國立臺灣大學電機資訊學院資訊工程學研究所

碩士論文

Graduate Institute of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

低階虛擬機器的全域冗餘儲存指令消去

Global Dead Store Elimination in LLVM

林以倫

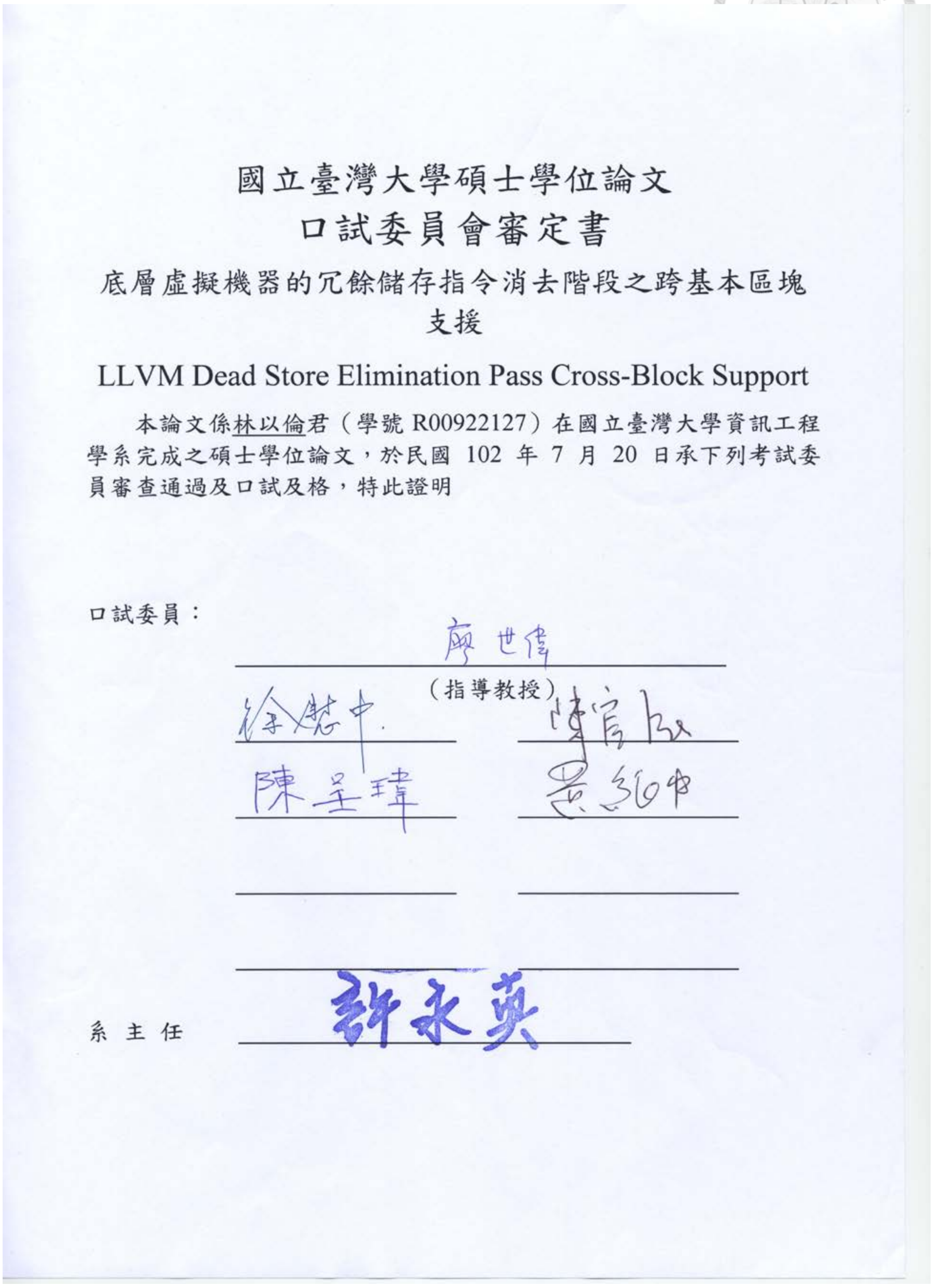
Yi-Lun Lin

指導教授：廖世偉 博士

Advisor: Shih-Wei Liao, Ph.D.

中華民國 102 年 7 月

July, 2013



國立臺灣大學碩士學位論文
口試委員會審定書

底層虛擬機器的冗餘儲存指令消去階段之跨基本區塊支援

LLVM Dead Store Elimination Pass Cross-Block Support

本論文係林以倫君（學號 R00922127）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 102 年 7 月 20 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

廖世偉
（指導教授）

徐慰中　　[illegible]

陳呈瑋　　[illegible]

系 主 任　許永英

誌謝

感謝指導教授廖世偉在學術研究上的大力幫助，以及實驗室強者學弟 TDY 及 Logan 的指教，還有各位實驗室子杰，老二，浩呆，FKG，安格斯，致遠，同學在研究上的陪伴。

中文摘要

我們優化了 LLVM 的 Dead Store Elimination，使其具有處理 global dead store instruction 的能力。增加三個功能，一是處理 Load-Store Redundancy，二是 Write-Write Redundancy，第三是 Write to Local Stack Object，這其中需要 Alias Analysis 的分析，以及新的演算法加入。

ABSTRACT

We optimize the Dead Store Elimination Pass in LLVM for handling global dead store instructions. Global Dead Store Elimination can handle Load-Store Redundancy, Write-Write Redundancy and Write to Local Stack Objects globally. It needs alias analysis and a new algorithim(revised DFS) is introduced.

