Determination of Safe regions in Optimized Dynamically Updatable Programs

Submitted in partial fulfillment of the requirements for the degree of

Master of Technology

by

Abhirup Ghosh

Roll No : 09305052

under the guidance of

Prof. D. M. Dhamdhere

Department of Computer Science and Engineering

Indian Institute of Technology, Bombay

June 2011

Declaration

Abhirup Ghosh

Roll No. : 09305052

Date: June 27, 2011

Abstract

Every software needs to be updated either to fix bugs or to add new features. To update a running system, developers typically shut down the system for a while, apply the patch and then resume execution. However, many continuously running system cannot afford to halt or would prefer not to. One of the most convenient solutions to this problem is dynamic software update where the program is updated while the program is in execution - the running program is stopped at program point p, then the program is updated to new version and after that execution resumes from p onwards. p should be a *safe point* to have the dynamic update *safe*. Till date, many dynamic updating systems have been developed like Ginseng, JVolve etc. None of the present literature has defined update safety in dynamically updatable programs. They have concentrated on type safety. We would define update safety in this report. Also, none of these works considered program optimization issues at the time of determining *safe points*. However, both the old version and new version codes are optimized by the compiler, so update to old version code may enable some optimization opportunities that were not possible otherwise or it may disable some previously performed optimizations. Therefore, not only the updates but also the optimizations and de-optimizations resulting from the updates change the code. In this report, we present a methodology to determine safe points in a program considering both updates and enabling and disabling of optimizations by the updates.

We have implemented a system in LLVM to determine the safe regions in a program considering program optimizations. Our system can be integrated with any dynamic update system.

Acknowledgments

I would like to express my sincere gratitude to my guide Prof. D.M. Dhamdhere. I am deeply indebted to him for the guidance and encouragement that he provided throughout the duration of the project. He has constantly motivated me to come up with my own ideas.

Abhirup Ghosh
IIT Bombay

Monday, June 27, 2011

Contents

List of Figures

Chapter 1

Introduction

Many systems need to run continuously. For example, reactive systems like chemical plant monitoring systems and mission critical systems like banking systems and online retails should run 24/7. Service oriented architecture uses pay-per-use business model, so temporary unavailability may cause failures and thus losses in business. Loss due to downtime in highly available systems may go up to thousands to millions of dollars. [1] reports that 75% of 6000 outages of highly available applications are because of software and hardware maintenance. Systems commonly have bugs and they need new features to be added. Traditionally, to update applications they are shut down - resulting in downtime. A solution to this problem is to use stand by systems. However, this solution is not only complex to implement but also not much cost-effective. A more elegant, simple and cost-effective solution is to update the software dynamically. Besides commercial need, home users also feel satisfied as a software update does not require restarting the application or in case of OS update, restarting the machine.

```
 S1: .....                    S1: .....
 S2: while (i<j) {            S2: while (i<j) {
 S3: ...                      S3: ...
 S4: c = a + 5                S4: c = a + 6
 S5: ...                      S5: ...
 S6 : print c                 S6 : print c
 S7: }                        S7: }

*Program Version V1*         *Program Version V2*
```

Figure 1.1: Motivating example

A dynamic update system stops (if the to be updated function is active) the execution of the program at a certain program point to update the program, then applies the dynamic patch, and after that resumes execution of the updated program from that program point onwards. A program cannot be stopped at any arbitrary program point to update. In Figure 1.1, program version V1 is modified to program version V2 by updating an operand of statement S4. If the program is stopped at S5 to do the update then variable c is already computed in the current iteration of the loop. Thus, when after update for the first time control reaches S6, variable c has the value (a+5) according to V1. However, it ought to have the value (a+6) according to V2. Hence, we call the update as *unsafe*. If the program is stopped at S3 and updated, then after resuming execution when the control reaches S4 it computes the value of the variable c as per V2. Therefore, when the value of c printed in S6 it is according to V2 resulting in a *safe* update.

In the last few decades, a significant amount of work has been done in the field of developing on-line update systems. However, none of these works have defined update safety. Most of them ignored update safety and concentrated more on type safety or they are relying on programmers for specifying safe update points and they do not have automatic methods to check validity of the update. Gupta, Jalote and Barua in [2] defined validity of an update. However, as we will comment in chapter 3 , this notion of validity is highly restrictive in the sense that many updates of practical utility would not be valid according to their definition. In this report, we would define update safety and we have developed a system to automatically determine safe update points. Detailed definition of update safety is later in chapter 3. In simple words, update safety can be described as - it is safe to apply an update U at execution point p only if all results produced during further execution of the program from point p will be according to the new version of the program. Then p is called a *safe point* for this update and the update is called as safe update. Otherwise, if the execution from the point of update is not according to the new version of the program then the update is called as unsafe update and the execution point is called as *unsafe point*. In the above example Figure 1.1, S3 is a safe update point whereas S5 is an unsafe update point.

A set of contiguous safe update points constitute a *safe region*. A set of contiguous unsafe update points constitutes an *unsafe region*. The whole program is divided into safe and unsafe regions. In the above example, code region from S4 to S6 is an unsafe region for the update specified. All other regions are safe in that code snippet. Safe updates are the requirement of every system. Thus, the need of finding the safe region or unsafe region is important.

While a program runs, generally it is an optimized one. Compiler optimizes a program at the time of compilation. Compiler optimization analyses and transforms the program code to reduce mostly execution time and rarely space. Every optimization is performed if some *pre-conditions* are satisfied in the code. If the pre-conditions of an optimization is satisfied in the code then the code is transformed according to some code change rules. If pre-conditions for optimization O are not satisfied in a program but satisfied after updating the program with update U then O is said to be *enabled* by U. Consider an optimization O is performed in a program and then the program is updated with update U. If the updated program no longer satisfies the pre-conditions of O then O should be undone. Here O is said to be *disabled* by U. Therefore, the update is not the only source of code change in a program - but the follow up optimization and de-optimization also changes the code. Hence, a safe point in an unoptimized program may not be a safe point in an optimized program. We illustrate it in the program of Figure 1.1.

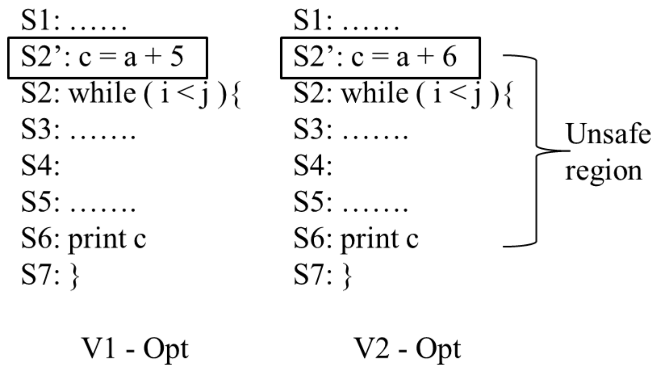


Figure 1.2: Motivating example Optimized

Program version V1 in Figure 1.1 is optimized with loop invariant code motion optimization to get V1-Opt in Figure 1.2. After update, optimization is still possible and analogously V2-Opt is obtained from V2. System would actually run the executable corresponding to V1-Opt. Previously when optimization was not considered, we decided S3 as a safe point. However, if optimization is considered and the update is applied at S3 then value of variable c is already computed outside the loop at S2'. Therefore, after update printing of variable c at S6 would be according to old version code. Thus, the update would be unsafe. Here, unsafe region is from S2' to S7. Therefore, it is clearly seen that optimizations should be considered while finding safe region in a program as the safe points in an unoptimized program may not be safe when the program is optimized.

1.1 Problem statement

The problem is to determine the safe region in a dynamically updatable program for applying an update considering optimization and de-optimization issues. It is about designing a system (S), which would analyze the program, updates and optimization issues to decide the safe regions. Figure 1.3 shows inputs and outputs of the system.

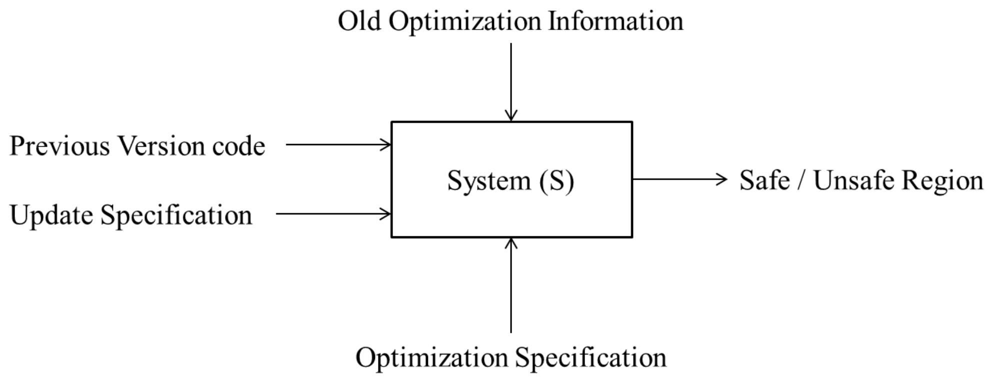


Figure 1.3: System to determine unsafe or safe region for an update

- Old version code: - S needs to know what the previous version of the code was.
- Update specification: - This includes information about the code changes in the old version - statements to be inserted or deleted. Every code change can be described in terms of these two fundamental code changes.
- Optimization specification: - Desired system should know the exact specification of the optimization that will be applied to the program by the compiler. Specification should include what are the conditions that a code must satisfy to enable an optimization (called as *pre-condition* of the optimization) and also if this optimization is enabled then what would be the change in the code (called as *actions* of optimization.)

- Old optimization information: - S needs to know about the optimizations that were performed in the previous versions of the code, more precisely what code changes were made during optimization of the previous versions. From this information, S decides if any of those optimizations are disabled.

1.2 Contribution of the work

To our knowledge, for the first time in literature we are defining the idea of safety in a dynamic update system and developing a system which automatically detects safe regions in a program. Our system considers optimizations to be done by the compilers and this is the first time in the literature where optimizations are considered while finding safe points. Given the previous version code, update specification, specifications of the optimizations to consider and information about the previously performed optimizations the system gives the unsafe regions in the program for that update. To summarize, contribution of our work is the following -

- This is the first attempt to define update safety in a dynamically updatable program.
- This is the first attempt in the literature to incorporate the idea of optimization and de-optimization for determining safe region for dynamic updates.
- The system has an input of optimization specification. Therefore, our system can work with any optimization that is specified. The system design is also independent of the specification language for the optimization. So, the limitation of any specification language may be eliminated by using other optimization specification languages.
- The system design can work with programs written in any programming languages.

1.3 Overview of the Dissertation

Section 1.1 describes the problem statement we are going to solve. Related works are described in chapter 2. In chapter 3 we will define update safety and also will discuss why the previous definitions of validity is not sufficient. Modules needed for the solution of the problem is discussed in chapter 4. Overview of the scheme is presented in section 4.5.

chapter 5 describes optimization specification language SpeLO [3]. In chapter 6 we will describe the design of our system. The detailed schematic diagram is in section 6.3. Implementation details are described in chapter 7. In this chapter we will describe pseudo codes of the relevant algorithms. Chapter 8 contains concluding remarks along with future scope of the system.

Chapter 2

Related Work

Dynamic software update system (DSU) is a widely studied topic in computer language domain. Most of the literature concentrates on implementing a working DSU system guaranteeing type-safety. They often pay less attention to determination of safe points for update and use very conservative approximation while determining safe points. Although DSU systems like Ginseng [4], JVolve [5] showed new perspective of thinking in this field. None of the current DSU systems consider compiler optimization issues while determining safe points.

2.1 Ginseng

Hicks et al. in [4] built a DSU implementation for C that would not not violate type-safety as well as would keep the data up-to-date. According to the authors. DSU system should have three essential characteristics -

1. DSU should not enforce extensive changes to applications to support dynamic update. Applications should be allowed to be written in natural style. Ideally application programmers should not have to worry about dynamic update system.

2. DSU should support as many types of update as possible. There should be minimum cases where the update is not performed because DSU does not support them.

3. The updates should be easy to write and they should be such that their correctness can be proved easily.

Authors in [4] have developed Ginseng, a DSU implementation for C aimed at single threaded server applications and it satisfies first two criteria.

2.1.1 Ginseng overview

Ginseng has three basic parts - a compiler, a patch generator and a run-time system.

- Compiler - It compiles programs to make them dynamically updatable. For example, Ginseng compiler declares a global function pointer for each function in the program and assigns current version functions to the corresponding function pointers. It also replaces all calls to functions by respective function pointers. Therefore, updating a function only requires assignment of the new version function to the global function pointer for that function.

- Patch Generator - Generates dynamic patch from the version data, old version code and the new version code. Dynamic patch consists of new and updated functions and global variables, type transformers and state transformers. By comparing new and old version of code, patch generator identifies those definitions that has been changed and then generates type transformers to convert the old values to new values. For example the update is to add new field in a structure. For this update, in all the existing instances of that structure, old values are copied and the extra field is added with a default value like null.

- Run time system - Dynamically loads and links the dynamic patch in next safe update point. There is a safety analysis process to determine whether a program point is safe or not. If a program point is safe then only patch is deployed.

2.1.2 Enabling on-line updates

There are two approaches to update functions and types in a program -

- Function Indirection:- Use a function pointer (global variable) to point to the current version of the function and whenever there is a need to call the function, call through the function pointer. If the function is modified then that function pointer points to the updated implementation.

- Type Wrapping:- Original type definition can be wrapped into a type containing extra information of version number and padding space for the extra fields that might be added in future. This extra padding enables dynamically updated types to update type instances in place i.e. updated data replaces the old data in the same memory location. So, the location of data is not changed due to update. To transform the type there should be a type transformer function that would make comparison of the type version being used and the previous version of the type and then make necessary changes. There are two approaches to deal with this problem -

- At the time of update, change all the instances of modified type (T). This can be done by garbage collector style approach (trace all instances of T that are reachable and then transform all of them).The system maintains a record of all live instances of the type T at every program point and transform the live instances. Disadvantage of this approach is that it increases transformation cost as at the time of update the system has to update all possible variable values.
- Lazy approach - Whenever a value of a changed type is accessed then its value is transformed according to the current type. This removes the disadvantage of the previous approach. Ginseng uses this approach to update type values.

