query is different from the previous one, the answer is *false*.

(3) If n is the original node of the query, and the original query is also the same as the current query, both the answer and $isSelf(e,n)$ are *true* where *isSelf* indicates that the query has been propagated back to the original location of inquired expression.

(4) If n is a node where e occurs in n, both the answer and $isReal(e,n)$ are *true*.

Rule (2) takes account of the situation where some operands of the expression are changed, so that the current value is different from inquired expression. In this situation, the answer *false* should be returned. Other rules correspond with the above explanation.

Insertable gives the details of the condition for insertions. As mentioned above, if answers *true* and *false* are returned to a node where *isReal* and down-safety are *true*, expressions are inserted into predecessors that had returned *false*. Note here that, the down-safety condition is ignored for a loop invariant expression in order to promote speculative code motion out of a loop. The loop invariant expressions are identified through a predicate *isSelf*. These conditions are formally described as follows:

Definition 3.2.3 (Insertion condition)**.**

$$isReal(e,n) \overset{def}{\Leftrightarrow} e \text{ occurs in } n \ \vee \sum_{p \in pred(n)} isReal(e,p) \tag{3.3}$$

$$isSelf(e,n) \overset{def}{\Leftrightarrow} \text{the original expression of } e \text{ is in } n \ \vee \sum_{p \in pred(n)} isSelf(e,p)$$

$$Insertable(e,n) \overset{def}{\Leftrightarrow} \sum_{p \in pred(n)} isReal(e,p) \wedge (DownSafe(e,n) \vee \sum_{p \in pred(n)} isSelf(e,p)) \tag{3.4}$$

PREQP also checks down-safety through query propagation. The answer depends on *transPhi* and *Local*. However, *Local* ignores rule (3) in the query propagation. This definition is presented in that of *DownSafe*.

Definition 3.2.4 (Down-safety based on query propagation)**.** *Down-safety is defined with performing the replacement of a variable defined by a ϕ function with a suitable argument.*

$$DownSafe(e,n) \overset{def}{\Leftrightarrow} (n \neq \textbf{\textit{end}}) \ \wedge \prod_{s \in succ(n)} Local(e',s) \vee Transp(e',s) \wedge DownSafe(e',s) \tag{3.5}$$
$$where\ e' = transPhi(e,n,s)$$

After inserting expressions, PREQP inserts a ϕ function if *NAnswer* is *true* and the size of the node's *pred* is more than 1.

Example.

As mentioned above, the application of PREQP to the program in Fig. 3.2 (a) causes copy propagation twice. Because the copy propagation based on SSA form checks all of the nodes dominated by a node containing the

definition and the dominance frontiers of them; therefore, Nodes 3, 4, 5, and 6 are checked twice.

Consider the expression $\texttt{j}_2\texttt{+1}$, which is redundant because the expression computes the same value as $\texttt{i}_2\texttt{+1}$. However, PREQP cannot eliminate the expression because the query of PREQP pessimistically finds equality between expressions. When a query regarding $\texttt{j}_2\texttt{+1}$ is propagated to Nodes 3 and 5 from Node 4, the new queries $\texttt{j}_1\texttt{+1}$ and $\texttt{j}_3\texttt{+1}$ are generated. These expressions cannot reach any same expression through the further propagation. Finally, considering the expression $\texttt{x}_1\texttt{+3}$ that is not redundant, the fact cannot be found without propagating a query to Node 1 through all the nodes.

End of Example.

Compared with the traditional PRE technique based on data-flow analysis, PREQP can improve the analysis efficiency for many programs because it efficiently captures many second-order effects by repeating the redundancy elimination and copy propagation processes within limited program regions without analyzing the entire program. In some programs, however, the analysis efficiency of PREQP can be worse than the traditional PRE because the nodes traversed during the copy propagation and query propagation in PREQP still include many nodes not contributing to the final result.

3.3 Effective Demand-driven PRE

In this section, we provide the details of EDDPRE. At the beginning, we describe an overview of EDDPRE's algorithm, and then, we present the details of each steps with pseudo code.

3.3.1 Algorithm Overview

EDDPRE consists of the following two steps:

1. Optimistic GVN

   This step identifies the same expressions by assigning value numbers for them. The value numbers can be managed by a hash-table *valueTable* where expressions with the same value number can be assumed to be expressions generating the same value, even if their lexical forms are different. In addition to the hash-table, for each CFG node, GVN creates two tables: a *local value occurrence table* and a *path value occurrence table*. The local value occurrence table of node n records value numbers occurred in n. The path value occurrence table of a node n manages the occurred value numbers on all paths from the start node to the exit of n. This path value occurrence table is used to quickly know the preceding occurrence of inquired expression without

query propagation. In addition, EDDPRE can *optimistically* assigns value numbers to ϕ functions with cyclic dependence inside a loop, which contributes to detecting the same expressions with induction variable.

2. Redundancy Removal

   In this step, redundant expressions are removed through query propagation and transformation. Query propagation is carried out for each expression in topological sort order. Once the query propagation for expression e results in *true*, insertion points are determined. During the transformation, suitable expressions and ϕ functions carrying their values are inserted. Finally, if e is fully redundant, it is replaced with the variable containing the value of the available expressions. Note that EDDPRE speculatively inserts expressions only if e is a loop invariant expression as well as PREQP.

3.3.2 Optimistic Global Value Numbering

GVN traverses the dominator tree in depth-first and left-first search. Once an expression has been visited, GVN assigns a value number to it. The value number of the expression can be obtained through the hash-table *valueTable* by using a variable name or a tuple of an operator and value numbers of operands as a key. The value number of an operand can also be obtained from *valueTable* by using the operand's variable name as a key. If the same tuple has already existed in *valueTable*, the expression is assigned to the value number; otherwise, the expression is assigned to a new value number that is recorded in the hash-table, using the tuple as a key. If the expression is on the right-hand side of an assignment, the variable on the left-hand side is assigned to the same value number. GVN assumes that every function call returns a different value; therefore, each variable defined by a function call is assigned a different value number.

EDDPRE assumes that the child nodes of each node in the dominator tree are sorted in topological sort order. To assign value numbers to arguments of ϕ functions as many as possible before assigning value numbers to the ϕ functions, GVN should traverse in topological sort order in CFG. Traversing the dominance tree in depth-first and left-first search can be proven to correspond with the topological sort order in the CFG through lemmas 1 and 2, where we assume that n_i and n_j are two arbitrary children of node n in the dominance tree, and that N_i and N_j are the subtree node sets that include the n_i and n_j as roots in the dominance tree, respectively. Fig. 3.3 depicts the relation between the subtrees of node n.

Lemma 1. n_{jd} is n_j if there is an edge (n_{is}, n_{jd}) in the CFG where nodes n_{is} and n_{jd} are included in N_i and N_j, respectively.

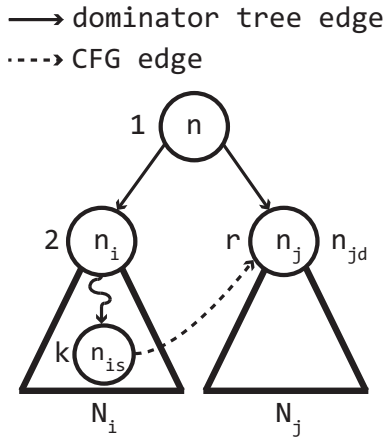


Figure 3.3: Proof of equality between traversals of topological sorted dominance tree in depth-first and left-first search and a topologically sorted CFG. We assume $2 < \mathtt{k} < \mathtt{r}$.

Proof . If we assume that n_{jd} is not n_j, n_{jd} is included in the subtree that has one of the children of n_j as a root. n_{is} is not dominated by n_j; therefore, some paths do not include n_j from the start node to n_{jd} through edge (n_{is}, n_{jd}) in the CFG. This contradicts the fact that n_{jd} is dominated by n_j. □

Lemma 2. $^{\forall}n_{ia} \in N_i$ precedes $^{\forall}n_{ja} \in N_j$ in the topological sort order in the CFG if n_i precedes n_j in the topological sort order in the CFG.

Proof . Assume that N_j includes a node n_{js} preceding $n_{id} \in N_i$ when n_i precedes n_j. At this time, there is an edge from n_{js} to n_{id} in the CFG; therefore, n_{js} precedes n_i according to the lemma 1. In addition, considering that n_s dominates n_{js}, n_j precedes n_{js} in the CFG. As a result, because n_i precedes n_j, This contradicts the assumption. □

Lemma 2 suggests that the depth-first and left-first search order in the dominator tree corresponds to one of the topological sort orders of CFG if children of each node in the dominator tree has been sorted in the topological sort order of CFG.

For an acyclic CFG, the traversal can determine value numbers of all expressions. By contrast, for a general CFG with some cycles, the value numbers of several ϕ functions may not be determined in case where their arguments are defined in the bodies of loops. Assume a ϕ function p_n with arguments *nvargs*, the value numbers of which have not yet been determined, is found at node n. In such cases, in general, the *nvargs* should be conservatively assumed to have different value numbers. However, if all the

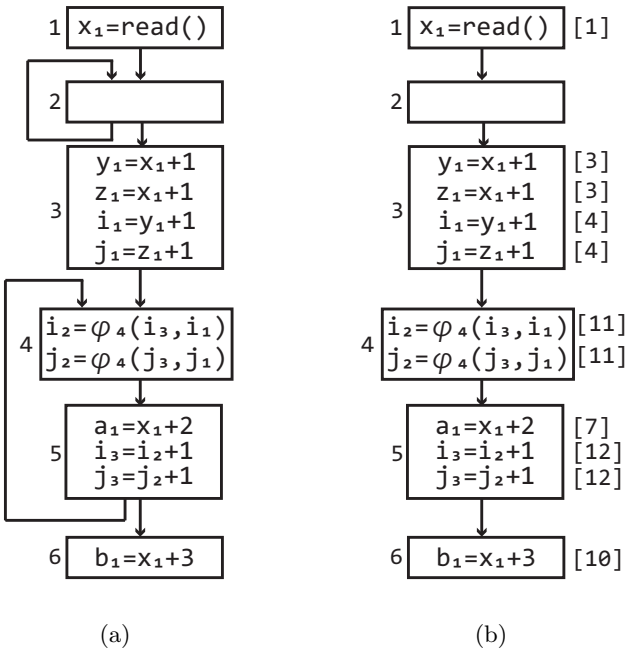


Figure 3.4: Effect of optimistic value numbering. (a) Original program. (b) A result of GVN that is carried out on the dominator tree.

predecessors corresponding to *nvargs* are dominated by n, GVN optimistically assigns the same temporary value number 0 to all *nvargs*, and then continues value numbering for the sub-dominator tree of n. Once GVN has been completed for the subtree, the value numbers of the *nvargs* have been determined; therefore, the value number of p_n can be also determined. At this time, the ϕ functions to which the same value number has been optimistically assigned may be partitioned into different value numbers. In this case, GVN is once again applied to the subtree of n. The optimistic GVN guarantees the assignment of correct value numbers in at most two traversals a ϕ function. Notice there that the repeated traversals are limited to the nodes dominated by the ϕ function.

Example.

Given a dominator tree for the CFG in Fig. 3.4 (a), GVN traverses the dominator tree, shown in Fig. 3.4 (b), using the depth-first and left-first search. Node 1 is visited first, and the expressions in the node are assigned value numbers. After that, Nodes 2, 3, and 4 are processed as well in this order. In Node 3, variables $\mathtt{i}_1$ and $\mathtt{j}_1$ are assigned the same value numbers, and then the ϕ functions are assigned value numbers in Node 4. In this case, although arguments $\mathtt{i}_3$ and $\mathtt{j}_3$ have not yet been assigned value numbers,

optimistic GVN can be applied to the ϕ functions through assigning these arguments the temporary value number 0 because the unprocessed Node 5 of the predecessors corresponding to the arguments is dominated by Node 4. Once value numbers are assigned to statements of Nodes 4, 5, and 6, the ϕ functions are assigned value numbers once again. Finally, expressions are assigned value numbers as shown in Fig. 3.4 (b). In the figures of this thesis, we present each value number as a bracketed number. Note that the constant values `1` and `2` are also assigned value numbers [2] and [6], respectively.

End of Example.

For each node n, after assigning value numbers to all expressions included in n, these value numbers are recorded in a local value occurrence table of n. Moreover, for each CFG node n, GVN creates a path value occurrence table that is defined as the union of the local value occurrence table of n and the path value occurrence table of predecessors of n.

Pseudo Codes

The GVN algorithm is shown in Programs 3.1, 3.2, 3.3, and 3.4. Program 3.1 defines functions for the traversal of a dominator tree and the value numbering of statements. Program 3.2 defines functions for assigning a value number to each statement with hash-table *valueTable*. Program 3.3 defines two functions for checking whether optimistic value numbering should be performed and checking whether the result is correct. Program 3.4 shows a function for making a path value occurrence table for each CFG node.

At the beginning, the function *globalValueNumbering* is called to perform GVN. This function initializes a global variable *value* to generate a new value number when a new entry is added into *valueTable*. After the initialization, to traverse each node, it calls the function *traverseDomTree*. Finally, it calls function *makePathValueTable* to make path value occurrence tables. *traverseDomTree*(n, *optimistic*) calls the function *numbering* to assign a value number to each expression included in n first. If *optimistic* of the arguments is *true*, value numbers are optimistically assigned to the expressions. After the value numbering for the visiting node n, this function checks whether value numbers are optimistically assigned to ϕ functions in the n's children nodes by function *checkPhiArg* defined in Program 3.3. When the optimistic value numbering is performed, *checkPhiVal* is called for checking correctness of optimistic value numbering.

numbering assigns a value number to each ϕ function, each variable assigned the return value of a function call, and each expression. These expressions are identified by functions *isPhi*, *isFunc*, and *isExp*. If the argument *optimistic* is *true*, for checking the correctness of optimistic value numbering, ϕ functions are recoded into table *samePhis*.

Program 3.1 (Traversing of GVN)

Function: *globalValueNumbering*()
1: Initialize a global value *value*.
2: Call *traverseDomTree*(*root*, *false*) to visit the dominator tree.
3: Call *makePathValueTable*() to make path value occurrence tables.

Function: *traverseDomTree*(n, *optimistic*)
// **Arguments**: n is a node of dominator tree. *optimistic* represents
// whether the optimistic GVN is performed.
4: Perform value numbering to n by *numbering*(n, *optimistic*).
5: **for each** child *kid* of n in depth-first and left-first search.
6: **if** (*checkPhiArg*(*kid*))
7: Perform optimistic GVN by *traverseDomTree*(*kid*, *true*).
8: Check correctness of the optimistic GVN by *checkPhiVal*(*kid*).
9: **else**
10: Call *traverseDomTree*(*kid*, *optimistic*).

Function: *numbering*(n, *optimistic*)
// **Arguments**: n is a node of dominator tree. *optimistic* represents
// whether the optimistic GVN is performed.
11: **for each** statement *st* of n.
12: **if** ($isPhi(st) \vee isFunc(st) \vee isExp(rhs(st))$)
13: Get a value number *val* of *st* by *value*(*st*).
14: **if**(*optimistic* $\wedge$ $isPhi(st)$ $\wedge$ *val* is assigned to another ϕ function)
15: Record $lhs(st)$ in table *samePhis* for using in function *checkPhiVal*.
16: Record (*val*, $lhs(st)$) in the local value occurrence table.

Assigning a value number to a statement *st* is performed in function *value*(*st*) defined in Program 3.2. If *st* is a trivial assignment, this function returns a value number of the right-hand side of *st*. Notice that the right-hand and left-hand sides of a statement are represented by $rhs(st)$ and $lhs(st)$, respectively. If *st* is a function call, *value* generates a new value number for *st* and returns the value number. Otherwise, the function translates each operand of expression or argument of ϕ function in the statement into its value number at line 5, which is used to determine the value number of the statement. When a new value number is assigned to *st*, *newValue* increments the global variable *value*, and adds the value number with the expression as a key to *valueTable* before returning the value number. *valueTable* is used for getting the value number of an expression if it includes an entry for the expression, as shown in function *getValue*.

Program 3.3 shows functions *checkPhiArg* and *checkPhiVal* that determine whether the optimistic value numbering is performed, and checks correctness of the optimistic value numbering, respectively. *checkPhiArg* checks whether each argument of the ϕ function has already been assigned a value

Program 3.2 (Assigning value numbers)

Function: *value*(*st*)
// **Arguments**: *st* is a statement.
// **Return value**: a value number of *st*.
1: **if** (*st* is a trivial assignment) **return** *getValue*(*rhs*(*st*))
2: **if** (*isFunc*(*st*)) **return** *newValue*(*lhs*(*st*))
3: **if** (*isPhi*(*st*) $\wedge$ all value numbers of the arguments are same)
4: **return** the value number
5: Make *ve* by changing each operand of *rhs*(*st*) to its value numbers.
6: **return** *getValue*(*ve*)

Function: *newValue*(*exp*)
// **Arguments**: *exp* is an expression.
// **Return value**: a value number of *st*.
7: Increment *value*.
8: Record (*exp*, *value*), *exp* is a key of *value*, in hash-table *valueTable*.
9: **return** *value*

Function: *getValue*(*ve*)
// **Arguments**: *ve* is an expression whose operands are changed to
// corresponding value numbers.
// **Return value**: a value number of *ve*.
10: **if** (*ve* is recorded in *valueTable* as a key) **return** *valueTable.get*(*ve*)
11: **else return** *newValue*(*ve*) // Assign a new value number to *ve*.

number. If some arguments have no corresponding value number, and n dominates the predecessors corresponding to the arguments, the arguments are assigned the temporary value number 0. If n does not dominate the predecessor, *checkPhiArg* returns *false* and assigns a new value number to the ϕ function in a conservative manner.