CONTENTS

LIST OF FIGURES

Chapter 1 Introduction

The last decade has seen a proliferation of managed languages. This is motivated by the pressing needs for higher-level, more flexible, garbage-collecting languages such as Java and Python. Runtime has become essential. The challenge of building high quality language runtimes has increased significantly since the late 1990's when the Java and C# runtimes emerged. Thus, the need to update runtime data structure has skyrocketed. As a result, the store operations become more and more frequent than in a traditional native environment.

Although LLVM claims to support both native (C and C++) and managed (Java and JavaScript), LLVM is more ready for native. For instance, DSE in LLVM is not suitable for a dynamic environment. This thesis enhances DSE to make it global and suitable for managed languages.

Dead Store Elimination is a technique to reduce the redundant store instructions in programs. Store instructions cost more cycles than most instructions. A program with many dead stores will drastically decrease the efficiency. As a result, it is important for a program to reduce the unnecessary stores.

1.1 Dead Store Elimination Pass in LLVM

LLVM’s Dead Store Elimination(DSE) pass is not strong enough to handle some of the situations. The limitation is that it only focuses on dead stores in a single basic block; that is, a store is a dead store only if we can find out the dependent instruction in the same basic block, and the behavior of the dependent instruction make the store not needed.

To eliminates dead stores thoroughly, a cross-block based approach is appropriate. It is needed to extend the instruction depdence relation for crossing blocks. Control flow graphs(CFG) are also been introduced. Several issues should be concerned in the cross-block dead store elimination.

1.2 LLVM Code Representation

To utilize LLVM passes, programs should be transformed into LLVM code representation so that the passes of LLVM can manipulate programs. The LLVM code representation is designed to be used in three different forms: as an in-memory compiler IR, as an on-disk bitcode representation (suitable for fast loading by a Just-In-Time compiler), and as a human readable assembly language representation.

LLVM code representation is an SSA-based representation. A key design point of an SSA-based representation is how it represents memory. In LLVM, no memory locations are in SSA form, which makes things very simple.

1.2.1 Memory Access and Addressing Operations

“alloca” instruction

The ‘alloca‘ instruction allocates memory on the stack frame of the currently executing function, to be automatically released when this function returns to its caller. The object is always allocated in the generic address space (address space zero).

```
%ptr = alloca i32                             ; yields {i32*}:ptr
%ptr = alloca i32, i32 4                      ; yields {i32*}:ptr
%ptr = alloca i32, i32 4, align 1024          ; yields {i32*}:ptr
%ptr = alloca i32, align 1024                 ; yields {i32*}:ptr
```

Figure 1.1 Examples of “alloca” instruction

“load” Instruction

The ‘load‘ instruction is used to read from memory.

```
%ptr = alloca i32                               ; yields {i32*}:ptr
store i32 3, i32* %ptr                          ; yields {void}
%val = load i32* %ptr                           ; yields {i32}:val = i32 3
```

Figure 1.2 Examples of “load” instruction

“store” Instruction

The ‘store‘ instruction is used to write to memory.

```
%ptr = alloca i32                               ; yields {i32*}:ptr
store i32 3, i32* %ptr                          ; yields {void}
%val = load i32* %ptr                           ; yields {i32}:val = i32 3
```

Figure 1.3 Examples of “store” instruction

1.3 Memory Dependency Analysis in LLVM

The DSE pass use Memory Dependence Analysis. This is an analysis that determines, for a given memory operation, what preceding memory operations it depends on. It builds on alias analysis information, and tries to provide a lazy, caching interface to a common kind of alias information query. The dependency information returned is somewhat unusual, but is pragmatic.

If queried about a store or call that might modify memory, the analysis will return the instruction[s] that may either load from that memory or store to it. If queried with a load or call that can never modify memory, the analysis will return calls and stores that might modify the pointer, but generally does not return loads unless a) they are volatile, or b) they load from *must-aliased* pointers. Returning a dependence on must-alias'd pointers instead of all pointers interacts well with the internal caching mechanism.

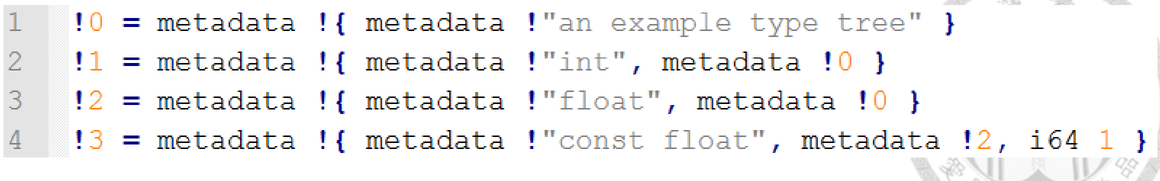
1.4 Alias Analysis in LLVM

Alias Analysis is a class of techniques which attempt to determine whether or not two pointers ever can point to the same object in memory. There are many different algorithms for alias analysis. You can use diferent alias analysis at the same time. LLVM chains the result. All of the AliasAnalysis virtual methods default to providing *chaining* to another alias analysis implementation, which ends up returning conservatively correct information (returning “May” Alias and “Mod/Ref” for alias and mod/ref queries respectively).

1.4.1 Type Based Alias Analysis

In LLVM IR, memory does not have types, so LLVM's own type system is not suitable for doing TBAA. Instead, metadata is added to the IR to describe a type system of a higher level language. This can be used to implement typical C/C++ TBAA, but it can also be used to implement custom alias analysis behavior for other languages.