In Ginseng, if function f is to be updated then the update does not reflect in the current execution of f. Update takes place at the next call to f. This happens because Ginseng uses function indirection technique to perform update a function. If f has an infinite loop then it may happen that current call to f never finishes and f is never updated. Solution to this problem is loop extraction. Code inside the loop is extracted to a function and the function is called from inside the loop in each iteration. Any update to the code in the loop is essentially an update to the extracted function. So, when update is performed current iteration of the loop is not affected but from the next iteration updated code is executed (new version of the extracted function is called).

2.1.3 Safety analysis

Ginseng requires the programmers to specify the update points by calling a function called DSU_UPDATE. It tracks the changes of the types, analyzes the dangerous aliases, looks for unsafe casts and void*. Ginseng uses constraint based, flow sensitive updatability analysis.

- Tracking changes to types:- It keeps track of the set of types that cannot be updated at a potential update point. Set of types that can not be updated is called *capability*. Therefore, other types that are not present in this set can be updated at that point.

- Abstraction-violating Aliases:- Aliases are dangerous to update. It is safe to mark the types that have aliases as non-updatable. This is a very strict restriction. More flexible one would be if a to be updated type(T) has any live instance with aliases, then T is to be added to capability set at that program point.

- Unsafe Casts and Void*:- Type casting is unsafe as the two types participating in casting may have different type transformers. A simple solution may be to mark all such types participating in casting as non-updatable. However, this is a strict restriction. Generic programming in C uses void*. However, void* should not be allowed. If it is allowed then all casting is possible via void* casting.

Safe region determination does not consider optimization and de-optimization issues. Hicks et al in [4] have taken an approach to find safe points at function level granularity. However, the idea of type wrapping and function indirection is useful to have dynamic updates enabled in a program. Although these code changes bring about overhead in the steady state execution. Restriction in using void* in C is a serious restriction for generic programming.

2.2 Dynamic Software Updates: A VM-centric Approach

Subramanian et al [5] discusses a DSU system - Jvolve, a DSU enhanced Java VM. Jvolve is safe and flexible. Current version of code and the new version of code are submitted to the update preparation tool (UPT). UPT prepares the JvolveTransformer.java file which contains the transformer functions. Then JVolve VM in next suitable DSU safe point applies the update.

Type update is taken care of by transformer functions. Transformer functions copy old fields' values from the old instance to new instance and add new fields with default values.

These functions are static functions. In general, transformation functions cannot access the private fields of a class and cannot modify final fields. Therefore, JVolve disables the access modifiers for their implementation.

2.2.1 DSU Safe Points

DSU safe point is the subset of VM safe point. Program points where it is safe for garbage collection and thread scheduling are called as VM safe points. Over these constraints, there are some more restrictions to make VM safe points DSU safe points. DSU safe points require that no threads' call stack contain restricted methods. Restricted methods can be of three categories -

1. Methods changed by updates.
2. Methods that are not modified directly but have called some functions that have been modified or used some modified classes.
3. User blacklisted methods - Programmers or users of the JVolve system specifies these methods as unsafe.

If a method is inline and it is modified then all the methods where it is used should be included in the restricted methods.

2.2.2 Reaching Safe Points

When the update is ready user sets a yield flag. In the next VM safe point all threads halt. Then JVolve VM analyzes if that point is a DSU safe point. If a restricted method is found in the call stack of any thread then a return barrier is set for the top most restricted method in each call stack. A return barrier replaces regular method return branch with the branch to the code to perform update. This lets the DSU know when that method is finished. Then the JVolve VM sets more return barriers if still some restricted methods are in the call stack or if there is no restricted method in any of the call stack then it starts updating.

2.2.3 Lifting category restriction

If the method falls in the restricted category (2) and it is active then Jvolve uses a standard scheme - OSR (On Stack Replacement) present in Jikes RVM. In this scheme methods can be recompiled while they are in the call stack. Jlikes uses this scheme for implementing run time optimization. After reaching yield point OSR re-compiles the topmost method in the thread stack and manages the stack frame modification for that function. Jvolve customizes this scheme to implement updating active methods that are in restricted category (2). This methodology is safe as functions and class object addresses are to be modified for this kind of restricted methods.

2.2.4 Installing Modified Classes

Jikes RVM maintains a RVMClass object for every class for internal metadata storage. It also has TIB (Type Information Block) which maps method offsets to actual implementation of a method. Every object of that class has a pointer to TIB. When a method is called, it is modified to its latest version by compilation. This is called as base compiled. Then in further usages, the optimizer with the help of profile information optimizes the method.

JVolve uses return barrier approach to synchronize threads. This approach introduces unnecessary delay in application for update. Whether DSU safe points would be found out depends largely on thread scheduling. Jvolve generates a transformer class which transforms the old version process state to new version process state. Thus, transformer class should have access to all fields of old version class as well as new version class. However, Java type system forbids accessing the private fields in the classes. To solve this issue, Jvolve disables access-modifier. Thus, it restricts one of the main object oriented concepts for the sake of dynamic update. On the brighter side, after update, the code is base compiled and then code is optimized incrementally with each use of that code segment.

2.3 Safe and Timely Dynamic Updates for Multi-threaded Programs

Neamtiu et al in [6] discussed implementation of the Ginseng project for multithreaded approach. Detecting safe points for dynamic updates in multithreaded system is more challenging than for the single threaded approach. A simple approach can be - block the thread when it reaches a safe point and wait for other threads to reach safe program points. But this would have the following problems -

- Threads may be in deadlock. E.g. one thread acquires a lock and it is waiting after reaching a safe point and another thread is waiting for that lock to reach safe point.
- Even if deadlock does not result, timeliness would suffer. One thread may reach safe point earlier than the other threads and wait unnecessarily for others.

To eliminate these problems Neamtiu et al in [6] proposed two techniques - *induced safe points and relaxed synchronization*. When all threads reach safe points then only the update is performed. If the number of safe points in each thread is small then it is unlikely to have all the threads reach safe points quickly. Increase in the number of safe points in each thread increases the chance of quickly reaching safe points of each thread and thus improving timeliness.

2.3.1 Induced Safe Points

This approach aims to increase the number of safe points. Programmer needs to specify a few safe update points. Programmer provides a set of safe update points - Σ , and the code changes - π. Changes(π) are the name of types and functions that are to be updated.

To introduce more safe points the programmer should specify a set of candidate safe points ξ. A candidate safe point $l \in \xi$ is called an induced safe point if the effect of applying the changes (π) is same as applying the changes (π) in one of the safe points in Σ.

These induced points can be determined by static analysis. It is to be examined if the candidate safe point introduces version consistent updates. These can be checked by analyzing prior effect (α_l) and future effect (ω_l). Prior effect denotes static approximation of the execution behavior from prior definite update point to this point (l) and the future effect denotes the approximation from the current program point to the next definite update point. These sets have the name of the types and the functions that have been accessed in prior or later to the program point respectively.

If changes(π) $\cap$ $\omega_l = \Phi$ then for update π, program points after l are induced update points till next definite update point. These are called *roll-forward induced update points.* If changes(π) $\cap$ $\alpha_l = \Phi$ then the program points that are prior to l till prior definite update point are said to be *rollback induced update points.*

2.3.2 Relaxed synchronization

Induced safe points increases the number of safe points in each thread. However, this method forces a thread to block at a safe point and wait for other threads to come. Therefore, the performance degradation continues to happen. This problem is solved by relaxed synchronization technique. Check in happens when program control enters a definite update point. The run time system is notified of this check in. Check out happen when the program control reaches the last update point. The system is updatable when all the threads have checked in. But when one of them checks out then the system is not updatable and DSU has to wait to have all the threads check in.

In [6], the authors discussed how to increase the number of update points in a program by using induced program points approach. They have described the concept of safe region instead of safe points as described by its predecessor researches. All the threads need not to reach same point but it is enough to have all the threads in a safe region to perform update. They do not consider any optimization and de-optimization issues here.

Chapter 3

Validity and Safety of Updates

Researchers have done significant amount of work in developing on-line update systems. However, most of the literature is built on the basis of type-safety and they depend on programmers to specify the safe points for applying safe updates. These systems can not automatically validate the safety of the update.

3.1 Validity of Update

In [2] Gupta, Jalote and Barua built a formal framework for the on-line version change. Authors have defined validity of on-line version change. A program π can be dynamically updated to π' using state mapping S by first stopping the process P at time t in some state s, then applying the updates which replaces π by π' and mapping the state using state mapping S and after that continuing P from p. A state s of a program π is reachable if and only if a process executing π reaches s for some inputs. According to Gupta et al. to have a valid update, after a 'transition period' after on-line update applied in state s, new version program should reach a reachable state of π' from s and behave as if the new version was running from its initial state. State, initial state and the state mapping are defined by the program model used.

We feel that this definition of validity is too restrictive and of little use in practice. In general, updates are for fixing bugs, adding new features. So, the program behavior from old version to new version changes drastically. Therefore, It would be very rare to have new version reaching a state where we can surely say that this state would have been reached if the new version program was running from the initial state. Thus, most of the updates would be invalid according to this definition.

In [2] Gupta, Jalote and Barua have also shown that it is undecidable to determine whether an update is valid. Let us assume that there is an arbitrary program π and it is in state s. It is updated to a new version π'. s is reachable in π. s is mapped for new version program using state mapping S. Then they showed that it is not decidable whether the update is valid.

Giuffrida and Tanenbaum in [7] said that for this proof of undecidability in [2], researchers have neglected update safety and they have concentrated on type-safety. Many researches ignore update safety and others have specified that a function should not be updated while it is active.

3.2 Our Idea of Safety

In this report, we are going to define *update safety.* If a dynamic update is to be performed in a function which is active then the program is to be stopped at a *safe point* to apply the update. After updating the program, the execution is continued from that program point onwards. Expected behavior of the program after the update point is according to the new version of the program. However this notion differs from validity in that behavior may not be identical with the behavior of the new version of the program if the new version was running from the initial state. Before defining safety formally, we would like to mention a few points to motivate the main idea about safety.

1. No new version code should use any old version value which ought to be changed in the new version. If they use such value then the outcome of the program would not be according to the new version code and thus resulting in unsafe update.

   For example, let us consider the example in Figure 3.1. Updates in the program are - changing the condition in the control statement at S3, inserting a print statement for variable c at S10 and modifying the assignment statement of c at S11. If the update is applied at S12 then value of variable c which will be used in the new version would be according to the old program version V1. Printing of c at S18 or evaluation of conditional expression at S13 would be according to the old value of c which ought to be changed in the new program version V2. Thus, the update at S12 would be unsafe.

2. If a value is not changed by the update but the use of the value is changed then there is no harm in doing that update anywhere. Let us again consider the example shown in Figure 3.1. If the update is applied to the program at S2, then it is safe although the statement which uses variable c at S3 is modified. This situation is safe as the value of the variable c is not modified in the new program version.

3. There is a catch in the previous condition. If the change in the use of the value directly or transitively changes any control dependence then the update can not be applied safely to any point in the changed control dependence region.

   For example in Figure 3.1, let us consider the case where the update is applied at S4. However, the new version of the program would never execute S4 due to the change in the control statement at S3. Therefore, update at S4 would be unsafe. Here the modification of use statement of c at S3 directly changes the control dependence starting from S3.

   Now let us consider the situation when the update is applied at S14. Here the control dependence starting from S13 changes transitively due to the update in assignment statement at S11. Thus, update at S14 would not be safe as it can not be determined whether the new program version V2 would take the same branch at S13 as old program version V1.

3.3 Definition of Safety

We define update safety as follows:

Let us assume that π is the old version program and π' is the updated program. π' is obtained by applying update U on π. Update at program point p is safe iff one of the following holds:

```
 S1: c = 10              S1: c = 10
 S2: ...                 S2: ...
 S3: if (c>1){            S3: if (c<1){
 S4:   ...               S4:   ...
 S5: else                S5: else
 S6:   ...               S6:   ...
 S7: }                   S7: }
 S8: print c             S8: print c
 S9: c = 4               S9: c = 4
S10:                    S10: print c
S11: c = 5              S11: c = 10
S12: a = c              S12: a = c
S13: if (a<10){         S13: if (a<10){
S14:   ...              S14:   ...
S15: else               S15: else
S16:   ...              S16:   ...
S17: }                  S17: }
S18: print c,a          S18: print c,a

       V1                      V2
```

Figure 3.1: Example safety conditions

1. Statement at p is not updated.
2. No values are live at p in both π and π'.
3. Both the following are satisfied
   (a) Values live at p are produced only by "old statements" and used by statements who are control dependent on such statements whose control conditions or their values have not changed either directly or transitively.
   (b) Non-live values at p in π may become live at p in π' if and only if both the following conditions hold
      i. Statements in π that produced them are not modified/deleted by U
      ii. No new statement in π' produce them such that they would be live at p in π'

Let us again consider the example shown in Figure 3.1. Updates in the program are - modification of control statement at S3, insertion of a print statement at S10 and modification of S11. We will now consider different program points in the program and see whether they are safe with respect to the above definition.

- *Any program point before S1:* These points are safe according to the condition 2 of the definition.
- *Any program point after S18:* These points are safe as specified in condition 2 of the definition.
- *Point S2:* At S2, variable c is live in both V1 and V2. Hence, the only condition that is to be considered is 3.a. In the new program version V2, value of the variable c is produced by old statement S1 and the other part of the condition is satisfied trivially as it is used in S3 which is not control dependent on any statement. Hence, S2 is a safe point as condition 3.a is satisfied.
- *Point S4:* At S4, variable c is live in both the program versions V1 and V2. Therefore, the only condition to be considered is 3.a. Value of the variable c is produced by the old statement S1. However, S4 is control dependent on S3 and S3 is changed which results in change in control dependence. Thus, condition 3.a is not satisfied and it is not a safe point.
- *Any point Sk in between S9 and S10:* At Sk, variable c was not live in the previous version code V1 but live in the new program version V2 due to the insertion of print statement at S10. Thus, condition 3.b is to be considered. S9 produced the value of c and it is not updated in the new version V2 and there is no new statement inserted that produced the value of c which is live at this point. Thus, according to the condition 3.b all points in between S9 and S10 are safe.
- *At point S12:* At S12, c is live in both the versions V1 and V2. Therefore, condition 3.a is to considered only. Variable c is produced by S11 which is updated in the new program version V2. Hence, condition 3.a fails and S12 is an unsafe point.
- *At point S14:* At S14, variable a is live in both the versions so the only condition that is to be considered is condition 3.a. The value of a live at S14 is produced by the old statement S12. S14 is control dependent on the control statement at S13. Update at S11 transitively changes the control dependence starting from S13. Thus, condition 3.a is not satisfied for this point and S12 is an unsafe point.