The function *checkPhiVal* returns *true* if some ϕ functions are assigned the same value number as some entries recorded in *samePhis* that is defined in function *numbering* of Program 3.1. Otherwise, it returns *false* after deleting the ϕ function in *samePhis* because the ϕ function needs to be assigned to a different value number.

After completing value numbering, EDDPRE creates a path value occurrence table for each node by calling *makePathValueTable* shown in Program 3.4. Remember that this function uses CFG nodes rather than dominator tree. This function uses a stack *worklist* to manage nodes. If there are some nodes in the *worklist*, a node n is popped, and then n's path value occurrence table is made. Whenever a value number is added into the path value occurrence table, the successors of n are pushed to *worklist*. The process is repeated until there is no node in the *worklist*.

Program 3.3 (Determine whether optimistic value number is performed and test its correctness)

Function: *checkPhiArg*(n)
// **Arguments**: n is a node of dominator tree.
// **Return value**: Decision of whether the optimistic GVN is performed.
1: **let** *ans* := *false*
2: **for each** ϕ function *phi* included in n
3: **for each** argument *arg* of *phi*
4: **if** (*valueTable* records the value number of *arg* as a key) **continue**
5: **if** (n dominates the corresponding predecessor)
6: Record (*arg*, 0) in hash-table *valueTable*.
7: *ans* := *true*
8: **else**
9: **for each** argument *zarg* that is assigned a value number 0 of *phi*
10: **if** (*valueTable.get*(*zarg*) = 0)
11: Remove the tuple of *zarg* from *valueTable*
12: **return** *false*
13: **return** *ans*

Function: *checkPhiVal*(n)
// **Arguments**: n is a node of dominator tree.
14: **for each** ϕ function *phi* included in n
15: Get a value number v of *phi* by *value*(*phi*).
16: Record (v, *lhs*(*st*)) in the local value occurrence table.
17: Get a ϕ function p included in *samePhis* corresponding to *lhs*(*st*).
18: **if** (v is different from a value number of p)
19: *traverseDomTree*(n, *true*)

Program 3.4 (Make path value occurrence table)

Function: *makePathValueTable*()
// Note: this function is performed on CFG.
1: **let** *worklist* := {*succ*(**start**)}
2: **while** *worklist* $\neq \emptyset$
3: Get a node n by pop from *worklist*.
4: Let *size* be a size of n's path value occurrence table *pvot*.
5: Add value numbers of n's local value occurrence table into *pvot*.
6: **for all** $p \in$ *pred*(n)
7: Add value numbers of p's path value occurrence table into *pvot*.
8: **if** (*size* $\neq$ the size of *pvot*)
9: *worklist.push*(*succ*(n))

3.3.3 Query Propagation

In this section, we explain how query propagation of PREQP is extended to EDDPRE. EDDPRE propagates a query that checks if some expressions

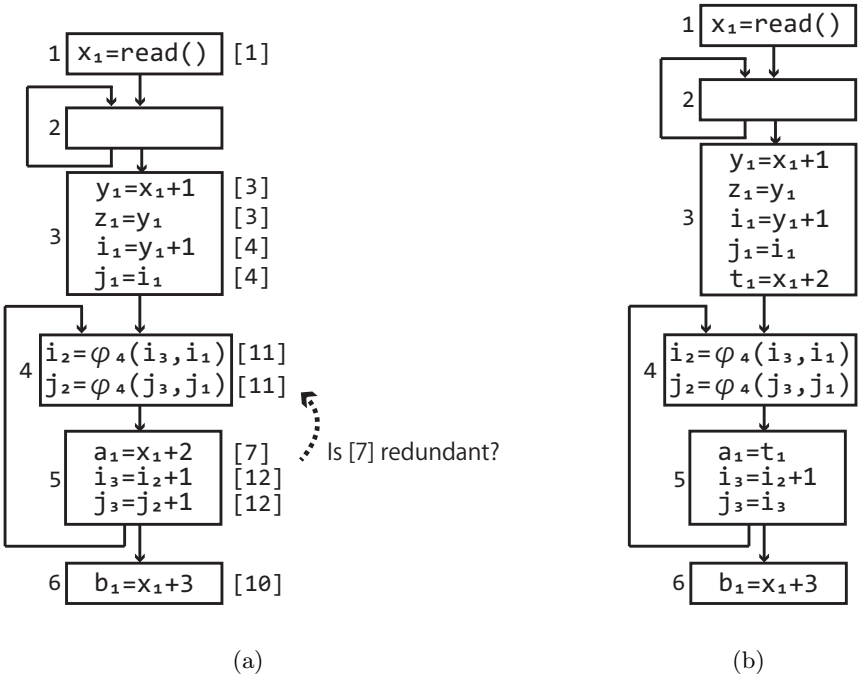


Figure 3.5: Effect of EDDPRE's query propagation. (a) Original program. (b) A result program.

have the same value number as e at each node. Before explain the detail of extension, we show how query propagation of EDDPRE is carried out.

Example.

Consider the loop invariant expression x_1+2 with value number is [7] at Node 5 in Fig. 3.5 (a). EDDPRE propagates a query "is [7] available?" to a predecessor Node 4. There is no expression in Node 4 with value number [7]; therefore, the query is further propagated to two predecessors Nodes 3 and 5. Following this, the query propagated to Node 3 is further propagated to Nodes 2 and 1 in this order, and then the answer *false* is returned to Node 4 as well as PREQP. By contrast, the query propagated to Node 5 obtains the answer *true* because x_1+2 itself is found at Node 5. The two answers *true* and *false* mean that x_1+2 is partially available at Node 4. Similar to PREQP, EDDPRE makes it available by inserting the statement t_1=x_1+2 into Node 3 without checking its down-safety. Finally, EDDPRE replaces x_1+2 with t_1, as shown in Fig. 3.5 (b).

Although PREQP performs a copy propagation for the replaced expression in order to change lexical representation of some expression, EDDPRE disuses copy propagation because of checking value numbering. Further-

more, for the expression $\texttt{x}_1\texttt{+3}$ at Node 6, the first occurrence of the value number is Node 6; therefore, Node 5's path value occurrence table does not contain the value number. That is, EDDPRE results in *false* as soon as propagating a query to only the predecessor Node 5 whereas PREQP has to propagate a query to all the nodes.

End of Example.

To achieve this extension, EDDPRE redefines predicts, *XAnswer*, *isReal*, *DownSafe*, and *Local*, as follows:

Definition 3.3.1. *XAnswer of equation (3.1) and isReal of equation (3.3) are extended for checking value numbers and ignoring transparency as follows:*

$$XAnswer(e,n) \overset{def}{\Leftrightarrow} (n \neq \textbf{\textit{start}}) \wedge (Local(e,n) \vee NAnswer(e,n)) \tag{3.6}$$

$$isReal(e,n) \overset{def}{\Leftrightarrow} \text{a value number of } e \text{ occurs in } n \ \vee \sum_{p \in pred(n)} isReal(e,p)$$

Because EDDPRE checks occurrences with the same value as inquired expression, the answer *XAnswer* is changed to ignore transparency that is used in checking the lexical equality.

Definition 3.3.2 (Down-safety of EDDPRE)**.** *DownSafe of equation (3.5) is also extended to check value numbers as follows:*

$$DownSafe(e,n) \overset{def}{\Leftrightarrow} (n \neq \textbf{\textit{end}}) \ \wedge \prod_{s \in succ(n)} Local(e',s) \vee DownSafe(e',s)$$
$$where\ e' = transPhi(e,n,s)$$

Definition 3.3.3 (Rules for the local answer to a query)**.** *Local*(e,n) *is redefined for checking occurrence of e's value number by the following rules, which are checked when a query is propagated to node n:*

(1) If n is a node where the query has already been propagated, and the value number of the current e is same as the previous one, the answer is *true*.

(2) If n is a node where the query has already been propagated, and the value number of the current e is different from the previous one, the answer is *false*.

(2') If n is a node where no value number of the query occurred on any path from the start to the exit of n, and the value number is not dependent on the ϕ function, the answer is *false*.

(3) If n is the original node of the query, and the original query is also the same as the current query, both the answer and *isSelf*(e,n) are *true*.

(4) If n is a node where the value number of e has been recorded in its local value occurrence table, both the answer and $isReal(e, n)$ are *true*.

Pseudo Codes

To perform query propagation of EDDPRE, EDDPRE calls the function *eliminate* shown in Program 3.5. For each expression, the function analyzes local redundancies by *localMap* recording value numbers for occurrences at the current node first. If the expression is locally redundant, it is replaced with a variable holding the same value. Otherwise, it calls function *propagate* to propagate a query after calling the function *initialize* for initializing the global tables that are used to record expressions and value numbers in query propagation. If the expression is redundant, it is replaced with the suitable variable.

We describe the pseudo code of *propagate* in Program 3.6. *propagate* determines an answer at the entry of node n after propagating queries to predecessors. Lines 2–4 perform the preparation for propagating a new query to the predecessors. *transPhi* of equation (3.2) is applied at line 2, and ne_p is then introduced to hold the return value. After obtaining the new query ne_p, a new value number of ne_p is determined at line 3, and the query is then recorded in order to insert an expression into some predecessors at line 4. At line 5, function *local* is called to obtain an answer that corresponds to *XAnswer*. Following propagations to all predecessors, the answer for n is determined by the answers obtained from its predecessors under equation (3.2). If the answer of this node is *true*, PRESR inserts expressions into the predecessors where the answers are *false*, and then inserts the ϕ function. Function *local* corresponds to equation (3.6).

Program 3.5 (Analysis redundancy)

Function: *eliminate*()

```
 1: for each CFG node n in toporogical sort order
 2:   Make a local occurrence map localMap
 3:   for each statement st of n
 4:     if (¬isExp(rhs(st))∨ st is a trivial assignment) continue
 5:     Get a value number val of st by value(st).
 8:     if (localMap.containsKey(val))
 9:       let predVar := localMap.get(val)
10:       Replace rhs(st) with predVar.
11:     else
12:       Put (val, lhs(st)) into localMap.
12:       Initialize global tables used during propagating a query.
13:       Let originalN be n.
15:       if (propagate(rhs(st), n))
16:         Replace rhs(st) with the introduced temporal variable.
```

Program 3.6 (Query propagation)

Function: $propagate(e, n)$
// **Arguments**: The inquired expression e, and visiting CFG node n.
// **Return value**: A tuple of *isAvail*, *isReal*, and *isSelf* that denote the
// answer of query at n, occurrence of e, and visited itself, respectively.
1: **for each** $p \in pred(n)$
2: Make a new array reference ne_p by $transPhi(e, p, n)$.
3: Determine a value number val_p of ne_p.
4: Record ne_p in order to insert it at the exit of p later if it is necessary.
5: $(isAvail_p, isReal_p, isSelf_p) = local(val_p, ne_p, p)$
6: **if** $(Insertable(e, n) \vee \prod_{p \in pred(n)} isAvail_p)$ // equation (3.2)
7: Insert ne_p made at line 4 into predecessors whose $isAvail_p$ is *false*.
8: Insert a ϕ function into the entry of n.
9: **return** $(true, \sum_{p \in pred(n)} isReal_p, \sum_{p \in pred(n)} isSelf_p)$
10: **else**
11: **return** $(false, false, false)$

Function: $local(val, e, n)$
// **Arguments**: The inquired expression e, and visiting CFG node n.
// **Return value**: A tuple of *isAvail*, *isReal*, and *isSelf* that denote the
// answer of query at n, occurrence of e, and visited itself, respectively.
12: **if** (n is the start node) **return** $(false, false, false)$
13: **if** (condition of Rule (1) is satisfied) **return** $(true, false, false)$
14: **if** (condition of extended Rule (2) is satisfied) **return** $(false, false, false)$
15: Record val in order to check Rules (1) and (2)
16: **if** (val is recorded in local value occurrence table of n)
17: **if** (n equals to *originalN* and e is same as the original expression)
18: **return** $(true, true, true)$ // Corresponding to Rule (3)
19: **else**
20: **return** $(true, true, false)$ // Corresponding to Rule (4)
21: **else if** (n's path value occurrence table does not record $val \wedge \neg dependPhi(e)$)
// Corresponding to Rule (2')
22: **return** $(false, false, false)$
23: **else**
24: **return** $propagate(e, n)$

3.4 Experimental Results

We implemented EDDPRE as a low-level intermediate representation converter using a COINS compiler [24]. To evaluate the benefits of EDDPRE as accurately as possible, we compared EDDPRE with PREQP and PRE*2, which applies PRE twice and copy propagation once between them. We used the machine with Intel Core i5-2320 3.00GHz as a CPU and Ubuntu 12.04 LTS as an OS.

Table 3.1: Execution time of objective code

Programs	A.PRE*2	B.PREQP	C.EDDPRE	(A-C)/A	(B-C)/B
equake	69.1 sec	65.2 sec	65.5 sec	5.2%	-0.5%
art	35.7 sec	36.1 sec	33.6 sec	5.9%	6.9%
mcf	34.3 sec	33.7 sec	33.7 sec	1.7%	0.0%
bzip2	73.3 sec	77.3 sec	75.4 sec	-2.9%	2.5%
gzip	103 sec	100 sec	99 sec	3.9%	1.0%
ammp	119 sec	118 sec	120 sec	-0.8%	-1.7%
vpr	68.4 sec	72.2 sec	65.9 sec	3.7%	8.7%
parser	102 sec	105 sec	104 sec	-2.0%	1.0%
twolf	110 sec	110 sec	108 sec	1.8%	1.8%

Table 3.2: Analysis time

Programs	A.PRE*2	B.PREQP	C.EDDPRE	(A-C)/A	(B-C)/B
equake	1,949 msec	879 msec	677 msec	65.3%	23.0%
art	384 msec	423 msec	271 msec	29.4%	35.9%
mcf	998 msec	866 msec	374 msec	62.5%	56.8%
bzip2	1,018 msec	1,104 msec	745 msec	26.8%	32.5%
gzip	2,669 msec	1,952 msec	1,087 msec	59.3%	44.3%
ammp	10,959 msec	6,035 msec	3,532 msec	67.8%	41.5%
vpr	5,047 msec	4,574 msec	2,498 msec	50.5%	45.4%
parser	3,744 msec	3,945 msec	2,265 msec	39.5%	42.6%
twolf	36,012 msec	14,484 msec	10,546 msec	70.7%	27.2%

We evaluated the effects of EDDPRE using three programs (equake, art, and ammp) from CFP2000 and six programs (mcf, bzip2, gzip, vpr, parser, and twolf) from CINT2000 in the SPEC benchmarks.

Table 3.1 shows the execution time results for PRE*2, PREQP, and EDDPRE. Regarding PREQP and EDDPRE, most of the programs were improved or matched when using EDDPRE. In particular, the efficiency of art and vpr were improved by about 6.9% and 8.7%, respectively. EDDPRE can eliminate more redundancies than PREQP because EDDPRE uses optimistic value numbering. However, compared with PRE*2, moving loop invariant expression speculatively may decrease the execution efficiency as well as PREQP.

Table 3.2 shows the analyzing time results for PRE*2, PREQP, and EDDPRE, all of which were improved by applying EDDPRE. In particular, the efficiency of twolf was improved by about 70.7% compared with PRE*2. The

Table 3.3: The time of query propagation

Programs	A.PREQP	B.EDDPRE	(A-B)/A
equake	661 msec	265 msec	59.9%
art	311 msec	74 msec	76.2%
mcf	667 msec	79 msec	88.2%
bzip2	837 msec	279 msec	66.7%
gzip	1,447 msec	260 msec	82.0%
ammp	4,516 msec	1,102 msec	75.6%
vpr	3,369 msec	773 msec	77.1%
parser	2,964 msec	753 msec	74.6%
twolf	10,662 msec	2,635 msec	75.3%

Table 3.4: The number of nodes which query propagated

Programs	A.PREQP	B.EDDPRE	A-B
equake	31,884	37,328	-5,444
art	10,149	5,949	4,200
mcf	10,271	3,503	6,768
bzip2	37,719	21,626	16,093
gzip	43,167	18,844	24,323
ammp	169,749	90,815	78,934
vpr	108,917	64,483	44,434
parser	96,056	53,484	42,572
twolf	497,177	343,105	154,072

efficiency of mcf was also remarkably improved by about 56.8% compared with PREQP.

Furthermore, Table 3.3 shows the query propagation time results for PREQP and EDDPRE, all of which were improved by applying EDDPRE. In particular, the efficiency of mcf was remarkably improved by about 88.2%. The number of nodes propagated by the queries using the two techniques is shown in Table 3.4, where EDDPRE visited fewer nodes than PREQP other than equake. PREQP does not generate queries for the expressions with operands that are defined in the node, whereas EDDPRE generates queries for the expressions because EDDPRE does not consider the definition. Thus, it is possible to that EDDPRE generates more queries than PREQP. However, the analytical efficiency of EDDPRE was better than PREQP, as shown in Table 3.3, because the answers to queries are obtained immediately for non-redundant expressions and EDDPRE does not need to

apply copy propagation.