The current metadata format is very simple. TBAA MDNodes have up to three fields, e.g.:



```
!0 = metadata !{ metadata !"an example type tree" }
!1 = metadata !{ metadata !"int", metadata !0 }
!2 = metadata !{ metadata !"float", metadata !0 }
!3 = metadata !{ metadata !"const float", metadata !2, i64 1 }
```

Figure 1.4 TBAA MDNodes

The first field is an identity field. It can be any value, usually an MDString, which uniquely identifies the type. The most important name in the tree is the name of the root node. Two trees with different root node names are entirely disjoint, even if they have leaves with common names.

The second field identifies the type's parent node in the tree, or is null or omitted for a root node. A type is considered to alias all of its descendants and all of its ancestors in the tree. Also, a type is considered to alias all types in other trees, so that bitcode produced from multiple front-ends is handled conservatively.

If the third field is present, it's an integer which if equal to 1 indicates that the type is "constant".

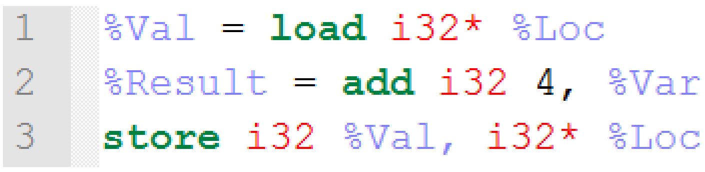
Chapter 2 Design and Implementation of LLVM Dead Store Elimination

As mentioned preveiously, LLVM’s DSE pass does not deal with cross-block dead store instructions. The following section will show what LLVM DSE does, and what dose not.

2.1 Single BasicBlock Dead Stores

The method of LLVM’s DSE for detecting dead stores can be classified into two category. The first one is Load-Store redundancy and the second one is Write-Write redundancy. For the first case, the latter store instruction is redundant, for the second one, the prior write instruction is redundant.

2.1.1 Load-Store Redundancy

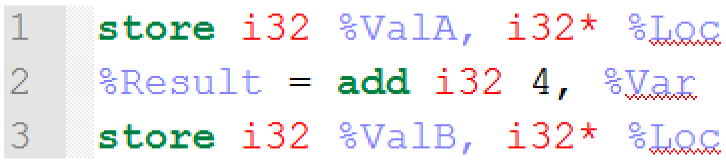


```
1  %Val = load i32* %Loc
2  %Result = add i32 4, %Var
3  store i32 %Val, i32* %Loc
```

Figure 2.1 An example of Load-Store redundancy

Figure 2.1 shows that storing the same value back to a pointer which just loaded from. Line 1 load a value to Val from Loc whitch pointed to, and line 3 store Val back to Loc. It is easy to find out the store at line 3 is a dead store.

2.1.2 Write-Write Redundancy



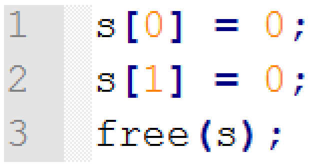
```
1  store i32 %ValA, i32* %Loc
2  %Result = add i32 4, %Var
3  store i32 %ValB, i32* %Loc
```

Figure 2.2 An example of Write-Write redundancy

Figure 2.2 shows that two consequtive stores modifying the same location without any interleaving use. Line 1 store ValA to location Loc, and line 3 store ValB to the same location. The second store is obviously redundant.

2.2 Handling Functions

2.2.1 Handle Free Call



```
1  s[0] = 0;
2  s[1] = 0;
3  free(s);
```

Figure 2.3 Store a value to a location which will be freed later

A store instruction storing to a location(in a part of a structure) which will be freed later is a redundant store. In Figure 2.3, line 1 and line 3 will be viewed as redundnat instructions.

When DSE pass visits a free call to free a memory location, it regards any memory write instruction referencing to the freed memory location as a dead instruction. The write instruction will be removed and recursively finding other dependent write instruction toward the unconditional branch predecessor of the basic block at which the free call located.

2.2.2 Handle End-Block

In the end-block of a function, it is no use to store a value to a local stack which will not be used later, or any memory location which does not escape. These stores are dead stores. DSE pass adds all the alloca instruction and heap pointers which do not escape the function to DeadStackObjectSet. Then examing all the instructions bottom-up in the end-block. If visiting a write instruction storing to any object in DeadStackObjectSet, then the write instruction is a dead store. If visits a load or load-like instruction referencing any object in DeadStackObjectSet, the object would be moved out DeadStackObjectSet.

2.3 Important Memory Dependence Analysis Functions

- getDependency

  Return the instruction on which a memory operation depends in a basic block.
- getNonLocalPointerDependency
- Return a set of instructions on which a memory operation depends in basic blocks.

Chapter 3 Global Dead Store Elimination

3.1 Control Flow Graph

A control flow graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution. To recognize dead stores, CFG should be introduced.

3.2 Dominator

Dominator information is required to find out global dead stores. In control flow graphs, a node d dominates a node n if every path from the start node to n must go through d. Notationally, this is written as d dom n. By definition, every node dominates itself.

3.2.1 Postdominance

Analogous to the definition of dominance above, a node z is said to post-dominate a node n if all paths to the exit node of the graph starting at n must go through z. Similarly, the immediate post-dominator of a node n is the postdominator of n that doesn't strictly postdominate any other strict postdominators of n.

3.3 Global Dead Stores Recognition

There are several global dead store types, and none of them can be removed by LLVM DSE.

3.3.1 Dependent Instruction

If instruction A is dependent to instruction B, then A and B both refer to or modify the same memory address. If we are not sure whether the memory space referred or modified by A and B is the same or not, we let them be dependent instruction to each other.