Hence, the safe regions in the given code snippet in the Figure 3.1 are - code region before S1, S1 to S2, S8 to S10 and the code region after S18. In subsection 4.4.3 we discuss how to state safety by using the notion of dependence. That specification is used later in our implementation.

Chapter 4

Solution Design

Core of the problem is to find safe regions in a dynamically updatable program and while finding such regions one has to consider resulting optimizations and de-optimizations. First step is to find what are the total changes in the code due to updates, optimizations and de-optimizations. Safe or unsafe regions in the code can be derived from total code-change information. In the example [Figure 1.2], it was shown that total code-change is not only due to update but also due to resulting optimization. Therefore, now the problem is to determine whether an update enables any optimization or disables an already performed one.

Program optimization is a code transformation depending on some textual pattern matching. Optimization takes place in a code segment if some predefined conditions are satisfied and then optimization transforms the code using some predefined code change rules. These predefined conditions are called as *pre-conditions*. Code changes, that take place when pre-conditions are true, are called as *actions* of the optimization. Pre-conditions and actions for optimizations are provided through optimization specification as described later in chapter 5.

If pre-conditions of an optimization is satisfied in a code then the optimization is said to be enabled in the code and actions for that optimization is applied to the code. Information about the enabled optimizations are kept as depicted later in section 6.1 to check for disabling in later versions. If an old optimization is disabled i.e. its pre-conditions are no longer satisfied in the code, then code changes of that optimization should be reverted. Thus, total code change due to an update is due to update, optimization and de-optimization. Every code change has a corresponding dependency relation [section 4.1] change(s) in the program as described later in section 4.2. From the information of changes in dependency relations unsafe region can be determined as specified later in section 4.4.

4.1 Dependency relation structure

Each dependency relation has one start statement and one statement or more than one end statements. Dependence relation can be divided into two types - data dependence and control dependence.

Data dependence concept used here directly follows the idea in [8] and [9]. Data dependence can be of three types.

- Flow Dependence - In Figure 4.1, S1 assigns a value to variable a and S2 uses it.

```
S1: a = b+c
S2: f = a+20
```

Figure 4.1: Example of flow dependence

Here S2 is dependent on S1. This type of dependency is known as *flow dependency* or true dependency.

Flow dependence or true dependence can be defined as a dependence relation from S_i and S_j in program P such that S_j follows S_i and S_i sets some memory location which is used by S_j and no other statement in the path from S_i to S_j sets that memory location.

- Output dependence - In Figure 4.2, S1 and S2, both write to same memory location

```
S1: a = b+c
S2: a = 10
```

Figure 4.2: Example of output dependence

- variable *a*. Here, there is a *output dependence* from S1 to S2. Output dependence can be defined as a dependence relation from S_i to S_j where S_i , S_j both write to same memory location and there is no write operation to that memory location in the path between S_i and S_j.

- Anti-dependence -

```
S1: a = b+c
S2: b = 20
```

Figure 4.3: Example of anti-dependence

In Figure 4.3, S2 is an assignment statement of variable *b* and the statement S1 uses *b*. There is *anti-dependence* between S1 and S2 as S2 should not be performed before S1 in which case b would have wrong value.

If there are two statements S_i and S_j such that S_j follows S_i and S_j sets some memory location and S_i uses that memory location then there is an anti-dependence between S_i and S_j.

Control dependence - A statement S_j is said to be control dependent on S_i if execution of S_i determines whether S_j will be executed or not. In this example 4.4, whether S2 will

```
S1: if (b>0)
S2:   a = b+10;
S3: Else
S4:   a = c+10
```

Figure 4.4: Example of control dependence

be executed or S4 will be executed depends on the outcome of the condition statement S1. Here, both S2 and S4 are control dependent on S1.

4.2 Fundamental code changes and corresponding dependency relation change

Any code change in a program can be expressed in terms of two fundamental code change operations. Combining those fundamental code changes one can build any complex code change. These fundamental code change operations are -

1. *insert [insert (S,beforeS)]* - Statement *S* is added before the statement *beforeS* in the program.

2. *Delete [delete (S)]* - Delete statement *S*.

In this literature the word *change* is used to mean any of insert or delete.

Code change in a program involves some change in dependency relation (except when dead code is deleted or inserted). For instance, deleting a statement results in deleting all dependency relations in the program related to that statement. Deleting statement S results in deleting dependency relations starting from S to other statements ($otherS$) and ending at S. Each code change has an effect on dependence relations in the code segment as shown in Table: 4.1.

Action	**(Pattern) Code Modification**	**(Depend) Dependence Modification**
insert	insert (S,beforeS)	insert_dependence (type, S, otherS) insert_dependence (type, otherS, S)
delete	delete (S)	delete_dependence (type, otherS, S) delete_dependence (type, S, otherS)

Table 4.1: Mapping from code change to dependence relation change

4.3 Optimization and De-optimization of a program

At first let us define some symbols motivated by [3] to describe optimization properties and enabling and disabling conditions of an optimization. $\sim$ is a boolean binary operator which returns true if the second operand satisfies condition specified in the first operand.

For example $D \sim C =$ true means condition D is satisfied by the code segment C. Negation of $\sim$ operator is represented as $!\!\sim$. For example $D \; !\!\sim C$ represents condition D is not satisfied by code segment C.

Optimization O is expressed as a combination of $\{[O^{pre}]\}$, which is the set of pre-conditions and $\{[O^{Act}]\}$, which is the set of actions. If preconditions $\{[O^{pre}]\}$ are satisfied by code segment C then C is changed according to $\{[O^{Act}]\}$.

An update (U) changes the old version code segment C to give a new version of the code segment C'. Thus, $C' = C \bullet U$. Here $\bullet$ is a binary operator, which indicates inclusion of code changes.

Code change (C_Δ) is said to enable optimization O if C_Δ makes the pre-condition $\{[O^{pre}]\}$ satisfied which was otherwise not satisfied in C.

$$C_\Delta \text{ enables } O \text{ if } C' = C \bullet C_\Delta \text{ AND } [O^{pre}] \; !\!\sim C \text{ AND } [O^{pre}] \sim C'$$

Code change (C_Δ) is said to disable an optimization if the C_Δ makes the pre-conditions not satisfied which otherwise was satisfied.

$$C_\Delta \text{ disables } O \text{ if } C' = C \bullet C_\Delta \text{ AND } [O^{pre}] \sim C \text{ AND } [O^{pre}] \; !\!\sim C'$$

If pre-conditions of the optimization O is satisfied in the code C then action for the optimization $[O^{Act}]$ is to be applied on C to have the changed code C'.

$$C' = C \bullet O^{Act} \text{ where } O^{pre} \sim C$$

Action part of any optimization can be specified using three types of code changes - insert some save statements, delete some statements and insert some statements. We would call these - *save statements, delete statements and insert statements.* In the following example of common sub-expression elimination we explain these types.

Here in this example [Figure 4.5], S1 and S2 has a common sub-expression - a+b. Therefore, after optimization the code change happens and the changed code is also shown in the Figure 4.5. S0 is the save statement which saves the value of the common sub-expression in a temporary variable t. S1 and S2 statements are delete statements and

```
    S1: x = a+b                    S0: t = a+b
    S2: y = a+b                    S1': x = t
    S3: print x, y                 S2': y = t
                                   S3: print x,y

  Un-optimized code                 Optimized code
```

Figure 4.5: Types of code change for an Optimization

S1' and S2' are insert statements. In many optimizations there is no save statement or insert statement. In that case, these sets would be null.

An update or a de-optimization may disable some previously performed optimizations. To check for disabling, information about previous optimizations is needed which can be represented as the set of save, delete and insert statements of previously performed optimizations. This information is saved as old optimization information file where each line describes one previously performed optimization. Details about the file is described later in section 6.1. Disabling of a previous optimization (O) can be done by checking whether the pre-conditions for the optimization, (O) on the basis of save, delete and insert statements given by the old optimization information, are still satisfied in the code. If this check finds that the code does not any longer satisfies the pre-condition of the optimization then de-optimization should be performed. To undo the optimization effect, save and insert statements should be deleted and delete statements should be inserted.

Suppose in the code pre-condition for optimization O was satisfied and it was optimized to code C after application of the code changes for O. Now a code change C_Δ takes place in the code and $[O^{pre}]$ is not satisfied any more. Code change to de-optimize O is represented as $O^{UndoAct}$. Therefore, de-optimized code C' would be -

$$C' = C \bullet O^{UndoAct} \text{ where } O^{pre} \sim C \text{ AND } O^{pre}! \sim (C \bullet C_\Delta)$$

Previous optimizations are checked for disabling in the order of their happenings. E.g. if application of O_1 happens before O_2 then checking for disabling of O_1 would be performed before O_2.

After all the de-optimization operations are performed in the code; optimization enabling is checked. If the optimization pre-condition is satisfied in the code then the optimization action is applied to the code to have the optimized code. Optimizations are checked for enabling in a pre-specified order. We assume the order is given by the system designer in the specification of optimizations is optimal.

4.4 Determination of Unsafe Region

Update changes the input code. These code changes bring about changes in relevant dependence relations in the program. Data dependency relation changes can happen in the form of insertion or deletion of flow dependence, anti-dependence or output dependence. Control dependency relations can also be affected. Change in dependence relation may undo some previously performed optimizations and may enable some new optimization opportunities. All these code changes make corresponding changes in dependency relation structure of the program. If a program in updated while it is executing a statement S that is included in the domain of a changed dependence relation, applying the update at S can lead to wrong results in the program. Hence we should avoid updating a program at such statements. For this purpose, we determine an unsafe region in the program with respect to an update U. We discuss this issue in this section.

4.4.1 Code changes to be considered in Unsafe Region determination

To analyze all possible code changes that can change a dependency relation consider Figure 4.6.

There is a data dependency in between program point p1 and p2 such that p2 follows p1. For this data dependency, p1 is the starting point and p2 is the ending point. Program point p1 and p2 have divided the whole program into three regions - R1, R2 and R3.

- Region R1: Set of program points that precedes p1 in the control flow of the program.
- Region R3: Set of program points that follows p2 in the control flow of the program.

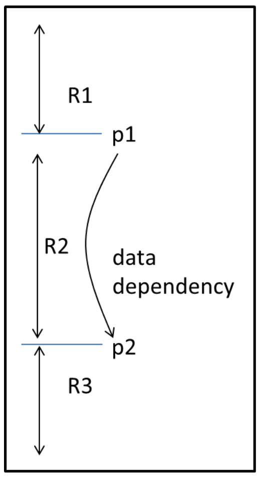


Figure 4.6: Possible code change region to change a dependency relation

- Region R2: Set of program points that follows p1 but precedes p2.

As long as pointer arithmetic is not considered, trivially it can be said that dependency relation in between p1 and p2 cannot be affected due to any code change in either R1 or in R3 or in both. However, code changes in the region R2 or changes in either p1 or p2 may affect the dependency relation. Therefore, the following kinds of changes can affect the existing dependence relations -

1. Change in the starting point of the dependence.
2. Change in the ending point of the dependence.
3. Change in some statements in between the starting point and the ending point of the dependence.

As a result of these code changes dependency relations change according to Table 4.1. Changes in dependency relation can be insertion or deletion of dependence relations. For example, let us consider again Figure 1.1. In program version V1, there was a flow dependence from S4 to S6 for variable c. In the new program version V2, assignment statement S4 is modified, so the flow dependence from S4 to S6 changes. If the update is applied at a program point in the region from S4 to S6 e.g. at S5, then it will be an unsafe update as S6 would then print the value of c according to V1. Hence, it can be argued that a dependence region, which is modified is to be considered for deciding whether it is an unsafe region or not.

4.4.2 Irrelevance of anti & output dependence in Safe Region determination

Update and resulting de-optimization and optimization creates many code changes. These code changes can be mapped to dependency relation changes as shown in Table 4.1. However, any change in anti-dependence or in output dependence is not relevant to determine unsafe region.

```
S1: x = a
S2: a = ...
```

Figure 4.7: Anti-dependence irrelevance in determination of safe region

All the programs considered in this report are single threaded, so the control flow is sequential in all the cases. Figure 4.7 shows a classic situation of anti-dependence where S1 is anti-dependent on S2. The following kinds of changes can insert a new anti-dependence between S1 and S2.

1. Add or modify S1 (starting statement of the dependence)
2. Add or modify S2 (ending statement of the dependence)
3. delete or modify relevant statements that follow S1 and precedes S2 in the control flow

For code changes as in case (1) and (2) -

- Anti-dependence insertion here has nothing to do with the determination of safety of any point before S1 because it is not included in the domain of the anti-dependence.
- Anti-dependence between S1 and S2 is with respect to the variable a. If the update is performed at a program point S_i that follows S1 and precedes S2 then the value of the variable a at S_i is same according to both new version and old version code. This is because assignment statement of a (S2) is after S_i. Therefore, region in between S1 and S2 is safe with respect to anti-dependence change.
- Anti-dependence insertion has no effect in determining if any point after S2 is safe because it is not included in the domain of the anti-dependence.

For code changes at S1 and S2, there may be some side effects like change or introduction of define and use variables at S1 and S2. These changes may introduce unsafe regions due to relevant changes in dependences but these are independent of anti-dependence introduction.

For code changes as in case (3) - Let statement S_k be deleted or modified where S_k follows S1 and precedes S2 in the old version of the code. Deletion or modification of S_k inserts anti-dependence between S1 and S2.

- Code region before S1 is safe as S1 precedes S_k in control flow. This is true for any statement that precedes S_k.
- Any statement that follows S_k is safe to update with respect to anti-dependence deletion. S_k may change some flow dependence between S_k and S_j where S_j is any statement that follows S_k and uses the left hand side variable of S_k. These changes give unsafe region from S_k to S_j. However, insertion of anti-dependence is not responsible for this unsafe region creation.