3.5 Related Work

The original PRE technique was proposed by Morel and Renvoise [57], which uses bi-directional data-flow analysis. This technique can eliminate some redundant expressions and move loop invariant expressions out of loops, but some redundant expressions are not removed because the technique does not insert expressions at not down-safe nodes.

In general, most PRE techniques increase the register pressure because the live-range of variables are extended by inserting expressions. To suppress the register pressure, a variant of PRE, LCM, was proposed [51, 52]. LCM moves expressions as early as possible as the first code motion, and then delays the expressions as late as possible as the second code motion. The first code motion enables eliminating all of the removable expressions, and the second one enables minimizing the live-ranges of the variables. Bodik et al. proposed another variant of PRE that eliminates all redundant expressions by copying certain parts of the program [5]. However, the copying process can change the reducible loops into irreducible loops. It is not possible for this technique to be used with other optimization techniques that are assumed to be applied to reducible program. For such a technique, the translation from an irreducible program into a reducible program may be effective, but tends to increase the program size. These PRE techniques need to apply copy propagation repeatedly to catch second-order effects. Kennedy et al. proposed PRE on SSA, SSAPRE, which exploits the properties of the SSA form [49]. SSAPRE produces a factored redundancy graph (FRG) for each expression, and then applies PRE to the FRG, which decreases the analyzing cost because of the sparse structure of the FRG. However, some redundancies are not eliminated because SSAPRE is based on the lexical equality of expressions.

GVN was extended from the local approach by Rosen et al. [67]. The GVN utilizes a hash-table to record the value numbers of each node. Expressions are moved up nodes to check whether the expression is recorded in the hash-table of the node, and then redundant expressions are eliminated. This technique uses query propagation and the loop information to analyze fully redundancy; therefore, it depends on the control flow structure. Furthermore, the technique requires the iterative applications of copy propagation repeatedly as well as PREQP. EDDPRE does not depend on any program structure, and performs the analysis efficiently because it does not need to use the loop information and copy propagation. Alpern et al. proposed another GVN technique that uses partitioning on a dependence graph of SSA, which is called value graph, to detect congruent expressions as candidates of redundant expressions. Because their technique calculates congruent parti-

tions as maximal flow points, it can detect congruent expressions with some induction variables [3]. However, redundancy elimination techniques based on their congruence cannot handle some redundant expressions eliminated by traditional techniques because of the value graph structure including ϕ functions. Ruthing et al. proposed a technique that canonicalizes the value graph with regard to the ϕ functions to remove the structure constraint [69], but its analysis is costly. Nie and Cheng proposed a technique based on the SSA form for sparse analysis, which eliminates as many redundant expressions as Ruthing's technique [59]. Click proposed a technique that extended Alpern et al.'s technique by moving loop invariant expressions out of the loops [21]. This technique requires the loop structure information because it moves expressions downward without moving into the loop, after moving upward speculatively.

Cooper and Xu proposed a technique that eliminates all of the fully redundant load instructions, which combined GVN with common sub-expression elimination [26]. This technique handles the redundancies by assigning a value number to a tuple with the operator of the store/load instruction and the value number of the address. VanDrunen and Hosking proposed a technique that eliminates partially redundant expressions based on the value number based method [81]. This technique defines the availability and anticipability based on value number, but needs to be repeatedly applied to eliminate all of the redundant expressions because it is not based on the data-flow equation of pure PRE. Odaira and Hiraki proposed the partial value number redundancy elimination (PVNRE) that combines GVN and PRE [60]. PVNRE maps the value numbers of the ϕ function and its arguments to another value number to eliminate lexically different partial redundancy. PVNRE defines that back edges are not transparent to prevent the movement of expressions with induction variables outside of the loop. The transparency of back edge means PVNRE is only applicable to programs without any irreducible loop. By contrast, EDDPRE does not depend on any control flow structure, and enables moving loop invariant expressions outside the loop speculatively.

3.6 Summary

In this chapter, we proposed a new effective demand-driven PRE (EDDPRE) that eliminates more redundant expressions than previous techniques by using optimistic value numbering and effective query propagation and recording occurrences of the value numbers. To demonstrate its effectiveness, we applied EDDPRE to several benchmark programs, which showed that EDDPRE improved efficiency of the analysis in all cases and the execution efficiency of generated object code in most cases.

Chapter 4

Demand-driven Scalar Replacement

In this chapter, we describe the algorithm of PRE-based scalar replacement (PRESR). Section 4.1 explains the motivation for PRESR. Section 4.2 summarizes the related works. Section 4.3 defines array reference representation. Section 4.4 gives the details of PRESR. Section 4.5 shows experimental results to demonstrate the effectiveness of PRESR, and Section 4.6 summarizes PRESR.

4.1 Motivation

PRE removes redundant expressions, and moves loop invariant expressions out of loops. Traditional PRE lexically detects partially redundant expressions based on data-flow equations, and then inserts expressions at suitable points to make them fully redundant, thereby removing them. PRE is a powerful code optimization technique; however, it poses two issues. First, each application of PRE has the potential for exposing other redundant expressions; therefore, copy propagation must be performed to reveal their lexical redundancy. The process tends to increase analysis costs. Second, PRE cannot be applied across several loop iterations. This restriction does not permit PRE to manage redundant expressions over some iterations, such as array references with various induction variables as their indices.

To address the first issue, we have proposed EDDPRE that is defined in Chapter 3. For each program represented in SSA form, EDDPRE applies GVN, and then for each expression e, backwardly propagates a query to determine if e is available in terms of its value number. An answer *true* means that e is available on the path on which the query was propagated whereas *false* means that it is not. The query generated at the origin of e is duplicated every time it is propagated to a join point. Consequently, in case where the answers are both *true* and *false*, EDDPRE inserts expressions

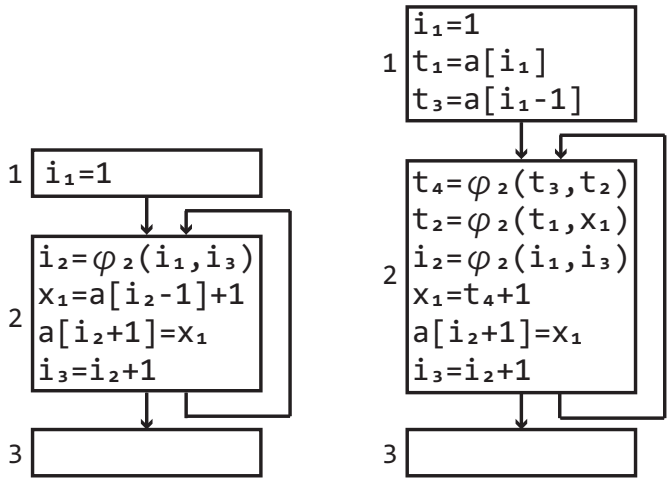


Figure 4.1: Effect of PRESR. (a) Original program. (b) After applying PRESR.

with the value numbers derived from e at each node where a *false* was obtained in order to make e fully redundant.

In this chapter, we address the second PRE issue by extending EDDPRE in order to remove redundant array references across several iterations of a loop in a manner similar to scalar replacement [15, 16, 66]. We call this extended EDDPRE PRESR. Compared to traditional scalar replacement techniques, PRESR can remove redundant array references over iterations with the following two features: 1) PRESR can be applied to any type of control flow structure without altering any loop structure, including irreducible loops, and 2) PRESR determines the most appropriate insertion points in a program through a single application. The first feature is useful because changing program structures may increase its size and number of instructions, such as goto instructions. The second feature contributes to reducing the number of instructions, such as trivial assignments and initializations.

PRESR propagates a query for each array reference. If the inquired array reference has some induction variables in its index, and the related query is propagated to the definition of the variable, then the query is modified by replacing the induction variables with the right-hand side of the definition. This modification enables a query to check previous iterations.

Example.

Consider the array reference `a[i2-1]` at Node 2 in Fig. 4.1 (a). PRESR backwardly propagates a query "Is the value number of `a[i2-1]` available?"

for the array reference from Node 2 to its predecessors, Node 1 and Node 2. At this time, since there is a ϕ function that defines $\texttt{i}_2$ and uses $\texttt{i}_1$ and $\texttt{i}_3$ as arguments at Node 2, queries regarding $\texttt{a[i}_1\texttt{-1]}$ and $\texttt{a[i}_3\texttt{-1]}$ are propagated to Nodes 1 and 2 as well as EDDPRE. Once the query is propagated to Node 1, *false* is obtained as an answer. On the other hand, for the query regarding $\texttt{a[i}_3\texttt{-1]}$ propagated to Node 2, the definition $\texttt{i}_3\texttt{=i}_2\texttt{+1}$ of $\texttt{i}_3$ is found at Node 2. Once a definition of an induction variable such as $\texttt{i}_3$ has been found, the variable is replaced with the right-hand side of its definition– in this case $\texttt{i}_2\texttt{+1}$ –to check availability in the previous iteration. Hence, a new query regarding $\texttt{a[i}_2\texttt{]}$ is generated and continuously propagated. Furthermore, applying the same process to the $\texttt{a[i}_2\texttt{]}$ results in a query regarding $\texttt{a[i}_2\texttt{+1]}$ at Node 2. Because an array reference, $\texttt{a[i}_2\texttt{+1]}$, identical to the inquired expression is found on the left-hand side of a statement at Node 2; therefore, the query gets the answer *true*. Consequently, as both *true* and *false* are obtained at the predecessors of Node 2, $\texttt{a[i}_2\texttt{]}$ is found to be partially available at Node 2. In this case, to make it available, PRESR inserts a statement $\texttt{t}_1\texttt{=a[i}_1\texttt{]}$ into Node 1, where the answer *false* was obtained. Furthermore, to carry the value of $\texttt{x}_1$ or $\texttt{t}_1$ to the point where the query was initiated, PRESR inserts a ϕ function $\texttt{t}_2\texttt{=}\phi_2(\texttt{t}_1,\texttt{x}_1)$ at the entry of Node 2. Following the insertion, the answer of query $\texttt{a[i}_2\texttt{]}$ at Node 2 is found to be *true*. Thus, once a query obtains an answer at a node, the answer is returned to the successor node. Hence, *true* is returned through the back edge to the successor 2. As a result, as well as the preceding process, *true* is obtained at Node 2 for the initial query regarding $\texttt{a[i}_2\texttt{-1]}$ following the insertion of statement $\texttt{t}_3\texttt{=a[i}_1\texttt{-1]}$ into Node 1 and a ϕ function $\texttt{t}_4\texttt{=}\phi_2(\texttt{t}_3,\texttt{t}_2)$ into Node 2. The ϕ function contributes to carrying the value of $\texttt{a[i}_1\texttt{-1]}$ or $\texttt{a[i}_2\texttt{+1]}$. Finally, PRESR replaces $\texttt{a[i}_2\texttt{-1]}$ with $\texttt{t}_4$ as a redundant array reference, as shown in Fig. 4.1 (b). Throughout this process, PRESR inserts compensation code, such as $\texttt{t}_1\texttt{=a[i}_1\texttt{]}$ and $\texttt{t}_3\texttt{=a[i}_1\texttt{-1]}$, at the most suitable points without checking the loop-nest level. Thus, PRESR can remove redundant array references at any level in a nested loop, such as $\texttt{a}_2\texttt{[i}_2\texttt{-1]}$ at Node 5 in Fig. 4.4.

End of Example.

The contributions of PRESR are summarized as follows:

- PRESR uses only pure SSA form for removing array references to be redundant in some iterations.

  PRESR can be easily implemented and combined with other SSA based optimization techniques in any compiler.

- PRESR can be applied to irreducible as well as reducible loops, and removes redundant array references at any nest-level in a nested loop.

Because query propagation does not depend on control flow structures, PRESR can be applied to programs that include some irreducible loops, which may be generated by aggressive optimizations [25], such as instruction aggregation [34] and removing redundant/dead expression[5, 6], apart from the single application of traditional scalar replacements.

- PRESR detects redundant array references, simultaneously inserting initializations for scalar temporaries, the insertion points suitable for which are globally determined as compensation code.

  Traditional scalar replacement techniques first detect redundant array references and then determine where to insert the compensation code. PRESR inserts array references and ϕ functions as compensation code, based on the answers returned from each query.

- PRESR removes redundant array references on demand.

  It is known that demand-driven analysis of PRE is more efficient than exhaustive analysis techniques [78].

4.2 Related Work

In this section, we compare PRESR with traditional scalar replacement and register promotion studies, revealing the differences between them.

4.2.1 Scalar Replacement

Scalar replacement is a code optimization technique that removes reuses of array references beyond loop iterations, to improve the effect of register allocation. The first technique was proposed by Callahan et al. [15]. Their technique detects redundancy by using a dependence graph; therefore, it is able to remove redundant array references included only in the innermost loops that consist of only one basic block.

Carr and Kennedy extended the original technique to manage control flow by incorporating PRE [16]. This is one of the most general techniques for scalar replacement; however, it includes an assumption regarding control flow that may limit its usefulness. The assumption is that this technique can be applied only to the innermost loops which have to be reducible.

Rishi et al. proposed a scalar replacement based on array SSA form that can manage control flow within and across loop iterations [66]. Their technique extended array SSA form, which is proposed in [33], to capture the availability of previous iterations by inserting header ϕ nodes at loop headers. To insert the nodes, it is necessary to assume that every loop has only

one incoming edge. Thus, their technique cannot remove array references that are included in irreducible loops. Although irreducible loops can be converted to reducible ones, the conversion process increases program size and analyzing costs. In addition, their technique introduces scalar temporaries that contain the values of redundant array references and removes the array references by replacing them with the temporaries. These processes are performed on the original input program. The code to initialize the scalar temporaries is inserted in the loop preheader. After transforming the program, the array SSA form must be reconstructed before any subsequent phase that eliminates redundant memory instructions, because the array SSA form depends on the occurrence of memory operations. These analyzing and transforming processes must be applied iteratively in each nesting level of the loop-nest tree beginning with the innermost loop, to remove all redundant array references.

In contrast, the process of analysis and transformation programs used in PRESR is comparatively simpler. If array references are determined to be partially available at certain nodes, modified array references are inserted in nodes that return *false* as a query answer. Because the propagation is independent of the control flow structure, array references can be inserted in the most appropriate nodes over the entire program, e.g. moving loop invariant array references out of loops over the several nesting levels of them. In addition, PRESR inserts ϕ functions to capture different values as well as EDDPRE, because PRESR is performed on pure SSA form. The ϕ functions can also enable reusing values across iterations. That is, PRESR does not require inserting copy assignments in loop bodies, unlike traditional scalar replacement techniques. This process can also be applied to irreducible loops as well as reducible loops without transforming loop structures, including zero-trip loops.

4.2.2 Register Promotion

Lu and Cooper proposed a register promotion that moves explicit memory references outside loops by using point-to analysis with data-flow analysis [56]. Lo et al. proposed a technique based on sparse PRE [55]. Their technique constructs a loop-nest tree, and then it speculatively moves memory references outside loops. These techniques replace memory references with register references, similarly to scalar replacement; however, these techniques do not remove redundancies across several iterations.

Bodik et al. proposed a path-sensitive register promotion that detects numerous redundant references across several iterations [7]. Their technique is based on a *value name graph* (VNG), which consists of address value slices. First, symbolic slices of the address operand are created by backwardly propagating addresses on the CFG. Then, the equality among the slices, which are initially separated, is exposed by GVN, so that address value slices are

obtained. Their technique assumes that the redundancies are removed on the VNG. The VNG creation process is similar to the analysis in PRESR, because these processes use GVN and backwardly propagate addresses or array references on the CFG. However, PRESR removes them immediately after detecting the redundancy, by speculative movement. PRESR can remove redundant array references more rapidly than the Bodik et al.'s approach.

4.3 Array Reference Representation

We assume that every array reference appears only in either the left-hand side or the right-hand side of a statement. That is, a procedure call $f(a[i])$ is split into two statements such as $t = a[i]$ and $f(t)$, and a store statement with nested array references $a[i] = a[a[i-1]]$ is split to three statements such as $t = a[i-1]$, $t_0 = a[t]$ and $a[i] = t_0$.

Similar to EDDPRE, each CFG node represents a basic block in PRESR. In case where store statements can be executed between load statements that access the same address, the results of the load operations may be different. To distinguish these load statements, the different versions are attached to the name of the arrays in the same way as scalar SSA form, such as $a_1[i_1]$.

We assume that the assignment to an array element defines the array itself; therefore, the name of the array is distinguished for each definition of the element. In addition, our SSA form increments the version number for arrays where a ϕ function is inserted in scalar SSA form. These version numbers for the arrays are managed in hash-table $arrayVersion[n]$. This hash-table is used to determine which version number was the largest at the exit of each node whenever a query has been propagated to predecessors.

Example.