To be more preciesely, a memory write instruction A writing to a memory address Ptr is dependent to a instruction B which would be execute later if

1. B is also a memory write instruction, and B writes to Ptr.
2. B is also a memory write instruction, and we don’t know where B writes to.
3. B is a memory read instruction, and B loads a value from Ptr.
4. B is a memory read instruction, and we don’t know where B loads a value from.
5. B is a call instruction, Ptr has escaped, and we can know B refers to or modifies Ptr.
6. B is a call instruction, Ptr has escaped, and we can not determine whether B refers to or modifies Ptr.

3.3.2 Load-Store Redundancy

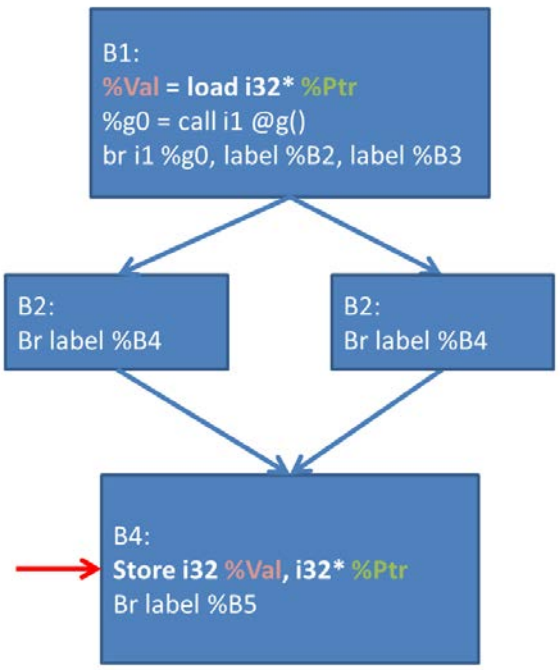


Figure 3.1 Load-Store Redundancy

At a basicblock, says Bi, a value loaded from a pointer, and all pathes from Bi to a basicblock, says Bj, containing a store instruction at Bj storing the value just loaded to the pointer. To be noticed is that all ofthe path from Bi to Bj should not exist any other memory write instruction to the address which the pointer points to.

3.3.3 Write-Write Redundancy

The following section dicusses some cases which results to wite-wite redundancy, and conclude what the attributes are that make the redundancy.

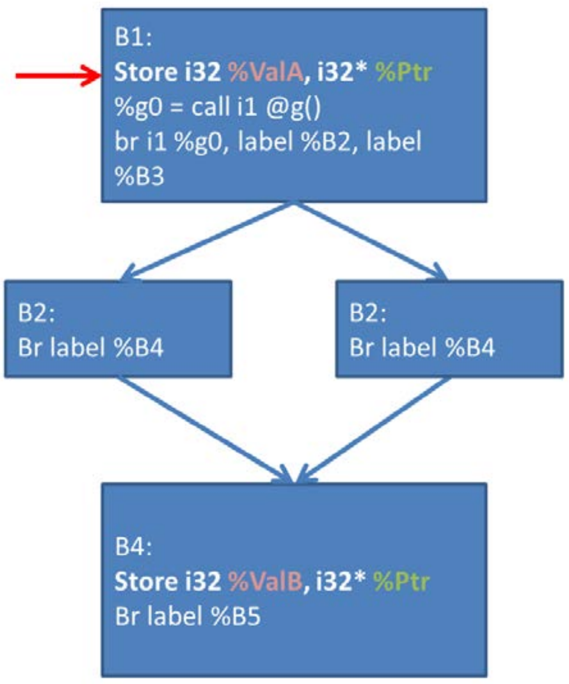

B1:
Store i32 %ValA, i32* %Ptr
%g0 = call i1 @g()
br i1 %g0, label %B2, label %B3
B2:
Br label %B4
B2:
Br label %B4
B4:
Store i32 %ValB, i32* %Ptr
Br label %B5

Figure 3.2 1st Write-Write Redundancy Example

Figure 3.2 shows that at a basicblock, says Bi, there exists a store instruction writes to a pointer, Ptr, for all path from Bi must pass a basicblock, Bj, and Bj exists a store instruction writes to Ptr. To be noticed is that there is no other load or store in all pathes.

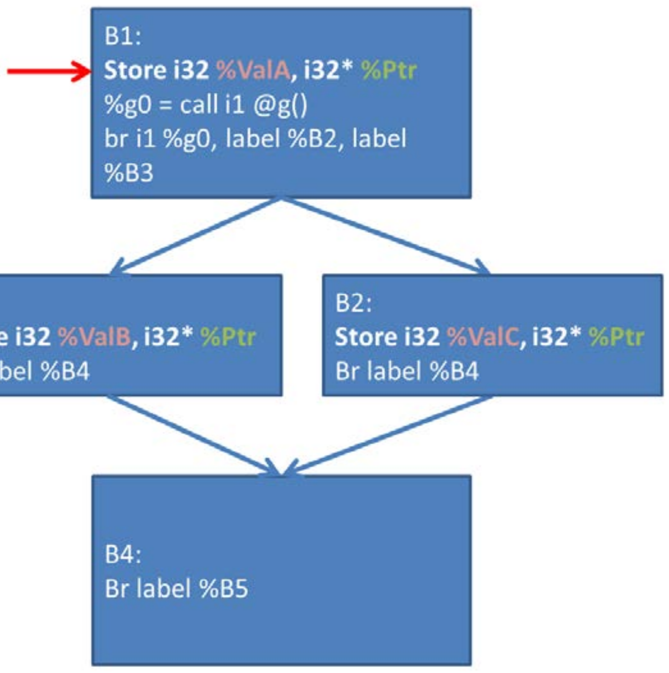

B1:
Store i32 %ValA, i32* %Ptr
%g0 = call i1 @g()
br i1 %g0, label %B2, label %B3
B2:
Store i32 %ValB, i32* %Ptr
Br label %B4
B2:
Store i32 %ValC, i32* %Ptr
Br label %B4
B4:
Br label %B5

Figure 3.3 2nd Write-Write Redundancy Example

Figure 3.3 shows that at a basicblock, Bi, there exists a store instruction writes to a pointer, Ptr, and all the successors of Bi must contain one write instruction writes to Ptr. To be noticed, there are no other store and load intructions in the basicblocks in the pathes leaving from Bi.