So, as a result of insertion of the anti-dependence there cannot be any unsafe region. In the same way, if any anti-dependence is deleted then there is no unsafe region due to change in anti-dependence. Therefore, information of anti-dependence change is irrelevant for determining unsafe region.

```
S1: a =4;              S1: a =4;
S2: if (c>10)          S2: if (c>10)
S3:    ...             S3:    a = 6;
S4: else               S4: else
S5:    ...             S5:    ...
S6: c = a + 14;        S6: c = a + 14;
S7: print c,a          S7: print c,a

    V1                      V2
```

Figure 4.8: Output-dependence irrelevance in determination of safe region

Now consider the case of output dependence. In Figure 4.8, V1 is the previous version of the code and V2 is the new version of the code. Update is to insert an assignment statement S3. Due to this update, output dependence in between S1 and S3 is inserted. Let us examine the different relevant code regions with respect to the newly added output dependence.

- Program region before statement S1 is not relevant to the determination of unsafe region as statements in this region is not included in the domain of output dependence region from S1 to S3.
- If the update is applied at a program point in between S1 and S2 then it will be safe as the value of the variable a is according to a old statement(S1) at this point. Therefore, the statements in the region in between the start and end statement of the changed output dependence is safe.
- If the update is applied at a program point after S2 then it is irrelevant with respect to the change in output dependency as that region is not included in the domain of output dependency.

Therefore, the output dependency change between S1 and S3 is irrelevant for determining unsafe region. Hence we conclude that determination of unsafe region in a program is independent of changes in both anti-dependence and output-dependence.

4.4.3 Algorithm to find Unsafe Region

Among all types of data dependences, only flow dependence is relevant while determining safe region. Along with it, control dependence changes would have to be considered. Changes in flow dependence can be of two types - insertion or deletion. Changes in flow dependence can result from changes in starting statement, ending statement or any other relevant statement in between starting and ending statement. To exhaustively consider all cases, all possible combination are considered here -

1. *Flow dependence insertion* -

   (a) *Changes in the starting point* - If an assignment statement is added or modified and due to that, there is a new flow dependency introduced with the changed statement as start statement of the new dependence, then the newly added flow dependency region (the region in between the starting statement and the ending statement of the dependence relation) is an unsafe region.

```
S1: a = ...          S1: a = ...
S2: ...              S2: a = ...
S3: S3: print a      S3: S3: print a

     V1                   V2
```

Figure 4.9: Flow dependence insertion due to change in starting statement

In Figure 4.9, V1 is the previous version and V2 is the new version. Update in V2 is to add S2. There is a new flow dependence region created from S2 to S3 due to this update. Therefore, unsafe region is from S2 to S3.

(b) *Changes in the ending point* - If there is a modification of an operand of the use statement or insertion of a new use statement then the expression changes so lhs value of the expression changes. Therefore, flow dependences starting from this statement give unsafe regions. Condition 1.a already handles this situation.

(c) *Changes in the code in between starting and ending point* - Relevant code changes in the region in between start and end point can introduce new flow dependence.

In Figure 4.10, previous version is V1 and newer version is V2. Update is

```
S1: b = 5             S1: b = 5
S2: c = b + 23        S2: c = b + 23
S3: b = 3             S3:
S4: c = b + 10        S4: c = b + 10
S5: print c           S5: print c

      V1                    V2
```

Figure 4.10: Flow dependence insert for code change in between start & end statement

to delete S3. S3 is in between an assignment statement (S1) of *b* and an use statement [1] (S4) of *b*. Because of the update, S1 to S4 flow dependency is introduced. In the process flow dependence between S3 and S4 is deleted. This deletion is due to deletion of start statement. This situation can be taken care

[1]Use statement for a variable is a statement where the variable is read i.e. the variable is used in that statement.

of by condition 2. So, flow dependency insertion due to change in code in between start statement and end statement of the dependency is not relevant to determine unsafe region.

2. *Flow Dependence Deletion* - Flow dependence can be deleted for changes in start statement or end statement of the dependence relation or any relevant changes in the code in between start statement and the end statement.

   (a) *Changes in the starting point of the dependence* - If a modification or deletion of the start statement of a flow dependence deletes the flow dependence relation, then the erstwhile flow dependence region is unsafe for that update.

   (b) *Changes in the end point of the dependence* - If changes in the use statement deletes flow dependence,then there is no unsafe region for this deletion, although new or modified flow dependences starting from these statements may give unsafe region but it can be handled by other conditions.

   (c) *Changes in the code in between the starting point and the end point* - If some flow dependences are deleted or inserted for this type of changes then those new flow dependences whether inserted or deleted would introduce unsafe region as appropriate but they can be handled by other conditions.

3. *Control Dependence change* - If a control statement (S) is modified then the statements that were control dependent on S in the old version code gives unsafe points. In Figure 4.11, S4 and S6 are control dependent on S3. In this example, S2 is deleted

```
S1: a = 10          S1: a = 10
S2: a = 20          S2:
S3: if (a>15)       S3: if (a>15)
S4:   ...           S4:   ...
S5: Else            S5: Else
S6:   ...           S6:   ...
     V1                  V2
```

Figure 4.11: Control dependence change

to get V2 from V1. For this deletion flow-dependence from S2 to S3 is deleted. So, program region from S2 to S3 is an unsafe region according to condition 2.a. Now, S4 and S6 are also unsafe points as the control dependence is changed. Thus, while considering control dependence effect on the determination of unsafe region, a program point should be a unsafe point if it is control dependent on an unsafe point.

4.5 Overview of the Scheme

To see the whole system scheme, let us put all the parts together. Previous version of the code is fed to the system. Then it is updated using the update specification specified by the programmer using insert and delete as described in section 4.2. Then using the old optimization information and optimization specification as presented later in chapter 5, it is decided whether any previously performed optimization is disabled. If any optimization is disabled then de-optimization is performed as described in section 4.3. After that it is checked whether there is any opportunity of new optimization and on such opportunity optimization is applied to the code as mentioned in section 4.3. After this the updated and optimized code is obtained. Then the dependence relations in the final code and the dependence relations before update is compared to get the changes in the dependence relation. This information is then mapped to the unsafe region according to the algorithm in subsection 4.4.3. The scheme is depicted in Figure 4.12.

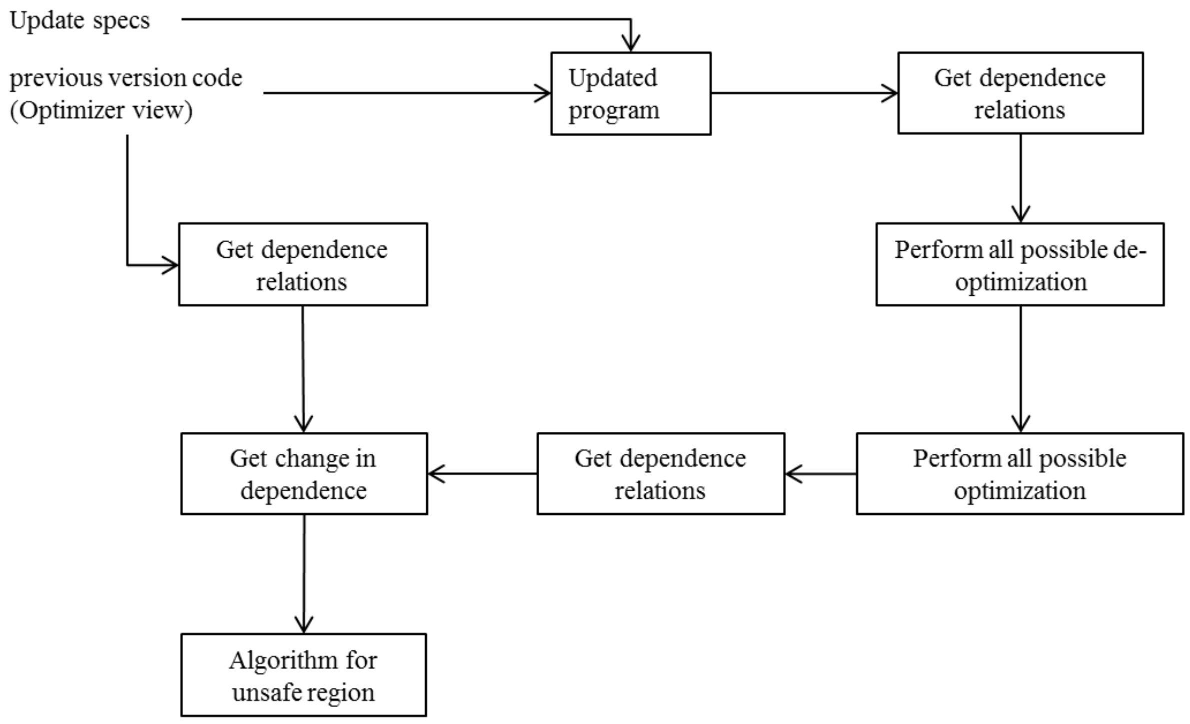

Update specs
previous version code (Optimizer view)
Updated program
Get dependence relations
Perform all possible de-optimization
Perform all possible optimization
Get dependence relations
Get change in dependence
Get dependence relations
Algorithm for unsafe region

Figure 4.12: Overview of the scheme

Chapter 5

Optimization Specification Language

Program optimization is a transformation of a code segment which is applied if the program code segment satisfies pre-condition of that optimization. Optimizations can be specified using specification languages describing the pre-condition and post-condition. *SpeLO* (Specification Language for Optimization)[3] specifies optimization in terms of pre-condition and action part. SpeLO is developed based on Gospel specification language [10][11]. If pre-condition is satisfied then action is to be applied in the input code. General specification format in this language is -

```
Optimization name
PRECONDITION
  Code Pattern
    [Quantifier ElementId[,pos]: Format_of_elements]+
  Depend
    [Quantifier ElementId[,pos]: Sets_of_elements, Dependence_conditions]+
ACTION
  [code_change]+
```

Figure 5.1: General format of *SpeLO* specification

In this language, to represent a statement, the lhs is called as operand1 or in short opr1. For a binary expression, two of its operands are called as opr2 and opr3. And the operator of the expression is represented as opcode.

1. *Pre-condition* is defined in two parts - code pattern condition part and dependency condition part.

(a) *Code pattern condition* - This gives a generic code structure that must be satisfied in the input code to perform the optimization. This has essentially the form -

```
Quantifier element_list: Format_of_elements
```

element_list is typically a set of statements. This set is chosen on the basis of the format conditions and the quantifier specified. Quantifier can be one of ANY, ALL or NO.

- ALL - Returns all statements where each of the statement satisfies the conditions specified.
- ANY - Returns a statement that satisfies the conditions specified. For example the condition.

  any S_i : S_i.opcode = copy AND type(S_i.operand1) = variable

  returns any copy statement that has right hand side as variable.
- NO - returns null set if the condition is satisfied.

Format_of_elements describes the format of the type of elements required. Usually, this part restricts the statement's operands and operator.

(b) *Depend conditions* - This gives the dependency conditions that are to be satisfied. This is of the form

Quantifier element,[pos]: element_list, dependence condition list;

Quantifier can be one of ANY, ALL or NO as in the code pattern condition part. In the process of specifying the dependence conditions, some other elements are also needed and they are specified in the form of sets_of_elements. It is used to check membership of an Element in a Set. For example mem(S, path (S_m, S_n)) checks whether S is a member of the path from S_m to S_n. Dependence conditions are specified in the form -

```
Type_of_dependence (S_start,S_end)
```

Type_of_dependence can be any of flow dependence, anti-dependence, output dependence or control dependence. S_start, S_end are the starting and ending statements of the dependence. While specifying a data dependence, one of the arguments in the function denotes the variable for which that dependence is introduced in the program. For example, in case of flow_dep(...), the first argument in the function specifies the variable for which flow dependence is introduced. Similarly, for anti-dependence specification, second argument in the function anti_dep(...) denotes the variable for which the anti-dependence is introduced.

Pos tag denotes the position of the relevant operand in the expression. This is an optional field.

Operand (S,pos) returns the operand in the statement at the position denoted by pos.

For example the dependence condition

$$\text{All } S_j : \text{flow\_dep } (S_i, S_j)$$

returns a set of all statements which are flow dependent on statement S_i.

2. *Action* - If the input code satisfies the preconditions of the optimization then there are some code changes performed. These code changes are specified in the action part of the specification of the optimization. There are four primitive code changes - *Insert, Delete, Move and Modify* of a statement. All code changes can be reduced to insert and delete as mentioned in section 4.2 Action part is described combining these code changes. Code changes have the following format -

    - *Insert* - Insert contains two arguments. The first one specifies the statement to be inserted with opcode, lhs operand and rhs operands. And the second argument specifies the statement before which the new statement is to be inserted.

    ```
    Insert (stmt_to_insert, beforeS)
    stmt_to_insert (opcode, lhs variable, rhs operands)
    ```

    - *Delete* - only specifies the statement to delete

    ```
    Delete (stmt_to_delete)
    ```

    - *Move* - Specifies movement of stmt1 before stmt2.

    ```
    Move (stmt1, stmt2)
    ```

    - *Modify* - stmt is modified according to the second argument in the function.

```
Modify (stmt, stmt_to_insert)
stmt_to_insert (opcode, lhs variable, rhs operands)
```

5.1 Examples

5.1.1 Copy Propagation

Example of copy propagation - Copy propagation optimization in SpeLO specification is -

```
COPY_PROPAGATION
PRECONDITION
   Code_Pattern :
      ANY Si: Si.opcode = copy AND type (Si.operand1) = variable
   Depend:
      ALL Sj,pos: flow_dep (Si,Sj);
      NO Sk: flow_dep (Sk,Sj) AND (Sk!=Si);
      NO Sp: member (Sp, path(Si,Sj)), anti_dep (Si,Sp);
ACTION
   Modify (Operand (Sj,pos), Si.operand1);
   Delete (Si);
```

Figure 5.2: Copy propagation optimization specification in SpeLO

- Code pattern condition part specifies that there should be some copy statements where right hand side of the statements should essentially be variables.
- Dependence condition part specifies about the dependency constraints to be satisfied.
  - There should be a flow dependence starting from the copy statement (S_i) to other statements (S_j).
  - There should not be any other flow dependency reaching (S_j).
  - There should not be any re-definition of the operand1 of S_i in the path from S_i to the S_j.

Action part describes the code changes that are to be performed in the input code if pre-condition is satisfied. Operands in (S_j) are to be modified and the copy statement (S_i) is to be deleted.