Consider the array reference $\mathtt{a}_3\mathtt{[i}_1\mathtt{]}$ at Node 4 in Fig. 4.2. Because there is a store statement at Node 3, the version number is incremented. If it was the definition of a scalar variable, a ϕ function would be inserted at the entry of Node 4. However, it is an array reference; therefore, as mentioned above, PRESR implicitly increments the version number at nodes where ϕ functions should be inserted in scalar SSA form, instead of inserting the ϕ function. Thus, in our SSA form, the left-hand side of the store statement, $\mathtt{a}_2\mathtt{[i}_1\mathtt{]}$, is lexically different from $\mathtt{a}_3\mathtt{[i}_1\mathtt{]}$ in Fig. 4.2, although they contain the same value. To detect the redundancy, PRESR changes the version number when propagating a query to each predecessor by using *arrayVersion* shown in Table 4.1. For example, in Fig. 4.3, PRESR identifies that the $\mathtt{a}_3\mathtt{[i}_1\mathtt{]}$ is partially redundant by propagating $\mathtt{a}_1\mathtt{[i}_1\mathtt{]}$ and $\mathtt{a}_2\mathtt{[i}_1\mathtt{]}$ to predecessors 2 and 3, respectively. Finally, the result shown in Fig. 4.3 can be obtained.

End of Example.

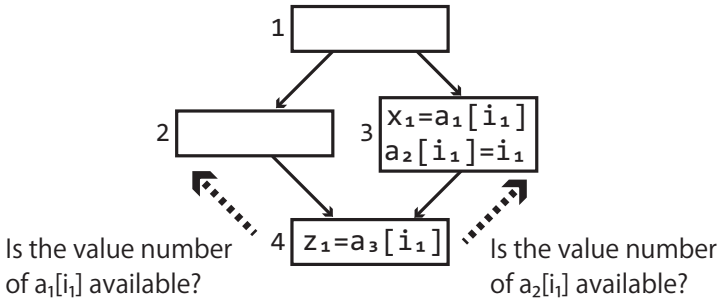


Figure 4.2: An example program after attaching versions for arrays. When PRESR propagates a query for $a_3[i_1]$, its attached version is changed by using *arrayVersion*, as displayed in Table 4.1.

Table 4.1: *arrayVersion*s from Fig. 4.2

node number	array version
1	1
2	1
3	2
4	3

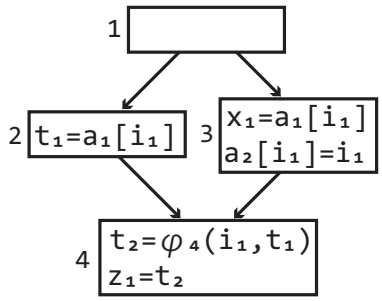


Figure 4.3: Resulting program after applying PRESR to Fig. 4.2.

4.4 Demand-driven Scalar Replacement

PRESR extends optimistic GVN and query propagation of EDDPRE to manage array references. In particular, in the query propagation, some operands of inquired expression have to be replaced with the right-hand side of their definitions when the query is propagated to their definitions in order to achieve the query propagation across several iterations.

4.4.1 Value Numbering for Array References

In addition to assigning a value number to each scalar expression, as defined in Section 3.3.2, PRESR also assigns value numbers to array references. The value number of an array reference $a_1[i_1]$ is defined by the tuple of an operator [] and the value numbers of the start address a_1 and index i_1. If the same tuple is found in the hash-table, its value number is assigned to the array reference; otherwise, a new value number is assigned to the array reference. Subsequently, the value number is recorded, rending the tuple a key in the hash-table. Similar to scalar expressions, these value numbers are recorded in the local value occurrence table and the path value occurrence table at each CFG node.

4.4.2 Redundancy Removal over Iteration

PRESR propagates a query for each array reference. The query propagation is basically same as EDDPRE, except for three extensions. First, *Local* is modified to allow a query to iteratively visit the same node, considering aliases. Second, *XAnswer* is modified for replacement of variables. Third, PRESR weakens the condition for speculative insertion in order to increase opportunities for removing redundant array references.

Extension of *Local*.

As shown how PRESR removes redundant array reference beyond the loop iterations with the motivation example in Section 4.1, a query should be iteratively propagated to a node. Moreover, for safe code motion, the aliases of memory references must be checked. The modified rules of *Local* are as follows:

Definition 4.4.1 (Rules for a local answer to a query)**.** *$Local(e, n)$ is redefined for iteratively propagating queries to a node by the following rules, which are checked when a query is propagated to node n:*

(1) If n is a node where the query has already been propagated, and the value number of the current e is same as the previous one, the answer is *true*.

(2) If n is a node where the query has already been iteratively propagated beyond a certain number of times, according to the number of boundaries, and the value number of the current e is different from the previous one, the answer is *false*.

(2') If n is a node where no value number of the query occurred on any path from the start to the exit of n and the value number is not dependent on the ϕ function, the answer is *false*.

(3) If n is the original node of the query, and the original query is also same as the current query, both the answer and $isSelf(e, n)$ are *true*.

(4) If n is a node where the value number of e has been recorded in its local value occurrence table, both the answer and $isReal(e, n)$ are *true*.

(4') If n contains any aliases of store statements, the answer is *false*.

Rule (2) of EDDPRE returns *false* if a query has already propagated twice to the same node. PRESR increases the number until it is the same as a threshold that is initially given as a parameter *visitLimNum*.

Extension of *XAnswer*.

To check redundancy in a previous iteration, PRESR extends equation (3.6) to replace each variable with the right-hand side of its definition, and then checks redundancy for the new query. Let $replaceOp(e, n)$ be a function that performs the replacement of the variable of e if n is a node where the operands are defined.

Definition 4.4.2 (*XAnswer* of PRESR)**.** *Let e' be the result of replaceOp. To replace variables, XAnswer is redefined as follows:*

$$XAnswer(e,n) \overset{def}{\Leftrightarrow} (n \neq \textbf{\textit{start}}) \wedge (Local(e,n) \vee \begin{cases} NAnswer(e,n) & if\ e = e' \\ Local(e',n) \vee NAnswer(e',n) & otherwise \end{cases}) \quad (4.1)$$

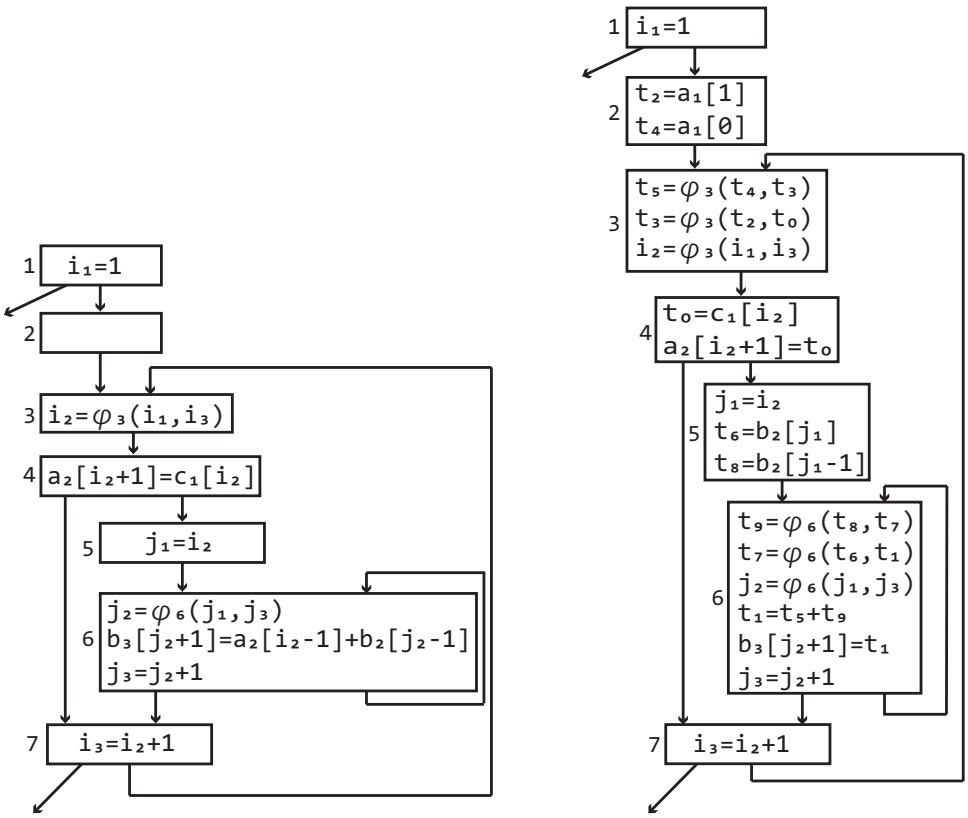
$XAnswer(e, n)$ first checks the occurrence of the value number of e if n is different from the start node. If the value number does not occur in n, *replaceOp* is called. The result of the replacement is represented as e'. Following the replacement, *XAnswer* checks the occurrence of the new value number of e'. If there is no occurrence of the new value number in n, a query for e' is propagated to predecessors by *NAnswer*.

Extension of speculative insertion.

PRESR weakens the condition for the speculative insertion. This insertion is performed when some variables in an expression are replaced with the right-hand side of their definitions. This extension increases the opportunity for removing redundant array references.

Example.

Consider the a_2`[i2-1]` at Node 6 in Fig. 4.4 (a). a_2`[i2-1]` has the same value as the a_2`[i2+1]`, which is defined two iterations prior, and a_2`[i2+1]` is outside the innermost loop. The redundant array reference can be removed



(a) (b)

Figure 4.4: Effect of PRESR. (a) Original program. (b) After applying PRESR.

by inserting two statements, t_2=a_1[1] and t_4=a_1[0], into Node 2, and two ϕ functions, t_3=ϕ_3(t_2,t_0) and t_5=ϕ_3(t_4,t_3), into Node 3 as shown in Fig. 4.4 (b). However, Node 2 is not down-safe for these two statements. That is, in order to remove the array reference, the array references should be speculatively inserted. Note that although b_2[j_2-1] can be removed by traditional scalar replacement techniques, they cannot remove a_2[i_2-1] by single application.

End of Example.

Definition 4.4.3 (Insertion of PRESR)**.** *To speculatively insert array references, Insertable (equation (3.4)) is redefined as follows:*

$$AlgRepl(e,n) \overset{def}{\Leftrightarrow} e \neq replaceOp(e,n) \vee \sum_{p \in pred(n)} AlgRepl(e,p) \tag{4.2}$$

$$Insertable(e,n) \overset{def}{\Leftrightarrow} \sum_{p \in pred(n)} isReal(e,p) \wedge (DownSafe(e,n) \vee \sum_{p \in pred(n)} isSelf(e,p) \vee AlgRepl(e,p))$$

$AlgRepl(e,n)$ indicates that some variables of e are replaced with the right-hand side of the definition at n, or there are predecessors of n that satisfy this predicate.

Pseudo Codes

We show pseudo codes of PRESR's query propagation in Program 4.1 and Program 4.2 that define functions *propagate* and *local*. PRESR extends function *propagate* to check whether some operands are replaced with the right-hand side of their definition for calculating *AlgRepl* of equation (4.2) at line 6. Further, PRESR extends function *local* to check the extended *Local* property at lines 11–17.

Program 4.1 (Query propagation)

Function: $propagate(e,n)$
// **Arguments**: The inquired expression e, and visiting CFG node n.
// **Return value**: A tuple of *isAvail*, *isReal*, and *isSelf* that denote the
// answer of query at n, occurrence of e, and visited itself, respectively.
1: **for each** $p \in pred(n)$
2: Make a new array reference ne_p by $transPhi(e,p,n)$.
3: Determine a value number val_p of ne_p.
4: Record ne_p in order to insert it at the exit of p later if it is necessary.
5: $(isAvail_p, isReal_p, isSelf_p) = local(val_p, ne_p, p)$
6: $algRepl[n] := \sum_{p \in pred(n)} algRepl[p]$
7: **if** $(Insertable(e,n) \vee \prod_{p \in pred(n)} isAvail_p)$ // equation (3.2)
8: Insert ne_p made at line 4 into predecessors whose $isAvail_p$ is *false*.
9: Insert a ϕ function into the entry of n.
10: **return** $(true, \sum_{p \in pred(n)} isReal_p, \sum_{p \in pred(n)} isSelf_p)$
11: **else**
12: **return** $(false, false, false)$

Program 4.2 (Analysis occurrence of value number)

Function: $local(val, e, n)$
// **Arguments**: A value number val, the inquired array reference e, and visiting
// CFG node n.
// **Return value**: A tuple of $isAvail$, $isReal$, and $isSelf$.
1: **if** (n is the start node) **return** $(false, false, false)$
2: **if** (the condition of Rule (1) is satisfied) **return** $(true, false, false)$
3: **if** (the condition of extended Rule (2) is satisfied) **return** $(false, false, false)$
4: Record val in order to check Rules (1) and (2)
5: Increment the visited number of n
6: **if** (val is recorded in local value occurrence table of n)
7: **if** (n equals to $originalN$ and e is same as the original expression)
8: **return** $(true, true, true)$ // Corresponding to Rule (3)
9: **else**
10: **return** $(true, true, false)$ // Corresponding to Rule (4)
11: **else if** (some operands of e are defined in n)
12: Make new array reference ne by $replaceOp$.
13: Let $nval$ be a value number of ne.
14: $algRepl[n] := (ne \neq e)$
15: **return** $local(nval, ne, n, true)$
16: **else if** (the condition of Rule (4') is satisfied)
17: **return** $(false, false, false)$
18: **else**
19: **return** $propagate(e, n)$

4.5 Experimental Results

We implemented PRESR as a low-level intermediate representation converter using a COINS compiler. We evaluated the effects of PRESR using loops from four programs (gzip, parser, bzip2, and twolf) from CINT2000, and two programs (art and equake) from CFP2000 in the SPEC benchmarks. We conducted various experiments on a machine equipped with a Xeon E5-1660 3.3GHz CPU and Debian 64bit OS. We set 3 to *visitLimNum* in this experiment, because this number is sufficient to detect redundancy for SPEC2000 benchmark programs.

To evaluate the benefits derived from PRESR, we compared it with TSR, as follows:

TSR applies traditional scalar replacement that is proposed in [16].

PRESR converts normal form to SSA form, applies PRESR, and converts SSA form back to normal form.

Table 4.2 and Table 4.3 list the execution time and dynamic load instruction count that was measured by PAPI [62] interface when TSR and PRESR were applied to the loops of the programs listed above. The performances

Table 4.2: Results of execution time

Program	TSR Time (sec)	PRESR Time(sec)	Improvement
gzip	77.1	76.0	1.4%
parser	85.0	83.7	1.5%
bzip2	64.6	63.4	1.9%
twolf	100	97.9	2.1%
art	32.6	31.5	3.4%
equake	27.7	27.0	2.5%

Table 4.3: Results of the dynamic number of load statement

Program	TSR Loads	PRESR Loads	Decrease
gzip	9.17E+10	8.58E+10	6.4%
parser	1.41E+11	1.34E+11	4.4%
bzip2	1.06E+11	9.71E+10	8.1%
twolf	1.04E+11	9.93E+10	4.8%
art	7.13E+10	5.12E+10	28.2%
equake	7.83E+10	7.15E+10	8.7%

of all the programs were improved when PRESR was applied. In particular, the execution efficiency of art was improved by about 3.4%. The dynamic number of executed load instructions for art decreased about 28.2%.

Next, we evaluated the analysis costs of PRESR, to demonstrate that PRESR's analysis is more efficient than the iterative application of the exhaustive technique. In this evaluation, we compared PRESR with PRE*2 and EDDPRE_MEM, as follows:

PRE*2 applies PRE twice, and also applies copy propagation between the two applications of PRE.

EDDPRE_MEM applies EDDPRE only to array references.

Figure 4.5 shows the ratio of analysis time of EDDPRE_MEM and PRESR to PRE*2. As demonstrated by the results, comparing PRESR with PRE*2, all programs were improved by applying PRESR. In particular, the improvement was about 67.9% in twolf. In contrast, comparing PRESR with EDDPRE_MEM, the analysis cost increased twice in all programs, though the number of iterative visits was less than *visitLimNum*.

In general, scalar replacement uses *register spill moderation* to suppress spills. In the evaluation, we applied only PRESR or traditional scalar replacement without the register spill moderation. It is straightforward to combine PRESR with the register spill moderation, because PRESR can simply decide not to be redundant without query propagation if the register

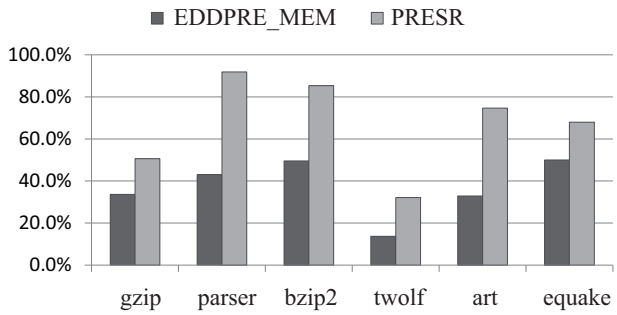


Figure 4.5: Ratio of analyzing costs, compared with exhaustive style of PRE.

spill moderator determines that registers are full. In addition, it would be effective to preferentially remove more costly array references such as [66]. If PRESR could had a list of array references sorted according to their cost, it could stop removing them in the case of no register available for removing them.

4.6 Summary

In this chapter, we proposed a demand-driven scalar replacement technique called PRESR that can remove all redundant array references from any level of loop in a single application without restriction to input control flow structures. PRESR removes redundant array references by applying query propagation. To detect redundancies across several iterations, the operand of the query is replaced with the right-hand side of the definition statement, if the query is propagated to a node that includes the statement. To show its effectiveness, we applied PRESR to several benchmark programs. The result showed that PRESR improved the execution efficiency of objective code in all cases. In future work, we intend to devise an additional technique to removes unnecessary store instructions based on query propagation techniques such as PRESR.