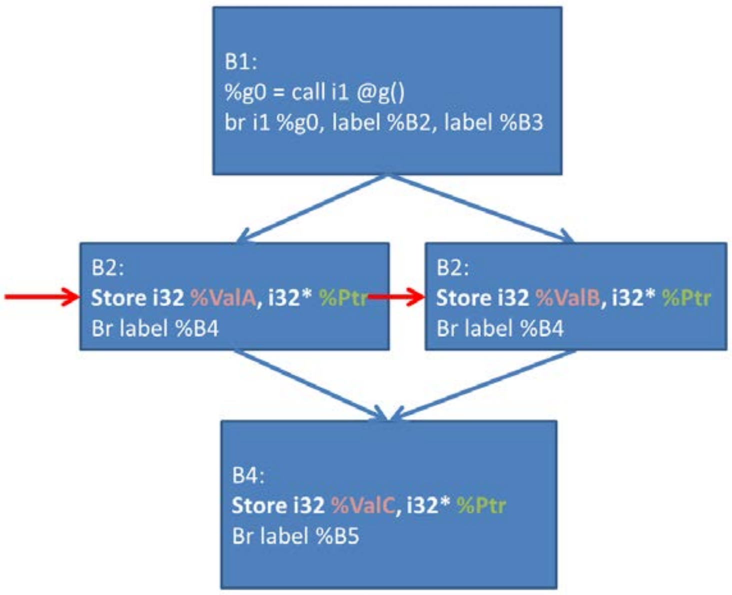

B1:
%g0 = call i1 @g()
br i1 %g0, label %B2, label %B3
B2:
Store i32 %ValA, i32* %Ptr
Br label %B4
B2:
Store i32 %ValB, i32* %Ptr
Br label %B4
B4:
Store i32 %ValC, i32* %Ptr
Br label %B5

Figure 3.4 3nd Write-Write Depdendency Eample

Figure 3.4 is jus like Figure 3.2 with 2 redundant store instructions.

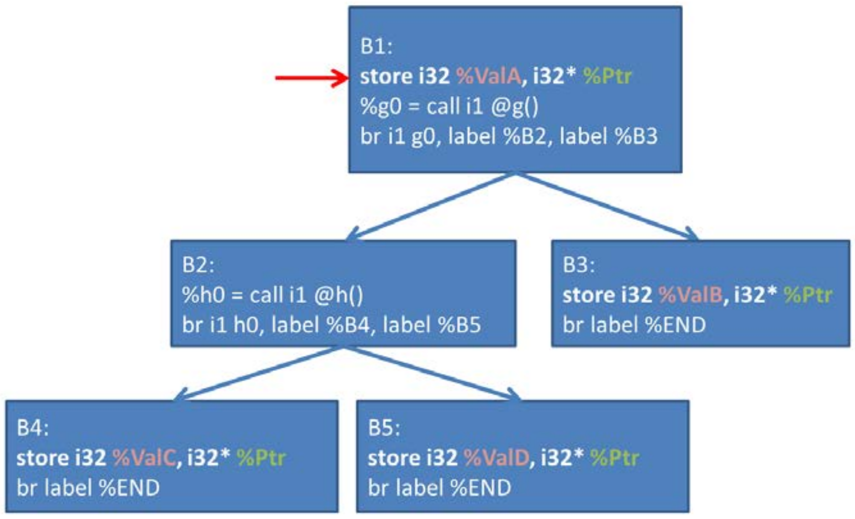

B1:
store i32 %ValA, i32* %Ptr
%g0 = call i1 @g()
br i1 g0, label %B2, label %B3
B2:
%h0 = call i1 @h()
br i1 h0, label %B4, label %B5
B3:
store i32 %ValB, i32* %Ptr
br label %END
B4:
store i32 %ValC, i32* %Ptr
br label %END
B5:
store i32 %ValD, i32* %Ptr
br label %END

Figure 3.5 4th Write-Write Dependency Example

Figure 3.5 shows that at a basicblock, Bi, there exists a store instruction writes to a pointer, Ptr, and all the successors of Bi must contain one write instruction writes to Ptr. If any of Bi's sucessors, says Bj, does not contain a write instruction to Ptr, then all of Bj's successors must contain a write instruction to Ptr.

To conclude above, if a write instruction writing to a pointer Ptr located at basicblock Bi is dead, for all pathes leaving Bi to basicblocks B_j B_{j+1}… B_k, B_j B_{j+1}… B_k should contain a write instruction writing to Ptr. If one of successors of B_i, B_m, does not contain a write instruction writing to Ptr, then all of the successors of B_m should contain a write instruction writint to Ptr.