5.1.2 Dead Code Elimination

Dead code elimination can be specified as depicted in Figure 5.3.

```
DEAD CODE ELIMINATION
PRECONDITION
   Code_Pattern :
      ANY Si: Si.opcode = assign OR Si.opcode = - OR
              Si.opcode = + OR Si.opcode = * OR
              Si.opcode = /
   Depend:
      NO Sj: flow_dep(Si,Sj)
ACTION
   Delete (Si);
```

Figure 5.3: Dead code elimination specification in SpeLO

5.1.3 Common Sub-expression Elimination

Common sub-expression elimination can be specified in SpeLO as described in Figure 5.4.

```
Common Sub-expression Elimination
PRECONDITION
   Code_Pattern :
      ANY Sn: Sn.opcode = + OR Sn.opcode = - OR
              Sn.opcode = * OR Sn.opcode = /
   Depend:
      Any Sm: Sm.opcode = Sn.opcode AND
              Sm.opr2 = Sn.opr2 AND Sm.opr3 = Sn.opr3
      No (Sk,pos): anti_dep(Sn,operand(Sk,pos))
                   AND flow_dep (operand(Sk,pos),Sm)
      All Sj: ctrl_dep(Sj,Sn) AND ctrl_dep(Sj,Sm)
ACTION
   Insert ((Sn.opcode,temp,Sn.opr2,Sn.opr3),Sn)
   Modify (Sn, (assign, Sn.opr1, temp)
   Modify (Sn, (assign, Sm.opr1, temp)
```

Figure 5.4: Common Sub-expression Elimination in SpeLO

Chapter 6

System Design

The system is to determine the safe regions for a dynamically updatable program for an update considering optimizations and de-optimizations. The overview of the scheme was included in section 4.5.

6.1 Maintaining information of different versions

If an optimization *O* is disabled which was performed in some previous version, then the code changed due to *O* is to be reverted e.g., consider the example shown in the [Figure 6.1].

```
S1: x = 2           S1: ......            S1: x = 2
S2: x = 3           S2: x = 3             S1': print x
S3: print x         S3: print x           S2: x = 3
                                          S3: print x
  Unoptimized         Optimized
               V1                                V2

               (a)                               (b)
```

Figure 6.1: From Left: Unoptimized V1, Optimized V1 (DCE) and V2

Here the update is to insert a use of x in the form of print at S1' or before S2. In V1, assignment to x in S1 was removed as it was dead. However in V2 this definition should be brought back as there is a use for x defined in S1' and that definition is no longer dead.

To undo the dead code elimination optimization performed in V1, definition of x at S1 should be inserted. However, we have to decide where to insert the assignment statement. If in between V1 and V2 there are many code changes due to updates, optimizations and de-optimizations then the place of insertion can not be specified by source line numbers. Source line number is not constant for any statement. If statement S has line number l in the old version of the code and there is an update of insert statement at line number i where i<l then line number of S in the newer version would be l+1. Conversely, if line number of statement S in old version is l and the code is modified by deleting the statement at line number i where i<l then, in the updated program version line number of statement S is l-1. Thus, line numbers of the statements change with code changes. Hence, we mark every statement with a *unique id* number that remains constant for that statement even if the statement is deleted by an optimization or its line number is changed. Previous optimization information and de-optimization code changes are specified using unique ids.

To check whether a previously performed optimization is disabled or not there should be some information about the previously performed optimization. As it was seen in section 4.3 every optimization can be represented with its code changes - save statements, delete statements and insert statements. Moreover, many optimizations are considered in the system, thus every such information should be marked with the optimization performed. Therefore, every optimization is given a unique id and with this each old optimization is tagged. This information about previously performed optimizations is stored in a file called old optimization information. Each line in this file has information about one optimization performed. Here save, delete, insert statements are represented as a comma separated list of unique ids. Format for this file is shown in Figure 6.2.

```
<optimization id>$<save stmts>$<delete stmts>$<insert stmts>$
```

Figure 6.2: Format of old optimization information

6.2 Different Program Views

In the system, three different levels of abstractions are to be maintained and for doing that three views of the same program is maintained. Views of a program that are maintained in the system are -

1. *Programmers' view* - This view is the code for programmers who would specify the updates in the code. Programmers would not know that there would be optimizations and de-optimizations inside the compiler. Therefore, the update specification would be according to the line numbers in this view. Programmers' view is only changed by the updates, it is not affected by the optimizations and de-optimizations.

2. *Logical program view* - After a series of updates, de-optimizations and optimizations, to check whether a previously performed optimization is disabled and if it is disabled then to determine the code changes, the history of changes in the code are to be kept. Logical view of the program has all the statements which are currently existing in the program and the statements that are deleted by optimization. Each statement has a unique id associated with it as mentioned in section 6.1 and a flag which denotes the state of the statement. State of a statement can be active in the current code, deleted by an optimization etc. In the logical program view, each line has a statement. There are three fields - unique id of the statement, flag representing the state of the source line in the program. Format of the logical program view is depicted in Figure 6.3.

   ```
   <unique id>$<flag>$<source line>$
   ```

   Figure 6.3: Logical program view format

   Flags are -

   - 'A' - Statement is active in the currently running program
   - 'S' - Statement inserted by optimization.
   - 'D' - Statement is deleted by an optimization. It is not present in the current running program.
   - 'U' - Statement is deleted by an update or a de-optimization. This statement can not be reverted back in the current running program.

   Updates and optimization and the de-optimization changes are reflected in the logical program view.

3. *Optimizer view* - Optimizer optimizes and de-optimizes the program. Optimizer view is the program which is currently running in the system.

Optimizer view includes the 'A' and 'S' flagged statements in the logical program view. Therefore, mapping between unique ids in logical program view and line numbers in optimizer view is quite trivial.

We use LLVM (Low Level Virtual Machine)[12] to implement our system, and the intermediate language generated by LLVM (LLVM IR) is used as the program input. Thus, LLVM IR can also be considered as another view used here. However, this is only for implementation purpose.

6.3 System Schematic

The system first applies the update specified as a system input, on the previous version of the code. After that it is checked whether there is any previously performed optimization that should be disabled. Information about the previous optimizations come from old optimization information. If a previous optimization is is to be disabled, code changes to apply for de-optimization are determined from old optimization information and the logical program view. With the code change, old optimization information is also to be modified by deleting the information of the disabled optimization. When all possible de-optimizations are done then it is checked if a new optimization is enabled. For this checking, pre-condition part of the optimization specification is used. If an optimization is enabled, then code changes are done according to the action part of the optimization specification. Code changes done are also recorded in the old optimization information. For code changes in updates, de-optimizations and optimizations, all three program views are to be changed. However, for de-optimizations and optimizations, programmers' view is not be changed.

Whenever there is a need of code change either for update or for optimization or for de-optimization, same things are to be done - modifying three program views and the old optimization information(as per needed). View sync functionality in the system does all these code changes - synchronizes all program views. After having the updated, optimized and de-optimized program, dependence relations which are inserted and deleted are considered to determine the unsafe region in the program for that update. Whole system schematic is depicted in Figure 6.4.

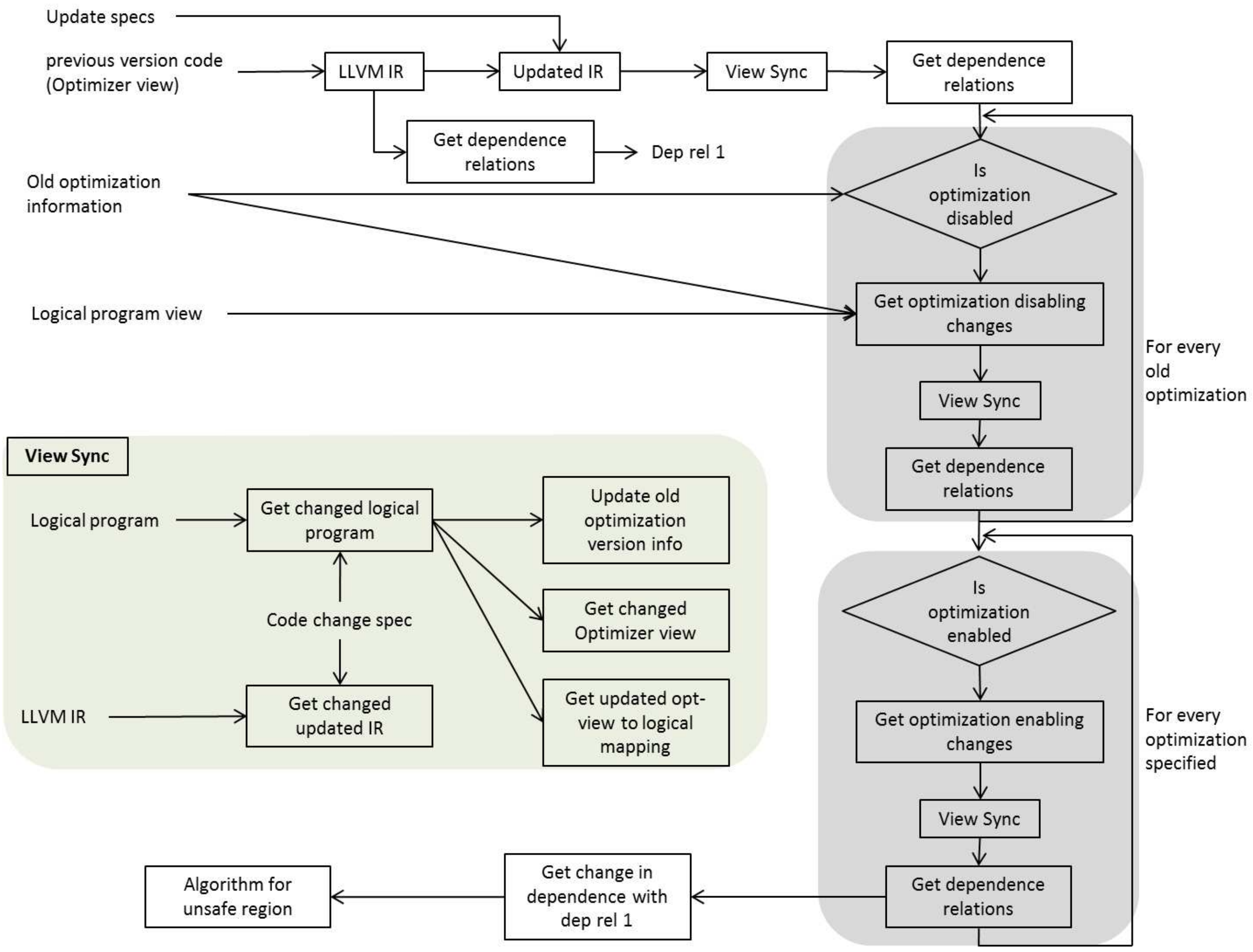

Update specs
previous version code (Optimizer view)
LLVM IR
Updated IR
View Sync
Get dependence relations
Get dependence relations
Dep rel 1
Old optimization information
Is optimization disabled
Logical program view
Get optimization disabling changes
View Sync
Get dependence relations
For every old optimization
View Sync
Logical program
Get changed logical program
Update old optimization version info
Code change spec
Get changed Optimizer view
LLVM IR
Get changed updated IR
Get updated opt-view to logical mapping
Is optimization enabled
Get optimization enabling changes
For every optimization specified
View Sync
Get dependence relations
Get change in dependence with dep rel 1
Algorithm for unsafe region

Figure 6.4: Overview of the scheme

Chapter 7

Implementation

The system described in section 6.3 is implemented in LLVM 2.8 (Low Level Virtual Machine) [12]. LLVM is a compiler which, provides source and target independent optimizer and code generation facility. LLVM generates an input language independent intermediate representation (LLVM IR) for every input program. Therefore, the implemented system can be made source language independent. All the core functionality in LLVM are well documented [13].

As mentioned in section 6.2 three views of a program are maintained in the system - programmers' view, optimizer view and logical program view. All these views are stored in separate files and whenever they are accessed then file read operation is used. We have loaded the program in LLVM and used unpotimized intermediate code generated by LLVM.

The system is implemented as a module pass in LLVM. It uses unpotimized form of LLVM IR with optimization level 0.

7.1 Data structures used

1. To store *dependency relation* - Structure is described in Figure 7.1.
   (a) source_stmt [type stmt.] - Dependency relation begins from this statement in the code
   (b) dep_type [type string] - Denotes type of the dependency relation.

```
<source_stmt, <dep_type, {dest_stmt}> > dep_rel
```

Figure 7.1: Data structure to store dependence relation

i. flow_dep - Flow dependence

ii. anti_dep - Anti dependence

iii. output_dep - Output dependence

iv. ctrl_dep - Control dependence

(c) dest_stmt [type stmt] - Dependency relation ends in this statement.

Although the system currently does not support control dependence.

2. To store *Dependency relation change* - This data structure is needed to store the changes in the dependency relation and apply the unsafe finding algorithm as described in subsection 4.4.3. Structure is described in Figure 7.2.
   change_type [type string] - Denotes the type of change of the dependency relation

```
<change_type, {dep_rel}> dep_rel_change
```

Figure 7.2: Data structure to store dependence relation changes

-

(a) add - Insert a new dependency relation

(b) delete - Delete an old dependency relation

3. *map_opt_view_to_logical* - This is a vector to store the mapping between the line numbers in the optimizer view and the unique ids of the statements.

4. *Expression class* - To store the operands, operators and the type of the instructions *Expression class* is created. LLVM converts a source statement into a set of instructions. For example, in Figure 7.3 LLVM IR for the statement $c = a + b$ is shown. In first two statements I1 and I2, operands a and b are loaded in temporary variables %0 and %1 respectively by LoadInst in LLVM. Then %0 and %1 are added by the

binary operator instruction[1] at I3 and result is stored in another temporary variable %2. In the next statement I4, the temporary variable %2 is stored in variable c. Now, from the binary operator instruction I3 we know the operator as addition. Actual operands are known from the load instructions in I1 and I2. From the store instruction in S4 we know the lhs of the statement. We store the operands, operator and lhs operand in the fields of Expression class instances. Here, we have described how a binary operator instruction is represented in LLVM IR. There are many types of instructions in LLVM like call instruction, comparison instruction, casting instruction etc.

```
I1: %0 = load i32* %a, align 4, !dbg !11
I2: %1 = load i32* %b, align 4, !dbg !11
I3: %2 = add nsw i32 %0, %1, !dbg !11
I4: store i32 %2, i32* %c, align 4, !dbg !11
```

Figure 7.3: LLVM IR corresponding to c = a + b

5. *ExpressionSet class* - To store a set of Expression objects. This class has methods for inserting, deleting and finding expressions from the Expression set. Expression related classes and the methods are implemented in Expression.h file.