Chapter 5

Global Load Instruction Aggregation

In this chapter, we describe the algorithm of multidimensional global load instruction aggregation (MDGLIA). Section 5.1 explains the motivation of this technique. Section 5.2 summarizes the related works. Section 5.3 gives preliminary definitions needed for the purpose of explaining MDGLIA, and summarizes lazy code motion (LCM). Section 5.4 gives the details of the extension of LCM to MDGLIA. Section 5.5 shows experimental results to demonstrate the effectiveness of MDGLIA. Section 5.6 summarizes MDGLIA.

5.1 Motivation

Most modern processors have some cache memories that are much faster than a main memory. Whenever the processor needs the data at address x in main memory, cache memory is checked whether a copy of the data is stored in the cache memory first. At this time, it is called *cache hit* if the data at x is found in cache memory; otherwise, it is called *cache miss*. In the case of the cache hit, because the data is obtained without any accessing main memory, the program is executed without stalling. Conversely, once the cache miss occurs, the processor fetches the data around x in the main memory and places them in the cache memory for cache hits later. In this case, the reference to x not only causes significant delay because of the fetching but also removes the old data in the same cache line. This means that continuous accesses to addresses that are distant each other in the main memory may result in cache misses, which can decrease the execution efficiency of the program.

Once a data at a specific array index is loaded from main memory, it is placed in cache memory along with other data belonging to the same array. Therefore, if the accesses to the same array are executed continuously, they may contribute to the cache hits. Furthermore, considering

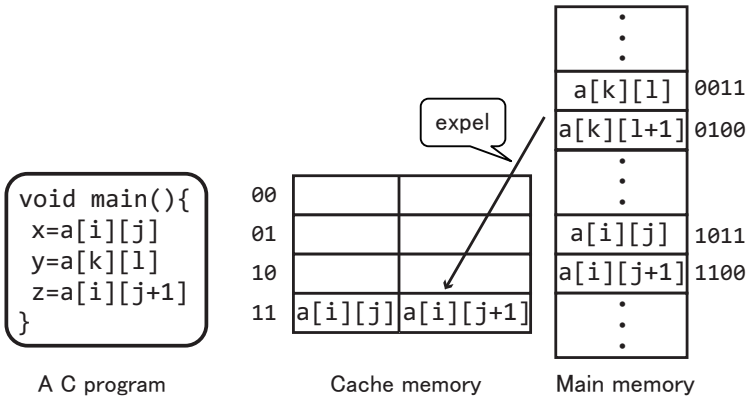


Figure 5.1: An example of a cache miss that is a target of MDGLIA.

that a multidimensional array represents an array of lower dimensional arrays, preferentially aggregating references with same indexes more in higher dimensions may further decrease cache misses.

Example.

Consider the array reference `a[i][j+1]` in Fig. 5.1. In this chapter, for ease of explanation, we assume that cache memory is directly mapped without loss of generality. That is, when the data are transferred from the main memory to the cache memory, the cache line is determined by the memory address modulo of the number of lines in the cache memory. The referenced data of `a[i][j+1]` is copied on the cache memory after the execution of array reference `a[i][j]`, but the execution of `a[k][l]` may expel it. Therefore, `a[i][j+1]` will get a cache miss. However, this cache miss can be prevented by moving `a[i][j+1]` immediately before `a[k][l]`.

End of Example.

We present a new cache optimization technique, MDGLIA, that continuously aggregates the array references with the same indexes more in higher dimensions. MDGLIA extended LCM to aggregate array references, without sacrificing the removal effects of redundant expressions. MDGLIA computes how many indexes of each array reference preceding a moved candidate are same as ones of the candidate, and then MDGLIA moves the candidate to the program points close to the references with the same indexes most in higher dimensions.

Example.

Consider array references in Fig. 5.2 (a). Aggregation is applied to each array reference traversing CFG in the *topological sort order*. First, MDGLIA moves the array reference `a[k][l]` immediately before the array reference `b[i]` because the execution of `b[i]` between `a[i][j]` and `a[k][l]` may

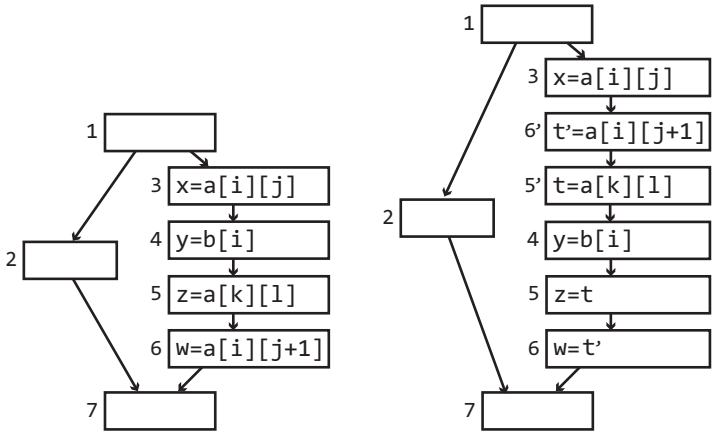


Figure 5.2: Effectiveness of MDGLIA. (a) Original code. (b) Result of applying MDGLIA.

cause the data of `a[k][l]` to be removed from the cache memory if the data of `b[i]` shares some cache lines for `a[k][l]`. Here, MDGLIA makes a new Node 5' for the moved array reference. Next, consider `a[i][j+1]` at Node 6. Although a sub-array `a[i]` is accessed at Nodes 3 and 6, another array `b` and sub-array `a[k]` are accessed between them. Hence, the data of `a[i][j+1]` may be removed from the cache memory after the execution of them. MDGLIA moves `a[i][j+1]` immediately before `a[k][l]` as shown in Fig. 5.2 (b).

End of Example.

The advantages of MDGLIA are summarized as follows:

- MDGLIA not only removes redundant array references but also decreases the number of cache misses, considering array dimensions.
- MDGLIA suppresses spills without decreasing the effect of preventing cache miss.

5.2 Related Work

5.2.1 Cache Optimization

Once a data in the memory is used, it tends to be used again in the near future, and the data stored around it tends to be used in much of the program. These two phenomena are called temporal locality and spatial

locality, respectively, which are utilized in order to improve the execution efficiency of the programs through cache memories. Popular techniques for enhancing the localities are due to transforming loop structures [1, 4, 44]. Although these techniques often greatly improve the execution efficiency of a program, its application tends to be limited to specific control structures such as a simple loop. However, MDGLIA is based on global code motion, which does not have to change the control structure of a program; therefore, MDGLIA can be applied to any programs.

There are some techniques that improve cache efficiency based on data layout. Cache-conscious data placement (CCDP) [14] reduces the cache conflict misses by considering the data layout. CCDP uses the *temporal relationship graph* (TRG). In the TRG, the nodes represent objects (e.g., functions, arrays, and global variables) to be placed into the data cache. The edges between the objects represent the estimated number of cache misses that would occur if the two objects mapped to the same cache set. A compiler assigns addresses to the objects based on the conflict cost metric calculated for the TRG so as to minimize the cache conflict misses. Although CCDP can reduce the cache conflict misses for a processor core with a single execution context, it loses much of its benefit in a multithreaded environment because inter-thread conflicts are not deterministic. Sarkar and Tullsen proposed a technique that extends CCDP to multithreaded architectures [71]. In their technique, the extra cost is generated so that threads share objects in cache blocks is added to each TRG edge as weight. Similar to CCDP, the compiler assigns addresses to the objects using a TRG so as to minimize the cache conflict misses cost. As another technique based on the object layout, Ishitobi et al. proposed a technique for suppressing the energy consumption of on-chip memory by considering memory allocation on a processor that has cache memory and scratchpad memory [47]. Their technique considers a cacheable region, a scratchpad region, and a non-cacheable regions so as to minimize the total energy consumption and the number of cache misses. These approaches deal with memory allocation methods for data but not the ordering method of the execution code.

5.2.2 Removing Redundant Expressions

The original PRE technique was proposed by Morel and Renvoise [57], which uses bi-directional data-flow analysis. This technique can eliminate some redundancies and move loop invariant expressions out of loops, but some redundant expressions are not removed because the technique does not insert expressions at non down-safe nodes. Dhamdhere extended this technique to insert expressions on edges [28], and Dhamdhere and Patil proposed another technique that removes redundancies based on uni-directional data-flow analysis [30]. Bodik et al. proposed the removal of all redundant expressions by copying certain parts of the program [5]. Kawahito et al. proposed

a technique that removes partially redundant load and store instructions by extended PRE [48]. These techniques remove redundant expressions, but decreasing the number of cache misses is out of their target.

As another technique based on code motion, there is a speculative code motion technique [55], which speculatively moves load instructions out of loops. This technique needs to recognize loop structures, whereas MDGLIA can be applied to an entire program without recognizing them, because MDGLIA is based on PRE.

5.3 Background

5.3.1 Program Representation

We assume that MDGLIA is applied to the intermediate representation converted from a source program, which is represented as a sequence of statements with at most an operator or a function, and to CFG each node of which represents a single statement rather than a basic block. The right-hand side of an assignment is called an expression. Some statements include load and store instructions with memory access, which is described as an array reference, e.g., $a[i][j]$ with address a and indexes i and j. The statement loading the data at memory location $a[i][j]$ into a virtual register x is expressed as assignment statement $x = a[i][j]$, which we call a *load statement*. Similarly, a statement storing the data in a virtual register x to a memory location $a[i][j]$ is expressed as $a[i][j] = x$, which we call a *store statement*.

We assume that the memory access solely appears in assignment. That is, a procedure call $f(a[i][j])$ is split into two statements such as $t = a[i][j]$ and $f(t)$. In addition, a store statement with a nested array reference $a[i][j] = a[b[i]][k]$ is split to three statements such as $t = b[i]$, $t_0 = a[t][k]$ and $a[i][j] = t_0$. Moreover, the load statement includes a temporary virtual register instead of original virtual register in the left-hand side. For example, the load statement $i = a[i][j]$ is split into two statements such as $t = a[i][j]$ and $i = t$ by introducing a temporary virtual register t. We assume that any arrays are laid out in row-major order on the memory such as arrays in C, which means that the leftmost index corresponds to the highest dimension of the array.

5.3.2 Lazy Code Motion

PRE tends to lengthen the live-ranges of variables carrying loaded values to their uses because it removes redundant expressions by inserting some expressions. LCM tries to address the problem by hoisting expressions as early as possible and delaying them as late as possible. The hoisting contributes to eliminating all removable expressions, and the delaying contributes to minimizing the live-ranges of variables.

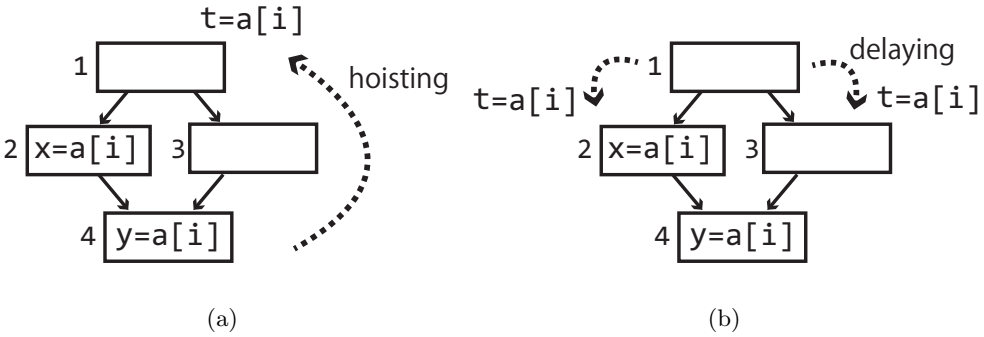


Figure 5.3: Code motions of LCM. (a) Hoisting expressions. (b) Delaying.

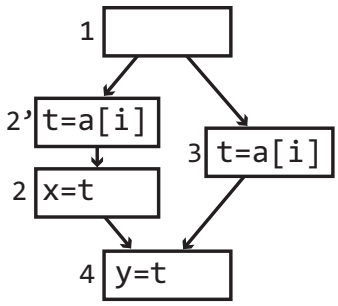


Figure 5.4: Result of applying LCM to Fig. 5.3 (a).

Example.

Consider an array reference `a[i]` at Node 4 in Fig. 5.3 (a). This array reference is partially redundant because it is redundant on a path through Node 2 whereas it is not on another path through Node 3. To remove this array reference, LCM determines CFG nodes at which array references can be inserted first. If it is inserted at Node 1, it is able to remove the redundant array reference by replacing the reference with the introduced temporary `t`. Notice here that, the live-range of `t` is lengthened; therefore, LCM delays the insertion node, as shown in Fig. 5.3 (b). Finally, LCM inserts the array references at Nodes 2 and 3, and then it replaces the original `a[i]` with `t`, as shown in Fig. 5.4.

End of Example.

These code motions have to satisfy two kinds of *safeties*: *down-safety* and *up-safety*. Down-safety is used to ensure that LCM does not introduce a new occurrence of the inserted expression on any execution path. Down-safety is represented by predicate *DownSafe*. In addition, up-safety is used to ensure

that there are some paths where the number of expressions is decreased by the insertions and removals. Up-safety is represented by predicate *UpSafe*. These safeties are defined under the condition of *transparency* that ensures that the value of an expression does not change at the program points of concern. LCM represents it by predicate *Transp*. Because the up-safety is one of the most important predicates of MDGLIA, we show the formal definition.

$$UpSafe(n) \overset{def}{\Leftrightarrow} (n \neq \textbf{start}) \wedge \prod_{p \in pred(n)} Comp(p) \vee (Transp(p) \wedge UpSafe(p)) \quad (5.1)$$

where the predicate $Comp(n)$ denotes that node n includes a same expression.

Example.

As shown in Fig. 5.3 (a), *DownSafe* of Nodes 1, 2, and 3 are *true* because of no modification for `i` nor `a[i]` at Nodes 2, 3, and 4. In contrast, *UpSafe* of all nodes are *false*. Because no occurrence of the array reference on a path from Node 1 to Node 3, $UpSafe(4)$ is *false*. Others nodes' *UpSafe* is obviously *false*.

End of Example.

LCM determines two kinds of insertion nodes based on the predicates *Earliest* and *Latest*, which determine insertion points in two code motions mentioned above, respectively. The predicate $Earliest(n)$ denotes that node n is the closest to **start** of the nodes m satisfying $DownSafe(m)$ or $UpSafe(m)$. The predicate $Latest(n)$ denotes that node n is the closest to the node c that satisfies $Comp(c)$ on each path from *Earliest* to **end**, and there is no node that satisfies the *Comp* on the path from *Earliest* to c. $Latest(n)$ is defined on the basis of maximal fixed points of the data-flow equation for the predicate $Delayed(n)$, which denotes that the expression can be delayed until the exit of node n.

Example.

Consider the `a[i]` in Fig. 5.3 (a). First, LCM decides that $Earliest(1)$ is *true* because $DownSafe(1)$ is *true*, and this node is the closest to the start node. Then, LCM delays the insertion points through the decisions of *Delayed* and *Latest*, which result in *true* for $Comp(2)$, *false* for $Delayed(4)$, and *true* for *Latest* at Nodes 2 and 3.

End of Example.

Here, as *Delayed* is also one of the most important predicates for MDGLIA, we present the formal definition.

$$Delayed(n) \overset{def}{\Leftrightarrow} Earliest(n) \vee (n \neq \textbf{start}) \wedge \prod_{p \in pred(n)} \neg Comp(p) \wedge Delayed(p) \quad (5.2)$$

LCM inserts expressions at the entry of nodes n satisfying the predicate $Insert(n)$, which denotes that n is one of nodes satisfying $Latest$. Note that LCM does not insert any expression without decreasing the number of expressions on some paths, because such insertions are unnecessary. Therefore, $Insert(n)$ is defined as $Latest(n) \wedge \neg Isolated(n)$ on the basis of the predicate $Isolated(n)$ which denotes that the insertion at n enables removing no expression other than original one.

5.4 Array Reference Aggregation

MDGLIA aggregates each array reference ar traversing CFG in the *topological sort order*. Similar to LCM, MDGLIA calculates $DownSafe$, $UpSafe$, $Earliest$, $Delayed$, and $Latest$ to determine nodes to move ar. In the delaying process, MDGLIA checks start addresses and the number of corresponding indexes of array references to aggregate them while considering the order of accesses to arrays.

5.4.1 Local Properties

For the checking addresses, MDGLIA defines local properties $SameAddr$, $Transp_e$, $Transp_{Addr}$, and $isSame$. $SameAddr$ and $isSame$ represent equalities of start addresses and whole expressions, respectively. $SameAddr(n)$ denotes that n contains a load statement referring to the same array reference as ar. $isSame$, corresponding to $Comp$ of LCM, denotes that $SameAddr$ is *true* and the indexes are same as ar. $Transp_e(n)$, corresponding to $Transp$ of LCM, denotes that there is no modification to ar and no store operation to the array referred by ar in n. $Transp_{Addr}$ denotes that there is neither modification to ar nor reference to arrays different from ar.