3.3.4 Write to Dead Stack Objects

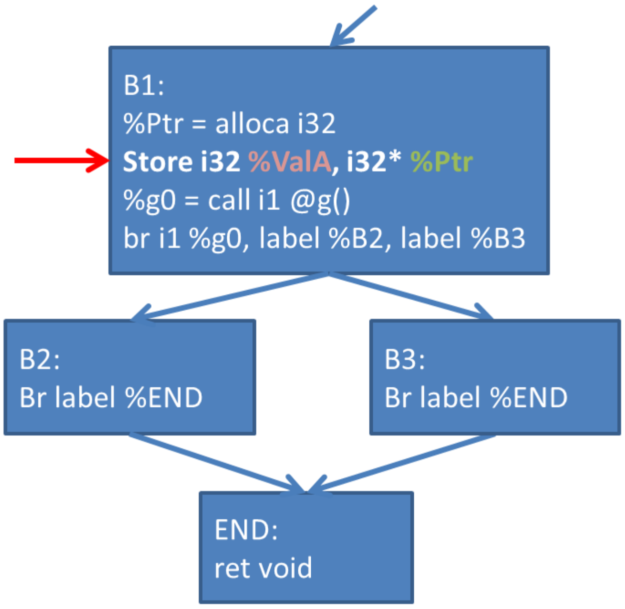
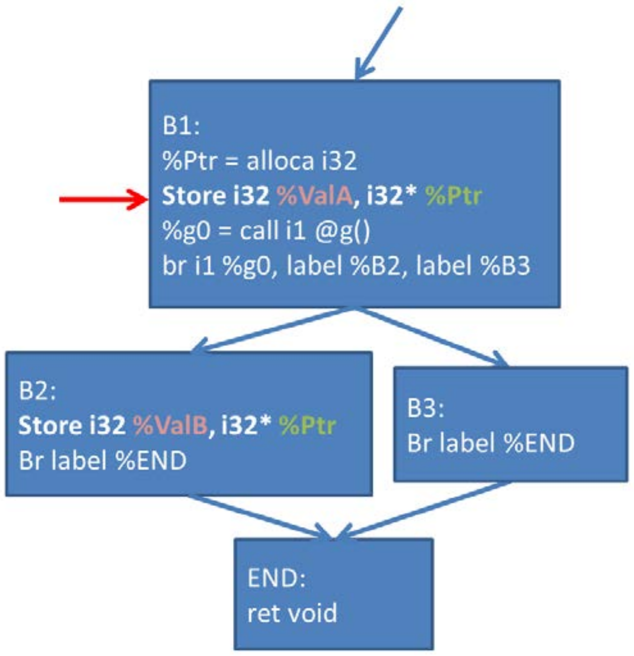


Figure 3.5 Write to dead stack object

If a write instruction writing to a pointer Ptr, and the memory location is never been loaded or rewritten. The write instruction is a redundant instruction. Ptr points to local stacks, or a heap address which never escapes the function.



```
B1:
%Ptr = alloca i32
Store i32 %ValA, i32* %Ptr
%g0 = call i1 @g()
br i1 %g0, label %B2, label %B3
```

```
B2:
Store i32 %ValB, i32* %Ptr
Br label %END
```

```
B3:
Br label %END
```

```
END:
ret void
```

Figure 3.6 Another write to dead stack object example

3.4 Implementation

3.4.1 Load-Store Dead Store Elimination

LLVM Memory Dependence Analysis provide a function to determine what instructions a memory instruction dependent to .

- getNonLocalPointerDependency - Perform a full dependency query for an access to the specified (non-volatile) memory location, returning the set of instructions that either define or clobber the value. The set of instructions must lead to where the store instruction located.

1. If we visit a store instruction storing value to a pointer, Ptr, querry function getNonLocalPointerDependency. The getNonLocalPointerDependency

function returns a set of instructions, S.

2. If the set S only contains a load instruction, and the value loaded by the load instruction is just the same as the stored value, then the store instruction is redundant.

3.4.2 Write-Write Dead Store Elimination

To detect this case, we use a modified depth first search. For the performance issue, the iterative DFS is applied.