7.2 Relevant functions

There are five types of functions that are defined in the system -

1. Utility Functions [2] - These functions are used as utility tools.
2. Functions for maintaining dependence relation - These methods do things related to dependence relations specially constructing dependence relations in the program.
3. Functions for view sync - This type of functions change the views of the program according to the code change specification provided.

[1]Binary operator instruction is an instruction with two operands. It is represented as BinaryOperator class in LLVM

[2]Here 'function' is used in the meaning of functionality. Actually all of these are methods as they are defined in C++ class. In this literature, the word 'function' and 'method' are used interchangeably

4. Functions related to optimization and de-optimization - These functions are built to check for optimization, de-optimization and make the code changes if necessary.

5. Functions to find the unsafe region in the program.

7.2.1 Utility functions

1. *construct_varset* - This method returns the set of local variables of a function passed as argument. It traverses over each instruction in the function and checks if the statement is an allocation instruction. If it is an allocation instruction, then the value declared is the variable declared.

2. *Display functions* - These functions are for printing the program views, mapping between line numbers in the optimizer view and the unique ids of the statements and also LLVM IR. There is also a DisplayAll function which calls all other display functions and prints all of them.

3. *find_line_number* - This function finds the line number in the optimizer view for a statement whose unique id is passed to the function.

Methods in Expression class find the operands from different types of instruction in LLVM IR using *createOP* function. LLVM loads each operand of a statement in unique temporary variables and then use those temporary variables. Algorithm for implementing this is described in algorithm 1.

Algorithm 1 Find the operands in a statement

```
1: Input: Value pointer to the temporary variable($v$) used in the instruction
2: Output: Value pointer to the actual variable used
3: loop
4:     if $v$ comes from a Load Instruction then
5:         Continue search in the load instruction which has loaded $v$
6:     else if $v$ is a constant then
7:         This operand is a constant
8:         Break the loop and return the constant
9:     else
10:        Break the loop and return the current load instruction being considered
11:    end if
12: end loop
```

Expression class has another field for storing the line number of the instruction and this class also has a provision of setting the line number of an instruction.

- *get_source_line_number* - This function returns the line number for the instruction passed to the function. It uses metadata or more specifically *DebugLoc* class to get the debug information about the instruction.
- *set_source_line_number* - This function sets the line number of an instruction as specified in the argument. It calls a function in *DebugLoc* class named as setLine to set the line number of the given instruction. This setLine function is added to LLVM by us. It simply sets the line number variable maintained by LLVM.

7.2.2 Functions for maintaining dependence relation

These functions find the dependence relations in a given function. Finding flow, output and anti dependence is done almost same way. First a store instruction is passed to the function to find any data dependence. From this instruction it recursively checks all paths to the end or to the next definition of the same variable to find use of the variable as necessary. The algorithm for finding flow dependency is described in algorithm 2.

Algorithm 2 Find flow dependences in the function

```
1: Input: A store Instruction ($SourceI$) of the variable $v$
2: Output: Flow dependences starting from $SourceI$ for $v$
3: for Each Instruction($CurrentI$) in the control flow graph following $SourceI$ do
4:    if $CurrentI$ is a store instruction of $v$ then
5:       return null
6:    else if $CurrentI$ has a use of $v$ then
7:       There is a flow dependence between $CurrentI$ and $SourceI$
8:    end if
9:    for Each successor of the current Basic Block which is not visited do
10:      Call itself recursively this function for successor basic block
11:   end for
12: end for
```

Methods for finding anti-dependence and output dependence works in a very similar way.

7.2.3 Functions for view sync

These functions change code as per the code change specification is given.

1. *View_Synchronize* - This function changes the code of different program views as necessary and also modifies old optimization information. Code change is specified in a file. Main working of this function is explained in algorithm 3. Also there is an example in section 7.3 explaining these view changes.

Algorithm 3 View_Synchronize

```
1: Input: Name of the file where the code change is specified (code_change_file), a flag
   whether its a update or optimization change
2: Output: Changes all the views and LLVM IR - synchronizes all.
3: for Each line in code_change_file do
4:     Call a function to change the LLVMIR
5:     Call a function to change the logical program
6: end for
7: Call a function to change Optimizer view
8: Call a function to change the line number information in the LLVMIR metadata
9: Call a function to change the mapping map_opt_view_to_logical
10: if This code change is optimization or de-optimization change then
11:     Call a function to modify old optimization information
12: end if
```

2. *chanegeLLVMIR* - This function changes LLVM IR according to a specified code change. Code change can be insert a line or delete a line. Its working is described in algorithm 4. Current implementation only allows insertion of statements with '=' i.e. assignment statements.

Algorithm 4 Change LLVM IR

```
1: Input: A code change
2: Output: Modified LLVM IR
3: Traverse through the IR and find the instruction($I$) to change
4: if code change is delete then
5:     Delete the instruction from IR. Delete all instructions with same line number
6: else if code change is insert or save then
7:     Get the lhs from the source line specified in the code change line
8:     Call a function to get the post-fix expression of the rhs of the source line in the
       code change line
9:     Call a function to get the LLVM instruction for each binary operation and add to
       IR
10:    For each inserted line set the line number as BASE+offset. BASE - very large
       integer and offset - line number of this statement in changed code
11: end if
```

3. *change logical program view* - Logical program view is changed in the sense that only the flag of the statements are changed in case of delete and in the case of insertion of statement, new unique id and suitable flag is assigned. The algorithm is described in algorithm 5.

Algorithm 5 Change logical program view

```
1: Input: A code change, logical program view, flag whether its a update or optimization
   change
2: Output: Modified logical program view
3: **if** Code change is to delete a statement **then**
4:     **if** Code change is for the update or de-optimize **then**
5:         Change the flag of that statement to ‘U’
6:     **else**
7:         Change the flag of that statement to ‘D’
8:     **end if**
9: **else if** Code change is insert or save a statement **then**
10:    Get the largest unique id ($lastID$) assigned to the current statements.
11:    Insert the line at the appropriate place as line number specified in code change
12:    Set the unique id of the statement as $lastID + 1$
13:    **if** it is a save statement **then**
14:        Make the flag as ‘S’
15:    **else**
16:        Make the flag as ‘A’
17:    **end if**
18: **end if**
```

4. *change LLVM IR line number* - After the changes in LLVM IR, the line number information is modified by this function. Working is depicted in a simplified way in the algorithm 6.

5. *Change Optimized view* - Optimizer view is constructed by only including the ‘A’ and ‘S’ flagged statements of the current logical program view. Its working is quite trivial as described in algorithm 7.

7.2.4 Functions related to optimization and de-optimization

Functions to implement the optimizations and de-optimizations are currently implemented manually for common sub-expression elimination, dead code elimination and copy propagation optimization. Ideally, these all should be done from optimization specification automatically. Each optimization has been given a unique id. E.g. let us look into the

Algorithm 6 Change LLVM IR line number information

```
1: Input: code change specification
2: Output: LLVM IR with modified line number
3: $offset = 0$
4: for each instruction in the LLVM IR in the control flow sequence do
5:    $current\_line\_number$ is the line number of the current instruction
6:    if this instruction was next to a deleted instruction then
7:       $offset := offset - 1$
8:    else if $current\_line\_number >$ BASE i.e. this is a inserted line then
9:       $offset := offset + 1$
10:   end if
11:   if $current\_line\_number >$ BASE i.e. this is a inserted line then
12:      Set the line number as $current\_line\_number - BASE + offset$
13:   else
14:      Set the line number as $current\_line\_number + offset$
15:   end if
16: end for
```

Algorithm 7 Change Optimizer view

```
1: Input: logical program view
2: Output: Changed optimized program view
3: for each line in logical program view do
4:    if flag of this line is ‘A’ or ‘S’ then
5:       Add this line to the Optimizer view
6:    end if
7: end for
```

algorithms for enabling [Algorithm 8] and disabling [Algorithm 9] dead code elimination (DCE).

Algorithm 8 Check if dead code elimination(DCE) is enabled

```
1: Input: Dependence relations in the program
2: Output: Returns true if DCE is enabled and mark the dead code
3: Variable $flag := false$
4: for each instruction in the LLVM IR do
5:    if instruction is a store instruction then
6:       if there is no flow dependence starting from this instruction then
7:          Mark this store instruction as dead code
8:          Make $flag := true$
9:       end if
10:   end if
11: end for
12: return $flag$
```

Algorithm 9 Check if dead code elimination(DCE) is disabled

```
1: Input: Dependence relations in the program, old optimization information
2: Output: Returns true if DCE is disabled
3: for each delete statement for this old optimization information do
4:    Traverse the LLVM IR in all paths from the instruction corresponds to the next to
      the delete line
5:    if there is a use of the variable that is in the lhs of the deleted statement then
6:       DCE is disabled
7:       return True
8:    end if
9: end for
```

7.2.5 Functions to find Unsafe Regions

These functions find the dependence relation changes in the updated optimized program and then apply the unsafe region finding algorithm as described in subsection 4.4.3.

7.3 Example Run

Implementation of the system is described in this chapter. There is script to manage the running of the system. When the system executes for the first version of a program it

```
1  #include <stdio.h>
2
3  int main ()
4  {
5              int a;
6              int b;
7              int c;
8              int d;
9              a = 3;
10             b = 5;
11             a = 10;
12             printf ("%d%d",c,d);
13             return;
14 }
```

(a) Optimizer view / source code of version V1

```
1$A$#include <stdio.h>$
2$A$$
3$A$int main ()$
4$A${$
5$A$       int a;$
6$A$       int b;$
7$A$       int c;$
8$A$       int d;$
9$A$       a = 3;$
10$A$      b = 5;$
11$A$      a = 10;$
12$A$      printf ("%d%d",c,d);$
13$A$      return;$
14$A$}$
```

(b) Logical program view of version V1

```
0 -> 0
1 -> 1
2 -> 2
3 -> 3
4 -> 4
5 -> 5
6 -> 6
7 -> 7
8 -> 8
9 -> 9
10 -> 10
11 -> 11
12 -> 12
13 -> 13
14 -> 14
```

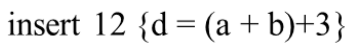
(c) map_opt_view_to_logical in version V1

Figure 7.4: Program views and mappings in program version V1

initializes logical program view, optimizer view and old optimization view. The system also keeps track of the relevant information like old optimization information and logical program view. Now we would show an example with three program versions. The example would show optimization enabling and disabling and unsafe region determination as reported by our system and along with the state of different data structures used in the system.

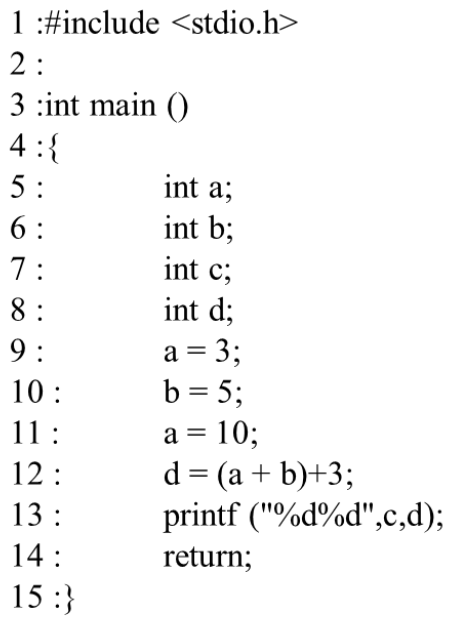
7.3.1 Program version V1

Initial source program is shown in 7.4a. Let us call this initial program version as V1. Source code specified here is also the initial optimizer view. Logical program view of the program is specified in 7.4b. Each line in the logical program view has the unique id for the statement, flag corresponding to the statement and the source line for the statement. Value of the data-structure map_opt_view_to_logical is shown in 7.4c.