To determine these predicates, we use predicates rhs, $isLoad$, $TopAddr$, $Store$, Def, and Var. $rhs(n)$ returns the right-hand side of a statement at the node n. $isLoad(n)$ denotes that node n includes a load statement. $TopAddr(ar)$ returns the start address of ar if ar is an array; otherwise, it returns $\perp$. $Store(n)$ denotes that node n includes a store statement that accesses the same array as ar. $Def(n)$ gives a variable defined at node n. $Var(ar)$ gives a set of variables which are used in ar. The local properties are formally defined as follows:

$$
\begin{aligned}
SameAddr(n) &\overset{def}{\Leftrightarrow} isLoad(n) \ \wedge \ (TopAddr(rhs(n)) = TopAddr(ar)) \\
Transp_e(n) &\overset{def}{\Leftrightarrow} Def(n) \notin Var(ar) \wedge \neg Store(n) \\
Transp_{Addr}(n) &\overset{def}{\Leftrightarrow} Transp_e(n) \ \wedge \ (\neg isLoad(n) \vee SameAddr(n)) \\
isSame(n) &\overset{def}{\Leftrightarrow} rhs(n) \neq \bot \wedge rhs(n) = ar
\end{aligned}
$$

5.4.2 Modified Global Properties

MDGLIA uses modified *UpSafe* and *Delayed* of LCM.

Extending *UpSafe*

$UpSafe(n)$ of equation (5.1) is modified to denote that there are some array references whose start addresses are same as ar on all sub-paths leading to the node n. This predicate is formally defined as follows:

$$
\begin{aligned}
UpSafe(n) \overset{def}{\Leftrightarrow} \ & (n \neq \textbf{start}) \ \wedge \\
& \prod_{p \in pred(n)} SameAddr(p) \vee (Transp_e(p) \wedge UpSafe(p))
\end{aligned}
$$

The modification not only contributes to detecting the access order to arrays, but also gives the criterion for speculatively moving array references.

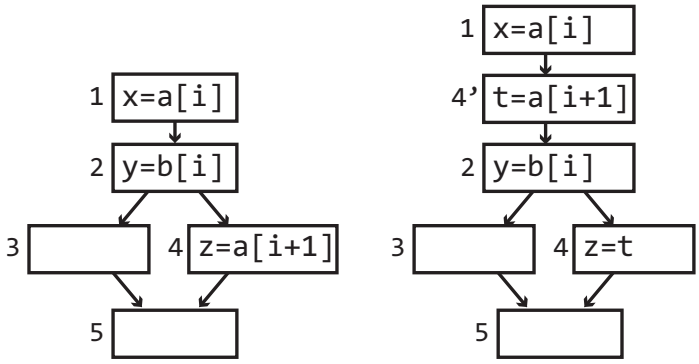
Example.

In Fig. 5.5 (a), consider applying MDGLIA to the `a[i+1]`. Because the array reference `a[i]` is executed at Node 1, $UpSafe(2)$ is *true*; therefore, $Earliest(2)$ is *true*. It leads $Delayed(2)$ to be *true*, but $Delayed(3)$ becomes *false* because $Transp_{Addr}(2)$ is *false*. As a result, $Latest(2)$ becomes *true*; therefore, `a[i+1]` is inserted before Node 2 as shown in 5.5 (b). Notice here that `a[i+1]` does not originally exist on the path through Node 3 in Fig. 5.5 (a). That is, this insertion is speculative. Speculative code motion may increase the number of the expression on some paths. However, if the speculatively inserted array references cause cache misses, the execution efficiency would be significantly decreased.

End of Example.

Extending *Delayed*

Delayed of equation (5.2) is modified to check whether *keep-order condition* and *keep-dimension condition* are satisfied.



(a) (b)

Figure 5.5: Speculative code motion. (a) Original code. (b) Moving array reference, not satisfying down-safety.

Keep-order Condition

The keep-order condition guarantees that there is no any reference to the array different from ar at n after preceding reference to the same array as ar. The keep-order condition is represented by $keepOrder(n)$, and defined as follows:

$$partialUpSafe(n) \overset{def}{\Leftrightarrow} \sum_{p \in pred(n)} UpSafe(p)$$

$$keepOrder(n) \overset{def}{\Leftrightarrow} \neg partialUpSafe(n) \vee Transp_{Addr}(n)$$

Example.

We show an example of the effect of aggregation considering keep-order condition in Fig. 5.6 (a). Consider moving the `a[k][l]` at Node 5. Nodes 3 and 5 contain array references that reference to array `a`; however there is an array reference that reference to array `b` between them. To move the `a[k][l]` immediately before Node 4, MDGLIA determines *Earliest* through checking *UpSafe* and *DownSafe* first. As there is no `a[k][l]` on the path through Node 2, *DownSafe*(1) and *UpSafe*(1) are *false*. These results cause *Earliest*(1) to be *false*. On the other hand, another path through Node 3 has the same reference at Node 5. Hence, *DownSafe*(3) and *Earliest*(3) are *true*, which causes *Delayed*(3) to be *true*. At each node from the Node 3, MDGLIA checks whether the keep-order condition is satisfied. Node 3 has an array reference that reference to the same array as `a[k][l]`; however,

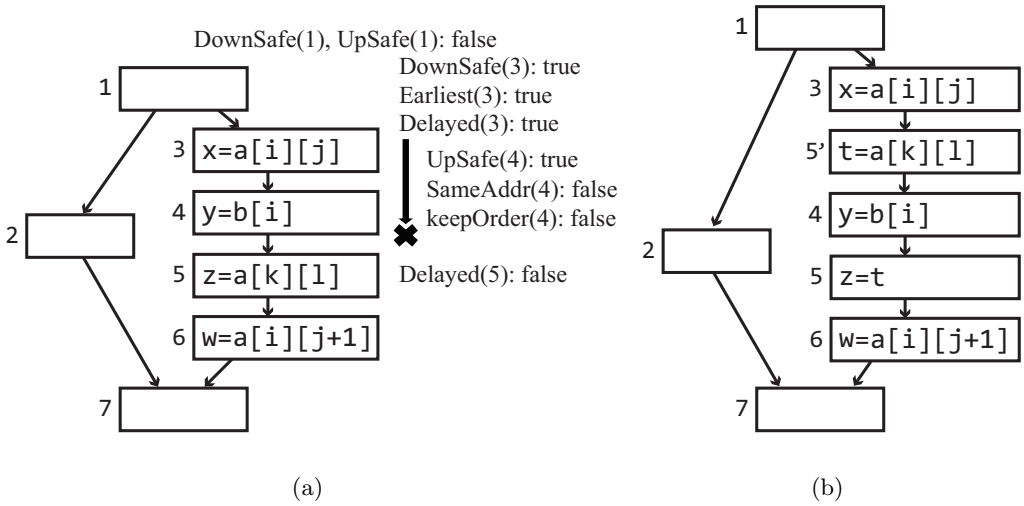


Figure 5.6: Effectiveness of extending *UpSafe*. (a) Result of computing data-flow equations to move `a[k][l]` at Node 5. (b) Result of moving the array reference.

Node 4 has another array reference that references to different array. Hence, *SameAddr*(3) is *true*, and then *UpSafe*(4) is *true*; however *SameAddr*(4) is *false*. These results cause *keepOrder*(4) and *Delayed*(5) to be *false*. Eventually, *Latest*(4) and *Insert*(4) are *true*, so that MDGLIA inserts an array reference immediately before the Node 4 and removes the original expression, as shown in Fig. 5.6 (b).

End of Example.

Keep-dimension Condition

The keep-dimension condition contributes to the aggregation of references to closer addresses in the same array through keeping the number of corresponding indexes from decreasing. To keep the number, this condition uses two kinds of predicates: number of corresponding indexes (nCI) and propagated nCI ($pnCI$).

nCI is the number of higher indexes corresponding with ar for representing closeness on the main memory between references to an array. $nCI(n)$ checks whether each index of the array reference at n is same as corresponding index of ar, one by one in left-first, whenever n contains a reference to the same array as ar. This predicate is formally defined as follows:

Definition 5.4.1. *We assume that ar is* $a[i_1][i_2]...[i_k]...[i_n]$*, and r is* $b[j_1][j_2]...[j_k]...[j_m]$*. Then,*

$$nCI(r) \overset{def}{\Leftrightarrow} \begin{cases} k & if\ a = b \wedge i_l = j_l \wedge \\ & \quad (k = n \vee k = m \vee i_{k+1} \neq j_{k+1}) \\ & \quad where\, \forall l \in \{1, ..., k\} \\ 0 & otherwise \end{cases}$$

The condition means that if their start addresses are same, these indexes are checked about whether the kth index is same or not until it is the last index.

$pnCI(n)$ denotes the number of indexes of the reference which has the most same indexes as ar on all execution paths to the exit of n from preceding node where *SameAddr* is *true*. *pnCI* can be defined by the data-flow equation based on *nCI* as follows:

$$pnCI(n) \overset{def}{\Leftrightarrow} \begin{cases} 0 & if\ n = \textbf{start} \\ nCI(n) & if\ SameAddr(n) \\ Max(\{pnCI(m) \mid m \in pred(n)\}) & otherwise \end{cases}$$

The equation can be iteratively solved as well as typical data-flow analysis. If node n includes a reference to the same start address as ar, then $pnCI(n)$ equals to $nCI(n)$. Otherwise, $pnCI(n)$ is the maximal value of the *pnCI* of all predecessors, except the start node.

The keep-dimension condition is represented by predicate *keepDimension* that is defined using *partialUpSafe* and *pnCI* as follows:

$$keepDimension(n) \overset{def}{\Leftrightarrow} \neg partialUpSafe(n) \vee \prod_{p \in pred(n)} pnCI(n) \geq pnCI(p)$$

This predicate denotes n does not include any reference with *nCI* less than the preceding references.

Delaying Considering Keep-order and Keep-dimension Conditions

MDGLIA checks whether the keep-order and keep-dimension conditions are satisfied in addition to *Delayed* of LCM. Once these conditions are introduced, *Delayed* is simply changed as follows:

$$\begin{aligned} Delayed(n) \overset{def}{\Leftrightarrow}\ & Earliest(n) \vee (n \neq \textbf{start})\ \wedge \\ & \prod_{p \in pred(n)} \neg isSame(p) \wedge keepOrder(p) \wedge \\ & \qquad keepDimension(p) \wedge Delayed(p) \end{aligned}$$

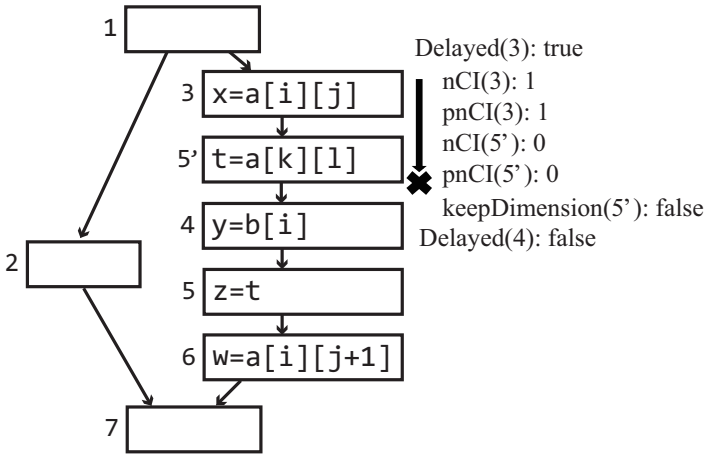


Figure 5.7: Computing closeness of each addresses of `a[i][l]` at Node 4 and *Delayed*.

Example.

In Fig. 5.7, we show results of data-flow analysis of MDGLIA when it moves the `a[i][j+1]` at Node 6 in Fig. 5.6 (b). Nodes 3 and 6 contain array references that reference to sub-array `a[i]`, but there is an array reference that reference to sub-array `a[k]` between them. To move `a[i][j+1]` immediately before Node 5', MDGLIA determines *Earliest* first. Because *Earliest*(3) is *true* as well as in the case where moving the `a[k][l]` in Fig. 5.6 (a), *Delayed*(3) is *true*. At each node following the Node 3, MDGLIA checks whether the keep-dimension condition is satisfied. Although *nCI* and *pnCI* of Node 3 are `1`, *nCI* and *pnCI* of the successor 5' are `0`; therefore, *keepDimension*($5'$) and *Delayed*(4) are *false*. Eventually, *Latest*($5'$) and *Insert*($5'$) are *true*; therefore, MDGLIA inserts an array reference immediately before the Node $5'$ and removes the original expression, as shown in Fig. 5.2 (b).

End of Example.

Finally, to help understanding of whole algorithm, we present formal definitions of all global predicates other than *UpSafe*, *keepOrder*, *keepDimension*, and *Delayed* as follows:

$$
\begin{aligned}
DownSafe(n) &\overset{def}{\Leftrightarrow} (n \neq \textbf{end}) \ \wedge \\
&\quad (isSame(n) \ \vee \ Transp_e(n) \ \wedge \prod_{s \in succ(n)} DownSafe(s)) \\
Safe(n) &\overset{def}{\Leftrightarrow} UpSafe(n) \vee DownSafe(n) \\
Earliest(n) &\overset{def}{\Leftrightarrow} Safe(n) \ \wedge \\
&\quad ((n = \textbf{start}) \ \vee \sum_{p \in pred(n)} \neg \, Transp_e(p) \vee \neg \, Safe(p)) \\
Latest(n) &\overset{def}{\Leftrightarrow} Delayed\,(n) \ \wedge \ (isSame(n) \ \vee \sum_{s \in succ(n)} \neg \, Delayed(s)) \\
Isolated(n) &\overset{def}{\Leftrightarrow} \prod_{s \in succ(n)} Latest(s) \vee \ \neg isSame(s) \wedge Isolated(s) \\
Insert(n) &\overset{def}{\Leftrightarrow} Latest(n) \ \wedge \ \neg Isolated(n) \\
Replace(n) &\overset{def}{\Leftrightarrow} isSame(n) \ \wedge \ \neg(Latest(n) \wedge Isolated(n))
\end{aligned}
$$

5.4.3 Application to the Entire Program

We design MDGLIA as a demand-driven analysis, which is applied to each array reference one by one, rather than exhaustive one. In general, PRE based approaches are known that an application of it exposes new redundant expressions. The effect is called *second-order effects*. Capturing them as many as possible results in removing more redundant expressions. However, capturing all the second-order effects requires iterative applications of PRE, which are costly because PRE is traditionally designed based on an exhaustive data-flow analysis [60, 78, 80]. On the other hand, the demand-driven applications of PRE to the entire program in the topological sort order enables efficiently capturing a lot of the second-order effects, as shown in Chapter 3. Furthermore, for the aggregating array references, demand-driven application contributes to suppressing unnecessary code motion.

Example.

In Fig. 5.8 (a), Nodes 2 and 3 might expel data that may be used later from a cache memory. At this time, if MDGLIA would move all array references in the exhaustive application such as the traditional PRE, references `a[i+1]` and `b[i+1]` would be moved as shown in Fig. 5.8 (b). As a result, even if the problems of cache misses are resolved, it would enhance register pressure of the temporary variable which had the value of `b[i+1]` because the unnecessary code motion of `b[i+1]` lengthened the live-range of it. By contrast, in demand-driven application, only the `a[i+1]` is moved to the

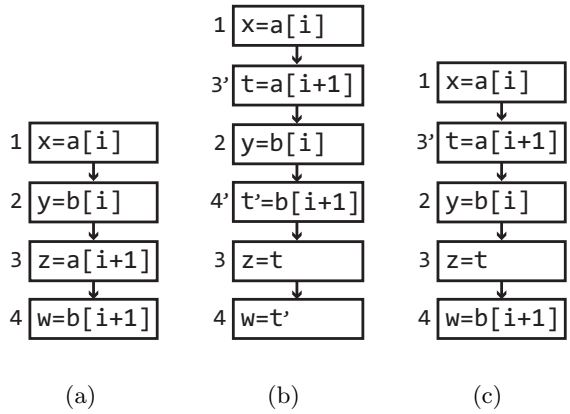


Figure 5.8: Preserving unnecessary code motion. (a) Original code. (b) Result of applying exhausive analysis version of MDGLIA to `a[i+1]`. (c) Result of applying demand-driven style MDGLIA to `a[i+1]`.

entry of Node 2 based on the demand-driven application manner as shown Fig. 5.8 (c).

End of Example.

5.5 Experimental Results

We have implemented MDGLIA as a low-level intermediate representation converter in a COINS compiler. To emphasize the benefits of MDGLIA, we compared MDGLIA with the following two optimizations.

LCM-MEM only removes redundant array references based on LCM.

GLIA aggregates references to the same array without considering their dimensions.

We evaluated the effects of these optimizations for two programs (equake and art) of CFP2000 and three programs (mcf, gzip, and twolf) of CINT2000 in the SPEC benchmarks on x86 machine whose CPU and OS are, respectively, Intel Core2Duo U9600 1.6GHz and CentOS. The system parameters of the machine are shown in Table 5.1.

To show how many cache misses can be decreased by GLIA and MDGLIA, we measured the following two hardware counters.

DCache_Repl: the number of replacement at L1 data cache.

L2_Lines_Out: the number of cache out from L2 cache.

Table 5.1: System parameters of cache memories

Parameters	L1D	L2
Total size (KB)	32	3,072
Line size (bytes)	64	64
The number of cache line	512	49,152
Associativity	8	12

We simply call these numbers L1D cache miss and L2 cache miss, respectively.