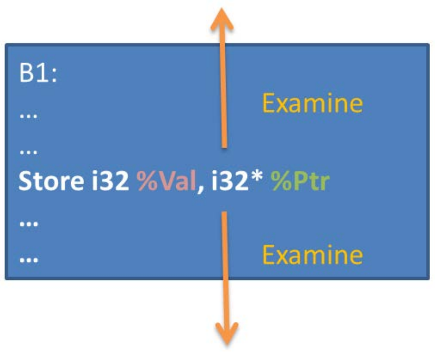
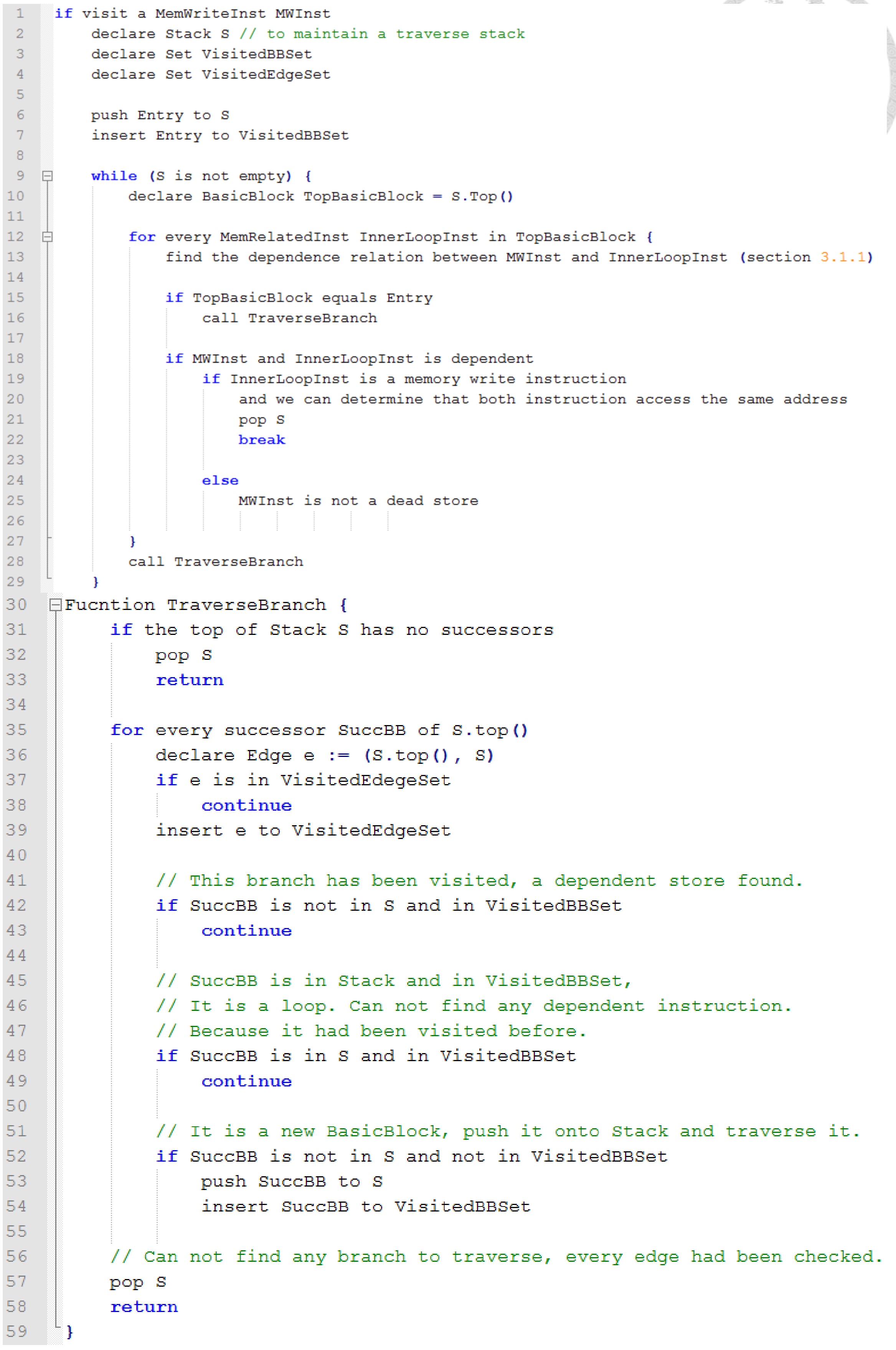


Figure 3.7 Examine the dependent relation of the memory write instruction in the basicblock

1. If we visit a memory write instruction, we need to examine (Figure 3.7) dependent relation of the memory write instruction in the basicblock. If there is a dependent instruction in the upward side, then we call the memory write instruction ‘local dependent’. If there is a dependent instruction in the downward side then the elimination pass is over, and the memory write instruction is not a global dead store.
2. We use DFS to walk through all possible path, examining every memory-related or function all instruction. (Figure 3.8 shows the pseudo code)



```
if visit a MemWriteInst MWInst
    declare Stack S // to maintain a traverse stack
    declare Set VisitedBBSet
    declare Set VisitedEdgeSet

    push Entry to S
    insert Entry to VisitedBBSet

    while (S is not empty) {
        declare BasicBlock TopBasicBlock = S.Top()

        for every MemRelatedInst InnerLoopInst in TopBasicBlock {
            find the dependence relation between MWInst and InnerLoopInst (section 3.1.1)

            if TopBasicBlock equals Entry
                call TraverseBranch

            if MWInst and InnerLoopInst is dependent
                if InnerLoopInst is a memory write instruction
                    and we can determine that both instruction access the same address
                    pop S
                    break

                else
                    MWInst is not a dead store

        }
        call TraverseBranch
    }
Fucntion TraverseBranch {
    if the top of Stack S has no successors
        pop S
        return

    for every successor SuccBB of S.top()
        declare Edge e := (S.top(), S)
        if e is in VisitedEdegeSet
            continue
        insert e to VisitedEdgeSet

        // This branch has been visited, a dependent store found.
        if SuccBB is not in S and in VisitedBBSet
            continue

        // SuccBB is in Stack and in VisitedBBSet,
        // It is a loop. Can not find any dependent instruction.
        // Because it had been visited before.
        if SuccBB is in S and in VisitedBBSet
            continue

        // It is a new BasicBlock, push it onto Stack and traverse it.
        if SuccBB is not in S and not in VisitedBBSet
            push SuccBB to S
            insert SuccBB to VisitedBBSet

    // Can not find any branch to traverse, every edge had been checked.
    pop S
    return
}
```

Figure 3.8 DFS Algorithm walking through all possible path

3. Afther executing algorithm in Figure 3.8, the memory write instruction still not be seen as live (contrary to dead), then it is a dead store.

3.4.3 Dead Store on Local Stack Object

1. A memory write instruction MemInst writes to a memory address, Ptr.
2. Use DFS in Figure 3.8 to find out whether there is a dependent instruction in the successors.