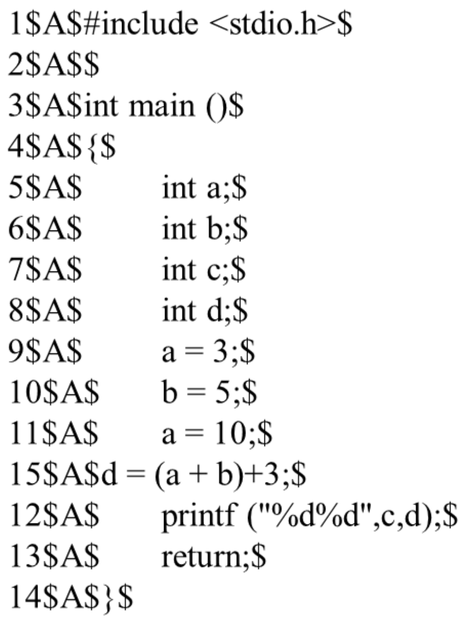
```
insert 12 {d = (a + b)+3}
```

Figure 7.5: Specification of update U1



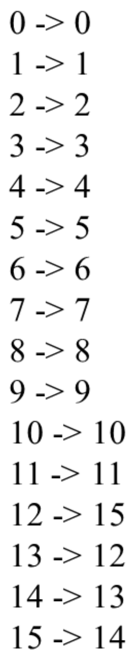
```
1 :#include <stdio.h>
2 :
3 :int main ()
4 :{
5 :            int a;
6 :            int b;
7 :            int c;
8 :            int d;
9 :            a = 3;
10 :           b = 5;
11 :           a = 10;
12 :           d = (a + b)+3;
13 :           printf ("%d%d",c,d);
14 :           return;
15 :}
```

(a) Updated optimizer view of V1



```
1$A$#include <stdio.h>$
2$A$$
3$A$int main ()$
4$A${$
5$A$        int a;$
6$A$        int b;$
7$A$        int c;$
8$A$        int d;$
9$A$        a = 3;$
10$A$       b = 5;$
11$A$       a = 10;$
15$A$d = (a + b)+3;$
12$A$       printf ("%d%d",c,d);$
13$A$       return;$
14$A$}$
```

(b) Updated logical program view of V1



```
0 -> 0
1 -> 1
2 -> 2
3 -> 3
4 -> 4
5 -> 5
6 -> 6
7 -> 7
8 -> 8
9 -> 9
10 -> 10
11 -> 11
12 -> 15
13 -> 12
14 -> 13
15 -> 14
```

(c) Updated map_opt_view_to_logical of V1

Figure 7.6: Updated program views and mappings in program version V1

Update to the program version V1 is shown in Figure 7.5. Let us name this update as U1. The system updates the program by inserting the statement specified in update specification(U1) before line number 12 of optimizer view. Updated logical program view as reported by the system is shown in 7.6b. The new statement is assigned unique id as 15 and its flag is set to 'A' as the statement is active in the program. This update also changes the mapping between unique id and the line numbers in the optimizer view. Updated mapping is shown in 7.6c. Updated optimizer view is presented in 7.6a. LLVM IR is also changed by inserting a set of instructions representing the newly inserted statement. Figure 7.7 presents the set of instructions that represents the newly inserted statement.

```
%temp1_ = load i32* %a
%temp2_ = load i32* %b
%updated_ = add i32 %temp1_, %temp2_
%updated_1 = add i32 %updated_, 3
store i32 %updated_1, i32* %d
```

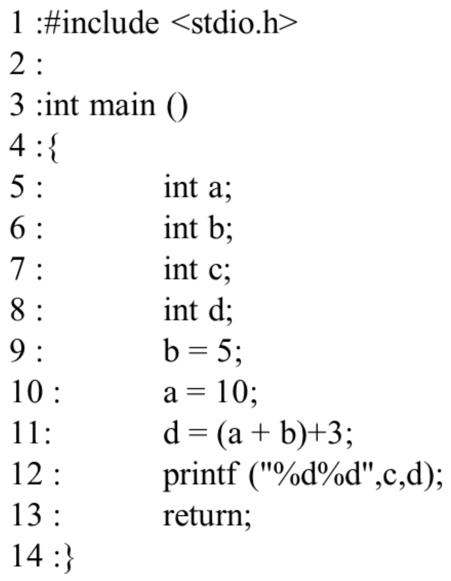
Figure 7.7: LLVM instructions to insert in IR due to the update

```
delete 9
```

Figure 7.8: Code change for DCE in program version V1

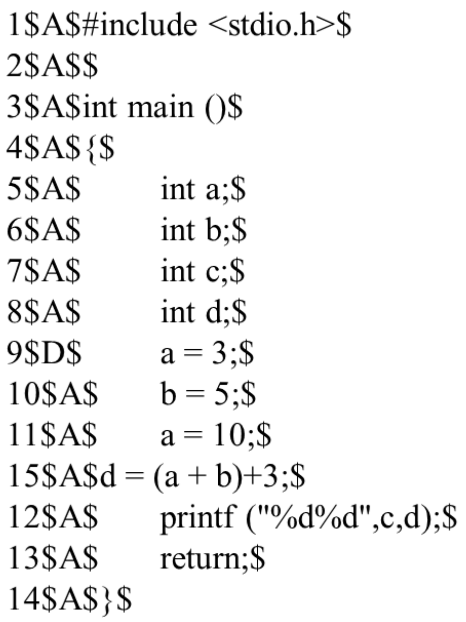
When the system has the updated code it checks for de-optimization. As there is no previous optimization, no de-optimization is possible. After that the system checks for optimization enabling. Then it finds that dead code elimination is enabled as the assignment statement with unique id 9 is dead. This checking is done by calling suitable functions defined in the namespace *DeadCodeElemination* in DCE.h file. To apply DCE, the system generates the code change to apply the optimization. Code change is presented in Figure 7.8. It is then applied in the code to have the final code for version V2. New version logical program view is shown in 7.9b. Here the code change is reflected by changing the flag to 'D' for the statement with unique id 9. Optimizer view of V2 is shown in 7.9a.

Now the system calculates the change in the dependence relation due to the update and the resulting optimization. Dependence relations of V1 and V2 as reported by the system are shown in Figure 7.10. Changes in the flow dependences are shown in Figure 7.11. From this information the unsafe region is calculated and reported as a set of unique ids of the unsafe points as presented in 7.12a. At this moment, old optimization information contains only one line for storing the information of dead code elimination performed in this version. Old optimization information file is shown in 7.12b. In the first field, 2 is the id for denoting that the information is about dead code elimination. For DCE, there is no save and insert statement, so next two fields are empty. Delete statement field contains 9 i.e. statement with unique id 9 is deleted as a result of DCE in the code.



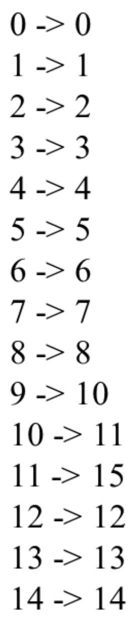
```
1 :#include <stdio.h>
2 :
3 :int main ()
4 :{
5 :          int a;
6 :          int b;
7 :          int c;
8 :          int d;
9 :          b = 5;
10 :         a = 10;
11:          d = (a + b)+3;
12 :         printf ("%d%d",c,d);
13 :         return;
14 :}
```

(a) Optimized optimizer view of V1



```
1$A$#include <stdio.h>$
2$A$$
3$A$int main ()$
4$A${$
5$A$      int a;$
6$A$      int b;$
7$A$      int c;$
8$A$      int d;$
9$D$      a = 3;$
10$A$     b = 5;$
11$A$     a = 10;$
15$A$d = (a + b)+3;$
12$A$     printf ("%d%d",c,d);$
13$A$     return;$
14$A$}$
```

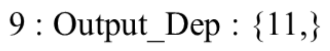
(b) Optimized logical program view of V1



```
0 -> 0
1 -> 1
2 -> 2
3 -> 3
4 -> 4
5 -> 5
6 -> 6
7 -> 7
8 -> 8
9 -> 10
10 -> 11
11 -> 15
12 -> 12
13 -> 13
14 -> 14
```

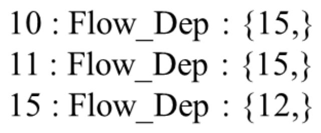
(c) Optimized map_opt_view_to_logical in version V1

Figure 7.9: Optimized program views and mappings in program version V1



```
9 : Output_Dep : {11,}
```

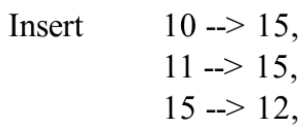
(a) Dependence Relations in V1



```
10 : Flow_Dep : {15,}
11 : Flow_Dep : {15,}
15 : Flow_Dep : {12,}
```

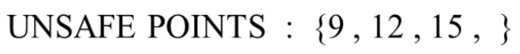
(b) Dependence Relations in V2

Figure 7.10: Dependence Relations in V1 and V2



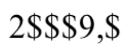
```
Insert      10 --> 15,
            11 --> 15,
            15 --> 12,
```

Figure 7.11: Changes in flow dependence from V1 to V2



```
UNSAFE POINTS : {9 , 12 , 15 , }
```

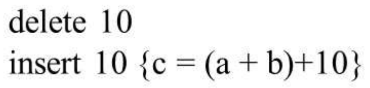
(a) Unsafe points in V1



```
2$$$9,$
```

(b) Old optimization information

Figure 7.12: Unsafe points for the update and old optimization information



```
delete 10
insert 10 {c = (a + b)+10}
```

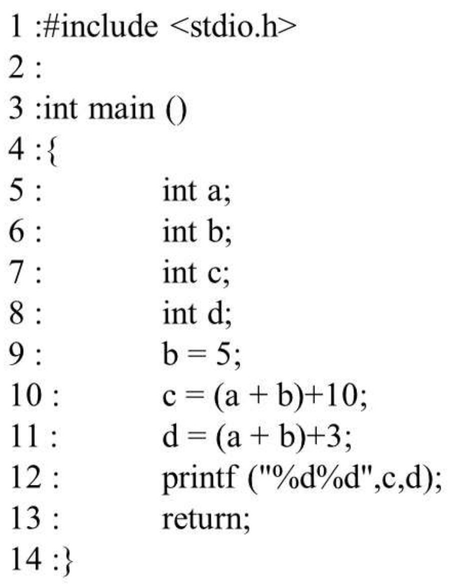
Figure 7.13: Specification of update U2

7.3.2 Program version V2

In the next version, update U2 is shown in Figure 7.13. The update specification instructs the system to delete the statement at line number 10 in the optimizer view. And next update specification instructs the system to insert a statement before line number 10 of optimizer view. The system applies the updates in the optimizer view, logical program view of program version V2. Updated logical program view, optimizer view and map_opt_view_to_logical are shown in Figure 7.14.

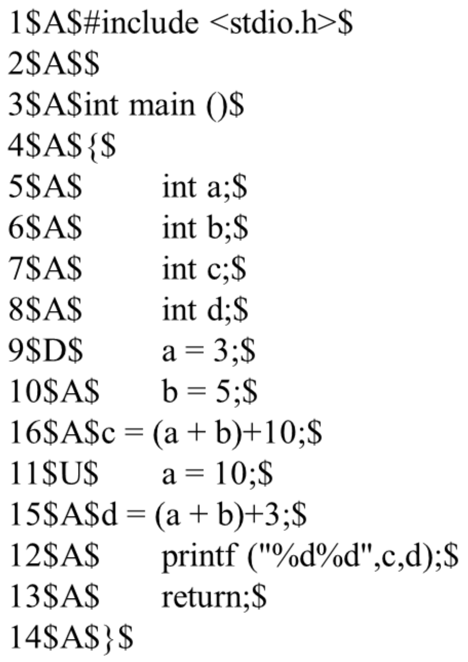
Now the system checks if the existing optimization is disabled. Due to the insertion of the statement with unique id 16 assignment statement of a at unique id 9 is no longer dead. Thus, previously performed DCE is to be undone. For doing this the system prepares a code change as shown in Figure 7.15. After disabling the dead code elimination, the optimizer view and the logical program view looks as in Figure 7.16. It can be noticed that the flag of the new assignment statement with unique id 17 is changed to ‘A’ in the logical program view.

Now the system checks for any optimization enabling. Here common sub-expression is enabled at unique id 16 and 15 with common sub-expression as $a+b$. The system identifies it by calling specific function in $CommonSubexpressionElemination$ namespace in CSE.h file. Then it prepares the code change specification for applying CSE in the code. Code change is presented in Figure 7.17. Save statement is to be inserted before line number 11 of optimizer view and the value of $a + b$ is to saved in a temporary variable. Then the



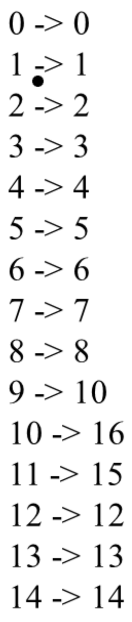
```
1 :#include <stdio.h>
2 :
3 :int main ()
4 :{
5 :            int a;
6 :            int b;
7 :            int c;
8 :            int d;
9 :            b = 5;
10 :           c = (a + b)+10;
11 :           d = (a + b)+3;
12 :           printf ("%d%d",c,d);
13 :           return;
14 :}
```

(a) Updated optimizer view of V2



```
1$A$#include <stdio.h>$
2$A$$
3$A$int main ()$
4$A${$
5$A$        int a;$
6$A$        int b;$
7$A$        int c;$
8$A$        int d;$
9$D$        a = 3;$
10$A$       b = 5;$
16$A$c = (a + b)+10;$
11$U$       a = 10;$
15$A$d = (a + b)+3;$
12$A$       printf ("%d%d",c,d);$
13$A$       return;$
14$A$}$
```

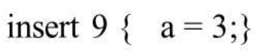
(b) Updated logical program view of V2



```
0 -> 0
1 -> 1
2 -> 2
3 -> 3
4 -> 4
5 -> 5
6 -> 6
7 -> 7
8 -> 8
9 -> 10
10 -> 16
11 -> 15
12 -> 12
13 -> 13
14 -> 14
```

(c) Updated map_opt_view_to_logical of V2

Figure 7.14: Updated program views and mappings in program version V2



```
insert 9 {   a = 3;}
```

Figure 7.15: Code change for DCE de-optimization in V2

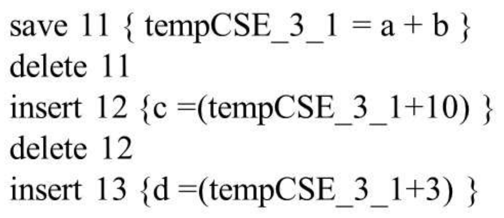
```
1 :#include <stdio.h>
2 :
3 :int main ()
4 :{
5 :          int a;
6 :          int b;
7 :          int c;
8 :          int d;
9:           a = 3.
10 :         b = 5;
11 :         c = (a + b)+10;
12 :         d = (a + b)+3;
13 :         printf ("%d%d",c,d);
14 :         return;
15 :}
```

(a) Optimizer view of V2

```
1$A$#include <stdio.h>$
2$A$$
3$A$int main ()$
4$A${$
5$A$        int a;$
6$A$        int b;$
7$A$        int c;$
8$A$        int d;$
9$D$        a = 3;$
17$A$       a = 3;$
10$A$       b = 5;$
16$A$c = (a + b)+10;$
11$U$       a = 10;$
15$A$d = (a + b)+3;$
12$A$       printf ("%d%d",c,d);$
13$A$       return;$
14$A$}$
```

(b) Logical program view of V2

Figure 7.16: Program views and mappings in program version V2 after de-optimization



```
save 11 { tempCSE_3_1 = a + b }
delete 11
insert 12 {c =(tempCSE_3_1+10) }
delete 12
insert 13 {d =(tempCSE_3_1+3) }
```

Figure 7.17: Code change for enabling CSE in V2

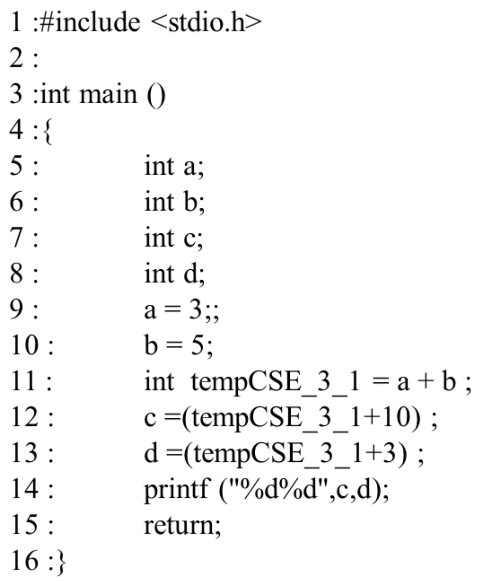
statements with redundant expression should be replaced with the temporary variable usage. Replacement of the variable is done by first deleting the statement and then inserting the desired statement. After code change is applied to the optimizer view and the logical program view are changed and they are as shown in Figure 7.18. It can be noticed that Save statement inserted with unique id 18 and its flag is set to 'S'; two statements inserted are with unique id 19 and 20; statements with unique id 16 and 15 are deleted.