The result of comparison among the cache misses for the three optimizations is shown in Fig. 5.9, and the results of the number of L1D and L2 cache misses, respectively, are shown in Tables 5.2 and 5.3. Comparing GLIA and MDGLIA with LCM-MEM, L1D cache misses occur to the same degree for four programs (art, mcf, gzip, and twolf), though they increased about 7% for a program (ammp). Comparing GLIA with LCM-MEM, L2 cache misses decreased for three programs (art, mcf, and gzip). In particular, the cache misses remarkably decreased about 19.9% for art. In contrast, they increased for two programs (ammp and twolf). Comparing MDGLIA with LCM-MEM, L2 cache misses decreased for four programs (art, mcf, gzip, and ammp). In particular, the cache misses remarkably decreased about 30.3% in art. On the other hand, they increased for a program (twolf). Comparing MDGLIA with GLIA, L2 cache misses decreased for four programs (art, gzip, ammp, and twolf). In particular, the cache miss remarkably decreased about 15.6% in ammp. For ammp, comparing these techniques with

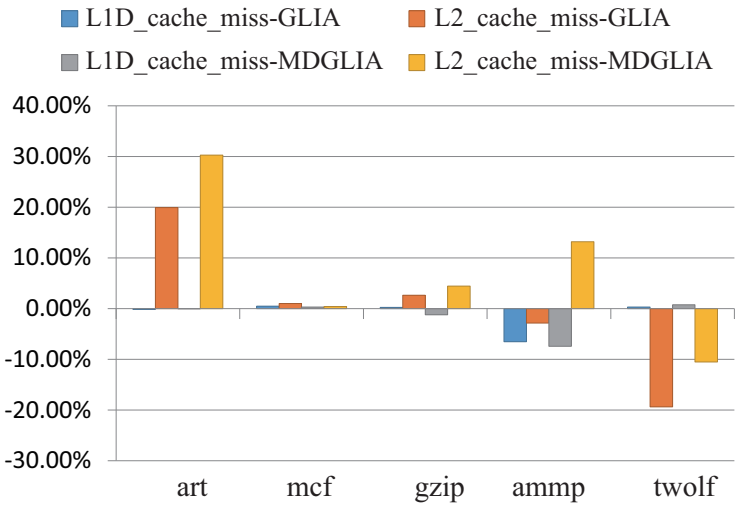


Figure 5.9: Decrease rate of cache misses.

Table 5.2: The number of DCache_Repl

Program	LCM-MEM	GLIA	MDGLIA
art	11,439,104,576	11,453,555,362	11,444,685,513
mcf	7,567,419,160	7,531,488,486	7,544,975,464
gzip	7,725,469,653	7,707,389,218	7,818,871,317
ammp	21,792,733,488	23,216,563,772	23,413,676,395
twolf	9,056,125,547	9,029,509,531	8,986,903,509

Table 5.3: The number of L2_Lines_Out

Program	LCM-MEM	GLIA	MDGLIA
art	366,823,834	293,702,375	329,018,854
mcf	727,422,092	720,133,268	724,349,339
gzip	25,856,048	25,177,224	24,706,624
ammp	408,120,338	419,706,166	354,268,633
twolf	1,518,718	1,812,932	1,678,713

LCM-MEM, although L2 cache misses increased for GLIA, they decreased for MDGLIA. This is because MDGLIA can aggregate more array references than GLIA, showing that the aggregation of the array references with the most similar indexes in higher dimensions is important.

In some programs, GLIA and MDGLIA caused cache misses to increase. The increase is considered to be derived from the following reasons: 1) the number of spills of temporary variables increased, and 2) speculative code motion lengthened execution paths without contributing to decrease of cache misses, as mentioned in Section 5.4.2.

The first reason can be considered to be for the use of LCM framework of these techniques, which is known to increase the number of spills [39] although LCM tries to decrease them by delaying insertion points. In addition, GLIA and MDGLIA tend to stop the delaying earlier than LCM.

Example.

Consider an array reference `a[i][j+1]` in Fig. 5.10 (a). The array reference is redundant at Node 6; therefore, it can be removed by LCM-MEM, as shown in Fig. 5.10 (b). On the other hand, this array reference is just moved in GLIA and MDGLIA, as shown in Fig. 5.10 (c) and Fig. 5.10 (d). This observations show that GLIA and MDGLIA may enhance more register pressure than LCM-MEM. If some spills occur, they may decrease the effects of GLIA and MDGLIA by the insertions of some store and load instructions between the array references to the same array. In other words, the spills may increase not only the length of some execution path but also the number of cache misses.

End of Example.

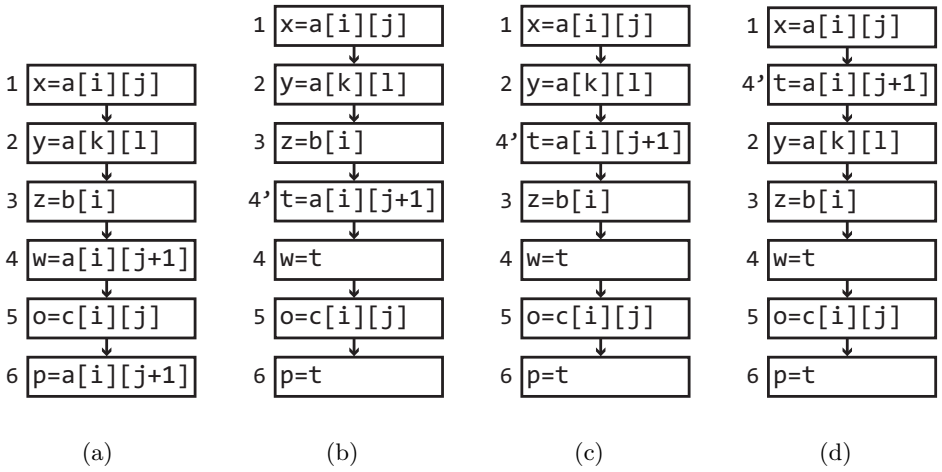


Figure 5.10: Difference of an insertion point for `a[i][j+1]`. (a) Original code. (b) Result of applying LCM-MEM. (c) Result of applying GLIA. (d) Result of applying MDGLIA.

We conducted experiments to confirm how the problems increased the number of cache misses.

Impact of Spill

We show the number of spills for the applications of three optimizations in Table 5.4. Comparing GLIA and MDGLIA with LCM-MEM, they caused more spills for four programs (mcf, gzip, ammp, and twolf). In particular, the number of the spills increased about 20% for twolf. However, in spite of the increase of the spills, L1D cache misses were held to the same degree as LCM-MEM for mcf and gzip. These additional results show that, the increase of the spills does not always increase the number of cache misses. Actually, applying MDGLIA to ammp decreased L2 cache misses although it increases the number of spills. In contrast, for twolf, the number of both spills and cache misses increased. It is considered that load/store for spills

Table 5.4: The number of register spills

Program	A.LCM-MEM	B.GLIA	C.MDGLIA	(A-B)/A	(A-C)/A	(B-C)/B
art	28	28	28	0.0 %	0.0 %	0.0 %
mcf	61	71	71	-16.4 %	-16.4 %	0.0 %
gzip	107	114	114	-6.5 %	-6.5 %	0.0 %
ammp	317	334	335	-5.4 %	-5.7 %	-0.3 %
twolf	804	962	979	-19.7 %	-21.8 %	-1.8 %

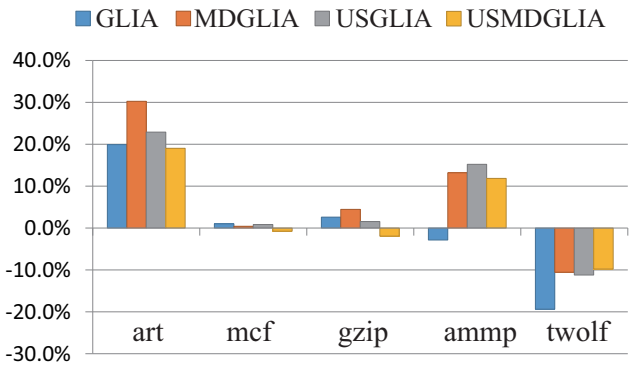


Figure 5.11: The decrease ratio of L2 cache miss for GLIA, MDGLIA, USGLIA, and USMDGLIA to the cache miss for LCM-MEM.

were inserted between references continuously moved by MDGLIA. Thus, a new register allocation technique for suppressing more cache misses is one of our future works.

Speculative Code Motion vs. Unspeculative Code Motion

In order to comparatively confirm how many number of cache misses the speculative code motion decreases, we implemented the following two optimizations that do not speculatively aggregate array references:

USGLIA aggregates array references with the keep-order condition based on *UpSafe* of equation (5.1).

USMDGLIA aggregates array references with the keep-order and keep-dimension conditions based on *UpSafe* of equation (5.1).

First, we show the result of L2 cache misses for USGLIA and USMDGLIA in addition to GLIA and MDGLIA in Fig. 5.11 and Table 5.5. Comparing GLIA with USGLIA, GLIA increased the number of cache misses for three programs (art, ammp, and twolf). In contrast, comparing MDGLIA with USGLIA, MDGLIA decreased the number of the cache misses for two programs (art and gzip). Comparing MDGLIA with USMDGLIA, MDGLIA decreased the number of the cache misses for four programs (art, mcf, gzip, and ammp). Thus, MDGLIA was better than these unspeculative versions on average. This is because speculative code motion can aggregate more array references than unspeculative ones. Table 5.6 shows the number of stopped delaying by checking keep-order or keep-dimension conditions in the speculative and unspeculative code motions. As shown in the table, the

Table 5.5: The number of L2 cache miss of USGLIA and USMDGLIA, and the cache miss ratio of them to the cache miss for GLIA and MDGLIA

Program	USGLIA	vs. GLIA	vs. MDGLIA	USMDGLIA	vs. MDGLIA
art	282,855,332	3.7 %	-10.6 %	297,012,498	-16.1 %
mcf	721,268,586	-0.2 %	0.4 %	732,708,340	-1.2 %
gzip	25,449,483	-1.1 %	-3.0 %	26,351,879	-6.7 %
ammp	345,935,455	17.6 %	2.4 %	359,701,112	-1.5 %
twolf	1,689,141	6.8 %	-0.6 %	1,667,434	0.7 %

Table 5.6: The number of aggregated array references under *keepOrder* and *keepDimension* in speculative/not speculative code motion

	Speculative		Not Speculative	
Program	*keepOrder*	*keepDimension*	*keepOrder*	*keepDimension*
art	438	22	145	22
mcf	750	298	104	71
gzip	829	134	182	77
ammp	8,043	2,015	719	512
twolf	30,399	7,313	8,855	811

speculative code motion was able to aggregate move more array references than unspeculative one.

Considering L2 cache miss when GLIA, MDGLIA, USGLIA, or USMDGLIA was applied to twolf, they increased L2 cache misses for the program. Although the increase for GLIA was about 20%, the other increases were about 10%. Furthermore, comparing USGLIA and USMDGLIA, the results of L2 cache misses are held to the same degree. These results indicate that GLIA speculatively inserts array references, which cause the number of spills to increase, leading to cache misses. On the other hand, speculative aggregation of array references with the most similar indexes in higher dimensions decreases the cache misses although MDGLIA increases more number of spills than GLIA. That is, the speculative aggregation under keep-dimension condition decreased the L2 cache misses.

Finally, these results shows that the aggregation of array references is more useful technique for decreasing cache misses than simply removing redundant array references. However, unspeculative code motion is better than speculative ones for some programs. Because the speculative code motion can move array references further than unspeculative one has potential, we believe that MDGLIA can be improved by profiling whether the aggregated array references get cache hit.

5.6 Summary

In this chapter, we have proposed a new global code motion technique for aggregating array references with the most similar indexes in higher dimensions of the same array in order to suppress cache misses. MDGLIA not only suppresses the cache misses but also suppresses spills through delaying the load instructions without changing their access order.

In order to show the effectiveness of MDGLIA, we applied it to some benchmark programs. As a result, we have shown that there are some programs in which it decreases the number of cache misses. As future work, we can consider 1) making a new register allocation that preferentially spills variables that will get cache hits, and 2) aggregating array references based on sophisticated informations of cache memory and their addresses on main memory.

Chapter 6

Conclusion and Future Direction

The optimization of memory hierarchy utilization is one of the most important techniques of code optimizations because accesses to the main memory remarkably decreases the efficiency of execution of objective code. To solve this issue, many researchers have proposed a lot of kinds of hardware structures and various code optimization techniques, such as register promotion, register allocation, and loop transformation. However, the penalty of access to the main memory tends to increase; therefore, the importance of the issue will increase further.

6.1 Summary of Contributions

This thesis presents a new code motion-based memory hierarchy utilization optimization framework. In particular, partial redundancy elimination (PRE) plays a fundamental role in the framework. PRE removes redundant array references by inserting expressions to suitable points in a program and replacing them with the introduced temporary variables. The framework extends the potential of PRE to remove array references that are redundant over iterations and suppress cache misses from occurring.

The main contributions of the framework are summarized as follows:

- **Removing redundant array references by efficient demand-driven analysis without sacrificing traditional PRE's power**

  The contribution is derived from the effective demand-driven PRE (EDDPRE). EDDPRE removes redundant array reference in about half cost of the traditional PRE in combination with global value numbering (GVN) and query propagation. When removing all redundant array references including second-order effects with the traditional PRE, because it detects redundant expressions based on their

lexical equality, iterative applications of copy propagation and PRE is required for removing lexically different expressions with the same value. On the other hand, GVN can reveal the redundant expressions even if they are lexically different. In addition, the query propagation can shorten range to be analyzed.

- **Decreasing the number of memory references by increasing the number of register references through efficient query propagation**.

  EDDPRE replaces redundant array references with a temporary variable holding value of inserted expressions. In addition, we extended EDDPRE to handle array references that are redundant across several iterations, such as scalar replacement. We named this extended EDDPRE PRE-based scalar replacement (PRESR). PRESR inserts compensation code for maintaining the behavior of a program at a less frequently executed point, and then it removes the redundant array reference. Array references accesses to cache memory whereas using temporary variables gets the data from register; therefore, EDDPRE and PRESR reduce the number of memory references. We showed that PRESR can improve the efficiency of execution of some benchmark programs. PRESR improved the execution efficiency of SPEC 2000 benchmark programs about 2% on average.

- **Decreasing the number of cache misses based on PRE framework**.

  We proposed two new cache optimization techniques, global load instruction aggregation (GLIA) and multidimensional GLIA (MDGLIA). GLIA aggregates array references for making accesses to the same array continuous because loading data at a specific array index from main memory places it in the cache memory along with other data belonging to the same array. Therefore, some array references can be executed before the reference data are removed from cache memory by the aggregation. To achieve aggregation, we extended PRE to move the references immediately after other preceding references to the same array and then delaying it immediately before another reference to a different array.

  We showed that GLIA basically works well. Comparing removal of redundant array references, the last level (L2) cache misses decreased remarkably about 19.9% in the best case. However, the number of the cache misses increased for some programs.

  In order to enhance effect of aggregation, MDGLIA extended GLIA to manage the array dimensions because a multidimensional array represents an array of lower dimensional arrays, preferentially aggregating

references with same indexes in higher dimensions may further decrease cache misses. MDGLIA computes the number of same indexes as a moved candidate each preceding array reference has, and then MDGLIA moves the candidate to the program points that are the closest to the reference with the most similar indexes in higher dimensions.

As a result, MDGLIA could decrease L2 cache misses greater than GLIA. Comparing MDGLIA with GLIA, L2 cache misses decreased for several programs. In particular, the cache misses remarkably decreased about 15.6% in the best case.

6.2 Future Direction

This section proposes the future directions of the framework. We proposed only code motion-based techniques; however, using others optimization may enhance our techniques. We believe following new techniques will obtain remarkable gains from this framework, and will be demanded.

Using Sophisticated Alias Analysis

As our framework moves load statements, these techniques must consider an *alias* that points to a memory location referred by another pointers for safe movement. Pointer analysis reveals a memory location for each pointer reference [45]; therefore, whether two pointer references refer to a same memory location or not becomes obvious [86].

To improve the precision of pointer analysis, many researchers have proposed several algorithms. These techniques can be roughly classified into *flow-sensitivity* [31, 42, 43, 58, 75, 85] or *flow-insensitivity* [41]. Flow-sensitive pointer analysis calculates the pointer information at each program point along control flows, whereas flow-insensitive pointer analysis ignores execution order. Furthermore, pointer analysis techniques can be divided into two classes *context-sensitivity* [73, 82, 88] or *context-insensitivity*. Context-sensitivity considers the values of the arguments of function calls.

In this thesis, we used a very simple alias analysis. Using the state of the art alias analysis or pointer analysis may reveal the potential of our framework.

Profile Guided MDGLIA

Although MDGLIA suppressed cache misses, it could perform more effective code motion if it can use detailed information such as the memory addresses of array data. The reason is that main memory data are generally placed on a cache memory based on the number of cache line and associativity. This

information will enable predicting whether each array reference will result in cache misses.

Cache Miss Sensitive Register Allocation

Although PRE is a powerful code optimization technique, the insertion and removal processes move some expression to nodes of the CFG closer to the start node; therefore, the register pressure tends to increase [17, 63, 83]. If a spill occurs, some store and load instructions are inserted at some program points. The extra execution cost derived from these additional instructions may be greater than the improvement due to the redundancy elimination. Because our framework is extended from PRE, our framework also tends to increase register pressure. To suppress this cost, register allocation should preferentially spill variables for which load/store instructions inserted by spilling result in cache hits. This cache miss sensitive register allocation may increase the impact of PRE and our framework.