A. If there are no dependent instructions in all pathes till function end-block.

B. or, if there exist dependent stores (accessing the same memory address as Ptr) in some pathes and no dependent instructions in all the other pathes till function end-block

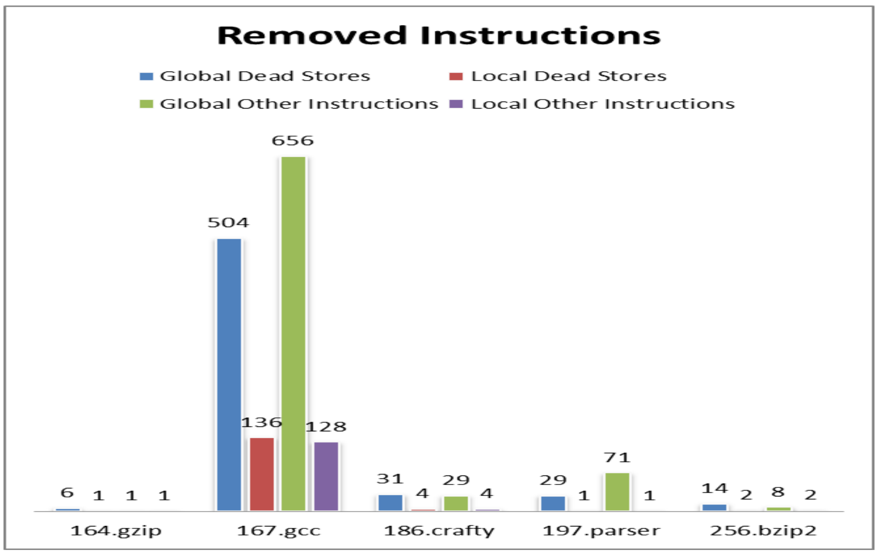
3. If the memory write instruction satisfied A or B, then it is a dead store.

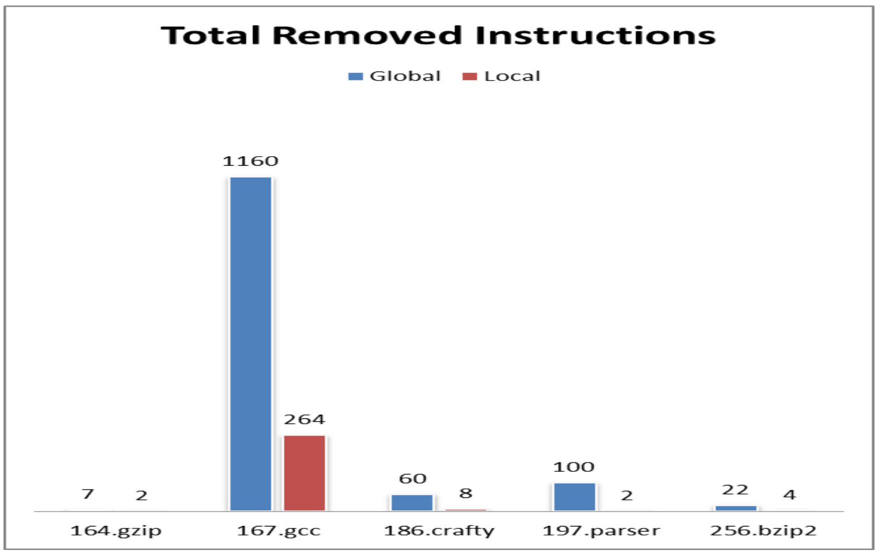
Chapter 4 Experiment

4.1 Platform Enviroment

- Intel Core i7-2600 CPU @ 3.40GHz
- Ubuntu 64bit 12.04 LTS
- Clang version 3.4 (trunk 184282)
- LLVM 3.4svn – optimized build with assertions
- Benchmark - SPEC2000 CINT2000

4.2 Result





	Global Dead Stores	Local Dead Stores	Global Other Instruction	Local Other Instruction	Global Total Instruction	Local Total Instruction
164.gzip	6	1	1	1	7	2
167.gcc	504	136	656	128	1160	264
186.crafty	31	4	29	4	60	8
197.parser	29	1	71	1	100	2
256.bzip2	14	2	8	2	22	4

The coloum "Other Instruction" refers to how man instructions are been removed due to the corresponding store instructions been removed. We can find that global DSE reduced many dead instructions.

Program Size (byte)			
	DSE	GDSE	Change%
164.gzip	62547	62575	0.04
167.gcc	1847494	1843528	-0.21
186.crafty	242067	242141	0.03
197.parser	153257	153291	0.02
256.bzip2	53983	53987	0.01

We can find that although there are many instruction had been removed, but the code size in the small program-size benchmark are a little bigger than the original DSE. In the bigger program-size benchmark (167.gcc), the number of deleted instructions is direct proportion to the program-size. I think it is due to the level of the instruction selection.

Performance			
	DSE	GDSE	Accelerate%
164.gzip	116.13	115.97	0.14
167.gcc	37.54	37.44	0.26
186.crafty	37.22	37.19	0.07
197.parser	117.35	116.94	0.35
256.bzip2	387.19	385.01	0.56

The performance has no significant improvement due to SPEC2000 is a C-based benchmark. There are not many store instructions comparing with managed programs.

Chapter 5 Conclusion

The global dead store problem is handled thoroughly without code-motion. Although GDSE is not shinning in the area of native language, it will have a great influence in the near future when the managed program are supported by the LLVM.

Chapter 6 Reference

(Lattner and Adve 2004, Lattner 2008)

Lattner, C. (2008). LLVM and Clang: Next generation compiler technology. The BSD Conference.

Lattner, C. and V. Adve (2004). LLVM: A compilation framework for lifelong program analysis & transformation. Code Generation and Optimization, 2004. CGO 2004. International Symposium on, IEEE.

Sair, S. and M. Charney (2000). "Memory behavior of the SPEC2000 benchmark suite." IBM TJ Watson Research Center Technical Report.

(Sair and Charney 2000)