Bibliography

[1] Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers: principles, techniques, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986).

[2] Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D.: *Compilers: Principles, Techniques, and Tools (2nd Edition)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006).

[3] Alpern, B., Wegman, M. N. and Zadeck, F. K.: Detecting equality of variables in programs, *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, New York, NY, USA, ACM, pp. 1–11 (1988).

[4] Appel, A. W.: *Modern Compiler Implementation in ML: Basic Techniques*, Cambridge University Press, New York, NY, USA (1997).

[5] Bodik, R., Gupta, R. and Soffa, M. L.: Complete removal of redundant expressions, *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, New York, NY, USA, ACM, pp. 1–14 (1998).

[6] Bodik, R. and Gupta, R.: Partial Dead Code Elimination Using Slicing Transformations, *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, New York, NY, USA, ACM, pp. 159–170 (1997).

[7] Bodik, R., Gupta, R. and Soffa, M. L.: Load-reuse analysis: design and evaluation, *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, New York, NY, USA, ACM, pp. 64–76 (1999).

[8] Boissinot, B., Darte, A., Rastello, F., de Dinechin, B. D. and Guillon, C.: Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency, *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, Washington, DC, USA, IEEE Computer Society, pp. 114–125 (2009).

[9] Braun, M., Buchwald, S., Hack, S., Leiba, R., Mallon, C. and Zwinkau, A.: Simple and efficient construction of static single assignment form, *Proceedings of the 22nd international conference on Compiler Construction*, CC'13, Berlin, Heidelberg, Springer-Verlag, pp. 102–122 (2013).

[10] Briggs, P. and Cooper, K. D.: Effective Partial Redundancy Elimination, *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, New York, NY, USA, ACM, pp. 159–170 (1994).

[11] Briggs, P., Cooper, K. D., Harvey, T. J. and Simpson, L. T.: Practical Improvements to the Construction and Destruction of Static Single Assignment Form, *Softw. Pract. Exper.*, Vol. 28, No. 8, pp. 859–881 (1998).

[12] Briggs, P., Cooper, K. D. and Torczon, L.: Improvements to graph coloring register allocation, *ACM Trans. Program. Lang. Syst.*, Vol. 16, No. 3, pp. 428–455 (1994).

[13] Cai, Q. and Xue, J.: Optimal and efficient speculation-based partial redundancy elimination, *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, Washington, DC, USA, IEEE Computer Society, pp. 91–102 (2003).

[14] Calder, B., Krintz, C., John, S. and Austin, T.: Cache-conscious data placement, *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VIII, New York, NY, USA, ACM, pp. 139–149 (1998).

[15] Callahan, D., Carr, S. and Kennedy, K.: Improving register allocation for subscripted variables, *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, New York, NY, USA, ACM, pp. 53–65 (1990).

[16] Carr, S. and Kennedy, K.: Scalar replacement in the presence of conditional control flow, *Softw. Pract. Exper.*, Vol. 24, No. 1, pp. 51–77 (1994).

[17] Chaitin, G. J.: Register allocation & spilling via graph coloring, *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, SIGPLAN '82, New York, NY, USA, ACM, pp. 98–105 (1982).

[18] Choi, J.-D., Cytron, R. and Ferrante, J.: Automatic Construction of Sparse Data Flow Evaluation Graphs, *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, New York, NY, USA, ACM, pp. 55–66 (1991).

[19] Chow, F., Chan, S., Kennedy, R., Liu, S.-M., Lo, R. and Tu, P.: A New Algorithm for Partial Redundancy Elimination Based on SSA Form, *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, New York, NY, USA, ACM, pp. 273–286 (1997).

[20] Chow, F. and Hennessy, J.: Register allocation by priority-based coloring, *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, SIGPLAN '84, New York, NY, USA, ACM, pp. 222–232 (1984).

[21] Click, C.: Global code motion/global value numbering, *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, New York, NY, USA, ACM, pp. 246–257 (1995).

[22] Cocke, J.: Global Common Subexpression Elimination, *Proceedings of a Symposium on Compiler Optimization*, New York, NY, USA, ACM, pp. 20–24 (1970).

[23] Cocke, J. and Schwarts, J. T.: *Programming Languages and Their Compilers: Preliminary Notes*, Courant Institute of Mathematical Sciences, New York University (1970).

[24] COINS: http://coins-compiler.sourceforge.jp/.

[25] Cooper D., K., Harvey, T. J. and Kennedy, K.: Iterative data-flow analysis, revisited, Technical report (2004).

[26] Cooper D., K. and Xu, L.: An efficient static analysis algorithm to detect redundant memory operations, *Proceedings of the 2002 workshop on Memory system performance*, MSP '02, New York, NY, USA, ACM, pp. 97–107 (2002).

[27] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N. and Zadeck, F. K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, Technical report, Providence, RI, USA (1991).

[28] Dhamdhere, D. M.: A fast algorithm for code movement optimisation, *SIGPLAN Not.*, Vol. 23, No. 10, pp. 172–180 (1988).

[29] Dhamdhere, D. M.: E-path_ PRE:partial redundancy elimination made easy, *SIGPLAN Not.*, Vol. 37, No. 8, pp. 53–65 (2002).

[30] Dhamdhere, D. M. and Patil, H.: An elimination algorithm for bidirectional data flow problems using edge placement, *ACM Trans. Program. Lang. Syst.*, Vol. 15, No. 2, pp. 312–336 (1993).

[31] Dillig, I., Dillig, T. and Aiken, A.: Sound, Complete and Scalable Path-sensitive Analysis, *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, New York, NY, USA, ACM, pp. 270–280 (2008).

[32] Fernandez, M. and Espasa, R.: Link-Time Path-Sensitive Memory Redundancy Elimination, *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, Washington, DC, USA, IEEE Computer Society, pp. 300– (2004).

[33] Fink, S. J., Knobe, K. and Sarkar, V.: Unified Analysis of Array and Object References in Strongly Typed Languages, *Proceedings of the 7th International Symposium on Static Analysis*, SAS '00, London, UK, UK, Springer-Verlag, pp. 155–174 (2000).

[34] Gal, A., Probst, C. W. and Franz, M.: HotpathVM: An Effective JIT Compiler for Resource-constrained Devices, *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, New York, NY, USA, ACM, pp. 144–153 (2006).

[35] Gargi, K.: A Sparse Algorithm for Predicated Global Value Numbering, *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, New York, NY, USA, ACM, pp. 45–56 (2002).

[36] George, L. and Appel, A. W.: Iterated register coalescing, *ACM Trans. Program. Lang. Syst.*, Vol. 18, No. 3, pp. 300–324 (1996).

[37] Gulwani, S. and Necula, G. C.: Global Value Numbering Using Random Interpretation, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, New York, NY, USA, ACM, pp. 342–352 (2004).

[38] Gupta, R., Berson, D. A. and Fang, J. Z.: Path Profile Guided Partial Redundancy Elimination Using Speculation, *Proceedings of the 1998 International Conference on Computer Languages*, ICCL '98, Washington, DC, USA, IEEE Computer Society, pp. 230–239 (1998).

[39] Gupta, R. and Bodik, R.: Register Pressure Sensitive Redundancy Elimination, *Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, CC '99, London, UK, UK, Springer-Verlag, pp. 107–121 (1999).

[40] Hailperin, M.: Cost-optimal Code Motion, *ACM Trans. Program. Lang. Syst.*, Vol. 20, No. 6, pp. 1297–1322 (1998).

[41] Hardekopf, B. and Lin, C.: The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code, *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, New York, NY, USA, ACM, pp. 290–299 (2007).

[42] Hardekopf, B. and Lin, C.: Semi-sparse flow-sensitive pointer analysis, *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, New York, NY, USA, ACM, pp. 226–238 (2009).

[43] Hardekopf, B. and Lin, C.: Flow-sensitive Pointer Analysis for Millions of Lines of Code, *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, Washington, DC, USA, IEEE Computer Society, pp. 289–298 (2011).

[44] Hennessy, J. L. and Patterson, D. A.: *Computer Architecture, Fifth Edition: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2011).

[45] Hind, M. and Pioli, A.: Which Pointer Analysis Should I Use?, *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, New York, NY, USA, ACM, pp. 113–123 (2000).

[46] Horspool, R. N. and Ho, H. C.: Partial Redundancy Elimination Driven by a Cost-Benefit Analysis, *Proceedings of the 8th Israeli Conference on Computer-Based Systems and Software Engineering*, ICCSSE '97, Washington, DC, USA, IEEE Computer Society, pp. 111– (1997).

[47] Ishitobi, Y., Ishihara, T. and Yasuura, H.: Code and Data Placement for Embedded Processors with Scratchpad and Cache Memories, *J. Signal Process. Syst.*, Vol. 60, No. 2, pp. 211–224 (2010).

[48] Kawahito, M., Komatsu, H. and Nakatani, T.: Partial redundancy elimination for access expressions by speculative code motion, *Softw. Pract. Exper.*, Vol. 34, No. 11, pp. 1065–1090 (2004).

[49] Kennedy, R., Chan, S., Liu, S.-M., Lo, R., Tu, P. and Chow, F.: Partial redundancy elimination in SSA form, *ACM Trans. Program. Lang. Syst.*, Vol. 21, No. 3, pp. 627–676 (1999).

[50] Kildall, G. A.: A Unified Approach to Global Program Optimization, *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, New York, NY, USA, ACM, pp. 194–206 (1973).

[51] Knoop, J., Ruthing, O. and Steffen, B.: Lazy code motion, *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, New York, NY, USA, ACM, pp. 224–234 (1992).

[52] Knoop, J., Ruthing, O. and Steffen, B.: Optimal code motion: theory and practice, *ACM Trans. Program. Lang. Syst.*, Vol. 16, No. 4, pp. 1117–1155 (1994).

[53] Knoop, J., Ruthing, O. and Steffen, B.: Partial dead code elimination, *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, New York, NY, USA, ACM, pp. 147–158 (1994).

[54] Lin, J., Chen, T., Hsu, W.-C., Yew, P.-C., Ju, R. D.-C., Ngai, T.-F. and Chan, S.: A Compiler Framework for Speculative Analysis and Optimizations, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, New York, NY, USA, ACM, pp. 289–299 (2003).

[55] Lo, R., Chow, F., Kennedy, R., Liu, S.-M. and Tu, P.: Register promotion by sparse partial redundancy elimination of loads and stores, *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, New York, NY, USA, ACM, pp. 26–37 (1998).

[56] Lu, J. and Cooper, K. D.: Register promotion in C programs, *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, New York, NY, USA, ACM, pp. 308–319 (1997).

[57] Morel, E. and Renvoise, C.: Global optimization by suppression of partial redundancies, *Commun. ACM*, Vol. 22, No. 2, pp. 96–103 (1979).

[58] Nasre, R.: Time- and Space-efficient Flow-sensitive Points-to Analysis, *ACM Trans. Archit. Code Optim.*, Vol. 10, No. 4, pp. 39:1–39:27 (2013).

[59] Nie, J. T. and Cheng, X.: An efficient SSA-based algorithm for complete global value numbering, *Proceedings of the 5th Asian conference on Programming languages and systems*, APLAS'07, Berlin, Heidelberg, Springer-Verlag, pp. 319–334 (2007).

[60] Odaira, R. and Hiraki, K.: Partial Value Number Redundancy Elimination, *Information Processing Society of Japan Transactions on Programming*, Vol. 45, No. SIG09(PRO22), pp. 59–79 (2004). (in Japanese).

[61] Odaira, R., Nakaike, T., Inagaki, T., Komatsu, H. and Nakatani, T.: Coloring-based coalescing for graph coloring register allocation, *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, New York, NY, USA, ACM, pp. 160–169 (2010).

[62] PAPI: http://icl.cs.utk.edu/papi/.

[63] Poletto, M. and Sarkar, V.: Linear scan register allocation, *ACM Trans. Program. Lang. Syst.*, Vol. 21, No. 5, pp. 895–913 (1999).

[64] Rastello, F., Ferriere, F. d. and Guillon, C.: Optimizing Translation Out of SSA Using Renaming Constraints, *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, Washington, DC, USA, IEEE Computer Society, pp. 265– (2004).

[65] Reif, J. H. and Lewis, H. R.: Symbolic Evaluation and the Global Value Graph, *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, New York, NY, USA, ACM, pp. 104–118 (1977).

[66] Rishi, S., Rajkishore, B., Jisheng, Z. and Vivek, S.: Inter-iteration Scalar Replacement Using Array SSA Form, *Proceedings of the 23rd international conference on Compiler Construction*, CC'14, Berlin, Heidelberg, Springer-Verlag, pp. 40–60 (2014).

[67] Rosen, B. K., Wegman, M. N. and Zadeck, F. K.: Global value numbers and redundant computations, *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, New York, NY, USA, ACM, pp. 12–27 (1988).

[68] Ruthing, O.: Optimal Code Motion in the Presence of Large Expressions, *Proceedings of the 1998 International Conference on Computer Languages*, ICCL '98, Washington, DC, USA, IEEE Computer Society, pp. 216–225 (1998).

[69] Ruthing, O., Knoop, J. and Steffen, B.: Detecting Equalities of Variables: Combining Efficiency with Precision, *Proceedings of the 6th International Symposium on Static Analysis*, SAS '99, London, UK, UK, Springer-Verlag, pp. 232–247 (1999).

[70] Ruthing, O., Knoop, J. and Steffen, B.: Sparse code motion, *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, New York, NY, USA, ACM, pp. 170–183 (2000).

[71] Sarkar, S. and Tullsen, D. M.: Compiler techniques for reducing data cache miss rate on a multithreaded architecture, *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, HiPEAC'08, Berlin, Heidelberg, Springer-Verlag, pp. 353–368 (2008).

[72] Scholz, B., Horspool, N. and Knoop, J.: Optimizing for Space and Time Usage with Speculative Partial Redundancy Elimination, *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '04, New York, NY, USA, ACM, pp. 221–230 (2004).

[73] Shang, L., Xie, X. and Xue, J.: On-demand Dynamic Summary-based Points-to Analysis, *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, New York, NY, USA, ACM, pp. 264–274 (2012).

[74] Sreedhar, V. C., Ju, R. D.-C., Gillies, D. M. and Santhanam, V.: Translating Out of Static Single Assignment Form, *Proceedings of the 6th International Symposium on Static Analysis*, SAS '99, London, UK, UK, Springer-Verlag, pp. 194–210 (1999).

[75] Sui, Y., Ye, S., Xue, J. and Yew, P.-C.: SPAS: Scalable Path-sensitive Pointer Analysis on Full-sparse SSA, *Proceedings of the 9th Asian Conference on Programming Languages and Systems*, APLAS'11, Berlin, Heidelberg, Springer-Verlag, pp. 155–171 (2011).

[76] Sumikawa, Y., Ojima, R. and Takimoto, M.: Demand-driven Scalar Replacement, *Computer Software* (2014). (in Japanese), to appear.

[77] Sumikawa, Y. and Takimoto, M.: Global Load Instruction Aggregation Based on Code Motion, *Proceedings of the 2012 Fifth International Symposium on Parallel Architectures, Algorithms and Programming*, PAAP '12, Taipei, IEEE Computer Society, pp. 149–156 (2012).

[78] Sumikawa, Y. and Takimoto, M.: Effective Demand-driven Partial Redundancy Elimination, *Information Processing Society of Japan Transactions on Programming*, Vol. 6, No. 2, pp. 33–44 (2013).

[79] Sumikawa, Y. and Takimoto, M.: Global Load Instruction Aggregation Based on Array Dimensions, *Proceedings of the 2014 Sixh International Symposium on Parallel Architectures, Algorithms and Programming*, PAAP '14, Washington, DC, USA, IEEE Computer Society, pp. 123–129 (2014).

[80] Takimoto, M.: Speculative Partial Redundancy Elimination Based on Question Propagation, *Information Processing Society of Japan Transactions on Programming*, Vol. 2, No. 5, pp. 15–27 (2009). (in Japanese).

[81] VanDrunen, T. and Hosking, A. L.: Value-based partial redundancy elimination, *In CC*, pp. 167–184 (2004).

[82] Whaley, J. and Lam, M. S.: Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, New York, NY, USA, ACM, pp. 131–144 (2004).

[83] Wimmer, C. and Franz, M.: Linear scan register allocation on SSA form, *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, New York, NY, USA, ACM, pp. 170–179 (2010).

[84] Xue, J. and Cai, Q.: A Lifetime Optimal Algorithm for Speculative PRE, *ACM Trans. Archit. Code Optim.*, Vol. 3, No. 2, pp. 115–155 (2006).

[85] Yu, H., Xue, J., Huo, W., Feng, X. and Zhang, Z.: Level by Level: Making Flow- and Context-sensitive Pointer Analysis Scalable for Millions of Lines of Code, *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, New York, NY, USA, ACM, pp. 218–229 (2010).

[86] Zheng, X. and Rugina, R.: Demand-driven alias analysis for C, *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, New York, NY, USA, ACM, pp. 197–208 (2008).

[87] Zhou, H., Chen, W. and Chow, F.: An SSA-based algorithm for optimal speculative code motion under an execution profile, *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, New York, NY, USA, ACM, pp. 98–108 (2011).

[88] Zhu, J. and Calman, S.: Symbolic Pointer Analysis Revisited, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, New York, NY, USA, ACM, pp. 145–157 (2004).