No other optimization is enabled in this version. Thus, Figure 7.18 has the logical program view and the optimizer view of program version V3. To determine the unsafe region dependence relations in V2 and V3 are determined by the system and they are shown in Figure 7.19. From this information the system determines the changes in the flow dependence relations [Figure 7.20]. unsafe points as reported in 7.21a. 7.21b reports the old optimization information. Information about CSE performed in this version is stored.

7.3.3 Program version V3

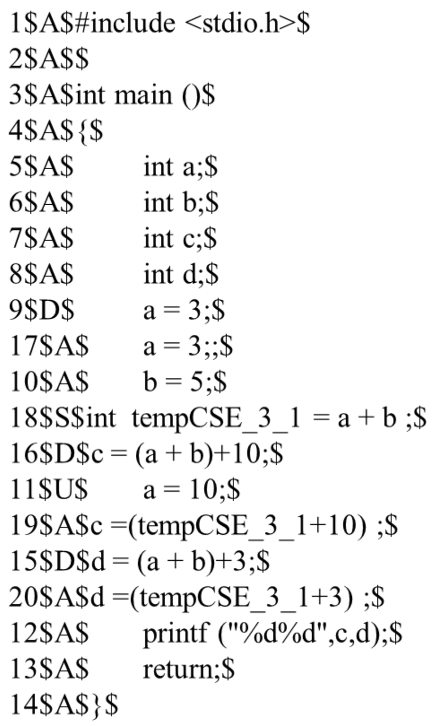
Update for the program V3 is shown in Figure 7.22. After applying the updates the changed program views and the mapping is shown in Figure 7.23.

After updating the program, the system checks for de-optimization and finds that the optimization that was performed in the previous version is to be disabled due to the update. Thus, it prepares the code change specification to de-optimize the program [Figure 7.24]. The system then applies the code changes to the program to get Figure 7.25.



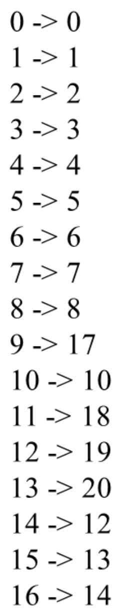
```
1 :#include <stdio.h>
2 :
3 :int main ()
4 :{
5 :         int a;
6 :         int b;
7 :         int c;
8 :         int d;
9 :         a = 3;;
10 :        b = 5;
11 :        int  tempCSE_3_1  = a + b ;
12 :        c =(tempCSE_3_1+10) ;
13 :        d =(tempCSE_3_1+3) ;
14 :        printf ("%d%d",c,d);
15 :        return;
16 :}
```

(a) Optimizer view of V2



```
1$A$#include <stdio.h>$
2$A$$
3$A$int main ()$
4$A${$
5$A$      int a;$
6$A$      int b;$
7$A$      int c;$
8$A$      int d;$
9$D$      a = 3;$
17$A$     a = 3;;$
10$A$     b = 5;$
18$S$int  tempCSE_3_1  = a + b ;$
16$D$c = (a + b)+10;$
11$U$     a = 10;$
19$A$c =(tempCSE_3_1+10) ;$
15$D$d = (a + b)+3;$
20$A$d =(tempCSE_3_1+3) ;$
12$A$     printf ("%d%d",c,d);$
13$A$     return;$
14$A$}$
```

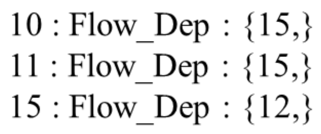
(b) Logical program view of V2



```
0 -> 0
1 -> 1
2 -> 2
3 -> 3
4 -> 4
5 -> 5
6 -> 6
7 -> 7
8 -> 8
9 -> 17
10 -> 10
11 -> 18
12 -> 19
13 -> 20
14 -> 12
15 -> 13
16 -> 14
```

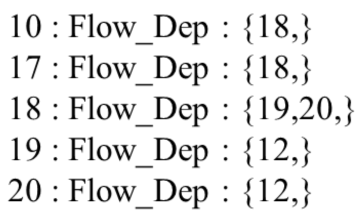
(c) map_opt_view_to_logical in V2

Figure 7.18: Program views and mappings in program version V2 after optimization



```
10 : Flow_Dep : {15,}
11 : Flow_Dep : {15,}
15 : Flow_Dep : {12,}
```

(a) Dependence Relations in V2



```
10 : Flow_Dep : {18,}
17 : Flow_Dep : {18,}
18 : Flow_Dep : {19,20,}
19 : Flow_Dep : {12,}
20 : Flow_Dep : {12,}
```

(b) Dependence Relations in V3

Figure 7.19: Dependence Relations in V2 and V3

```
Delete      10 --> 15,
            11 --> 15,
            15 --> 12,
Insert      10 --> 18,
            17 --> 18,
            18 --> 19,20,
            19 --> 12,
            20 --> 12,
```

Figure 7.20: Changes in flow dependence from V2 to V3

```
UNSAFE POINTS  :  {10 , 11 , 12 , 15 , 16 , 17 , 18 ,
19 , 20 ,  }
```

(a) Unsafe points in V2

```
1$18,$19,20,$16,15,$
```

(b) Old optimization information

Figure 7.21: Unsafe points for the update and old optimization information

```
insert 13 {a = 100}
```

Figure 7.22: Specification of update U3

```
1 :#include <stdio.h>
2 :
3 :int main ()
4 :{
5 :          int a;
6 :          int b;
7 :          int c;
8 :          int d;
9 :          a = 3;;
10 :         b = 5;
11 :int  tempCSE_3_1 = a + b ;
12 :c =(tempCSE_3_1+10) ;
13 :a = 100;
14 :d =(tempCSE_3_1+3) ;
15 :         printf ("%d%d",c,d);
16 :         return;
17 :}
```

(a) Updated optimizer view of V3

```
1$A$#include <stdio.h>$
2$A$$
3$A$int main ()$
4$A${$
5$A$       int a;$
6$A$       int b;$
7$A$       int c;$
8$A$       int d;$
9$D$       a = 3;$
17$A$      a = 3;;$
10$A$      b = 5;$
18$S$int  tempCSE_3_1 = a + b ;$
16$D$c = (a + b)+10;$
11$U$      a = 10;$
19$A$c =(tempCSE_3_1+10) ;$
15$D$d = (a + b)+3;$
21$A$a = 100;$
20$A$d =(tempCSE_3_1+3) ;$
12$A$      printf ("%d%d",c,d);$
13$A$      return;$
14$A$}$
```

(b) Updated logical program view of V3

```
0 -> 0
1 -> 1
2 -> 2
3 -> 3
4 -> 4
5 -> 5
6 -> 6
7 -> 7
8 -> 8
9 -> 17
10 -> 10
11 -> 18
12 -> 19
13 -> 21
14 -> 20
15 -> 12
16 -> 13
17 -> 14
```

(c) Updated map_opt_view_to_logical of V3

Figure 7.23: Updated program views and mappings in program version V3

```
delete 11
delete 12
insert 12 {c = (a + b)+10;}
delete 14
insert 14 {d = (a + b)+3;}
```

Figure 7.24: Code change for CSE de-optimization in V3

```
1 :#include <stdio.h>
2 :
3 :int main ()
4 :{
5 :            int a;
6 :            int b;
7 :            int c;
8 :            int d;
9 :            a = 3;;
10 :           b = 5;
11 :c = (a + b)+10;;
12 :a = 100;
13 :d = (a + b)+3;;
14 :           printf ("%d%d",c,d);
15 :           return;
16 :}
```

(a) Optimizer view of V3

```
1$A$#include <stdio.h>$
2$A$$
3$A$int main ()$
4$A${$
5$A$        int a;$
6$A$        int b;$
7$A$        int c;$
8$A$        int d;$
9$D$        a = 3;$
17$A$       a = 3;;$
10$A$       b = 5;$
18$D$int  tempCSE_3_1  = a + b ;$
16$D$c = (a + b)+10;$
11$U$       a = 10;$
22$A$c = (a + b)+10;;$
19$D$c =(tempCSE_3_1+10) ;$
15$D$d = (a + b)+3;$
21$A$a = 100;$
23$A$d = (a + b)+3;;$
20$D$d =(tempCSE_3_1+3) ;$
12$A$       printf ("%d%d",c,d);$
13$A$       return;$
14$A$}$
```

(b) Logical program view of V3

```
0 -> 0
1 -> 1
2 -> 2
3 -> 3
4 -> 4
5 -> 5
6 -> 6
7 -> 7
8 -> 8
9 -> 17
10 -> 10
11 -> 22
12 -> 21
13 -> 23
14 -> 12
15 -> 13
16 -> 14
```

(c) map_opt_view_to_logical in V3

Figure 7.25: Program views and mappings in program version V2 after optimization

```
10 : Flow_Dep : {18,}
17 : Flow_Dep : {18,}
18 : Flow_Dep : {19,20,}
19 : Flow_Dep : {12,}
20 : Flow_Dep : {12,}
```

(a) Dependence Relations in V3

```
10 : Flow_Dep : {22,23,}
17 : Flow_Dep : {22,}
          Output_Dep : {21,}
21 : Flow_Dep : {23,}
22 : Flow_Dep : {12,}
          Anti_Dep : {21,}
23 : Flow_Dep : {12,}
```

(b) Dependence Relations in V4

Figure 7.26: Dependence Relations in V3 and V4

```
Delete    10 --> 18,
          17 --> 18,
          18 --> 19,20,
          19 --> 12,
          20 --> 12,
Insert    10 --> 22,23,
          17 --> 22,
          21 --> 23,
          22 --> 12,
          23 --> 12,
```

(a) Changes in flow dependence from V3 to V4

```
UNSAFE POINTS  :  {12 , 15 , 18 , 19 , 20 , 21 , 22 ,
23 ,  }
```

(b) Unsafe points in V3

Figure 7.27: Unsafe points for the update in V3

There is no other de-optimization or optimization is possible. So, the Figure 7.25 is the final version code. Now the system calculates the dependence relations for V3 and V4 as shown in Figure 7.26. After this, changes in the flow dependence is calculated as depicted in 7.27a. From this information the unsafe points are determined [7.27b].

Chapter 8

Concluding Remarks

8.1 System capabilities

Building a dynamic software update system has been quite a popular field of work in the last few decades. However, no previous work has defined update safety. In this report, we have defined update safety and devised a method to determine safe regions in a program automatically. This is also the first system where safe regions in a dynamically updatable program are detected automatically as in most of the previous works, programmers have to specify the safe points. This is the first time where we have designed a system that considers optimizations and de-optimizations while determining safe regions in a program.

8.2 Status of implementation

We have implemented the designed system using LLVM 2.8. Given the previous version program, old optimization history and update specification the system can de-optimize and optimize the system as required and then it can output the unsafe points. The whole system is a pass in LLVM. Therefore, it is easy to install and use. Final LLVM IR from the system output can be passed to other LLVM passes such as code generation passes.

Current implementation of the system has the functionality for maintaining logical program view and optimizer view. Currently the system can optimize and de-optimize the program with dead code elimination, common sub-expression elimination and optimize with copy propagation. Given a program, update specification and old optimization information, currently our system can change the program views to reflect update, optimization and de-optimization and finally can calculate the unsafe region. Although the current system does not take into account control dependences at the time of determining unsafe region.

8.3 Future scope

Implementation of the system lacks some facilities. These can be improved in future.

1. System does not support optimizations through optimization specification. Currently, we have tested with Common Sub-expression Elimination, Dead Code Elimination and Copy Propagation. Code for all of these optimizations are done manually now. System would need an implementation of a parser of the optimization specification language to be used.

2. Currently the system only looks at the local variables in a function. Aliasing and global variables are not considered. Inter procedural dependency relations are also to be taken care of.

3. Programmers' view is not related to the core functionality of the system. Therefore, it is not currently implemented. However, to make this system ready for the real world, mapping between programmers' view and optimizer view should be developed.

4. System needs to be tested against large programs. This should be done to test different modules of the system.

Bibliography

[1] David E. Lowell, Yasushi Saito, and Eileen J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *In Proc. ASPLOS*, pages 211–223. ACM Press, 2004.

[2] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22, 1996.

[3] Min Zhao, Bruce R. Childers, and Mary Lou Soffa. A framework for exploring optimization properties. In *CC '09: Proceedings of the 18th International Conference on Compiler Construction*, pages 32–47, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for c. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 72–83, New York, NY, USA, 2006. ACM.

[5] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: a vm-centric approach. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 1–12, New York, NY, USA, 2009. ACM.

[6] Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. *SIGPLAN Notices*, 44:13–24, June 2009.

[7] Cristiano Giuffrida and Andrew S. Tanenbaum. A taxonomy of live updates. In *Proceedings of the 16th Annual Conference of the Advanced School for Computing and Imaging*, 2010.

[8] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9:491–542, October 1987.

[9] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29:1184–1201, December 1986.

[10] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19(6):1053–1084, 1997.

[11] Deborah Whitfield and Mary Lou Soffa. Automatic generation of global optimizers. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 120–129, New York, NY, USA, 1991. ACM.

[12] The llvm compiler infrastructure http://llvm.org/.

[13] The llvm doxygen http://llvm.org/doxygen/.

Appendix A

Optimization Implementation

As it was mentioned in subsection 7.2.4 current system has manual implementation of optimizations and de-optimizations. Currently we have implementation of common subexpression elimination, dead code elimination and copy propagation enabling and CSE and DCE disabling. Here we would discuss the relevant functions and algorithms we have for implementing CSE.

All the functions needed for implementing CSE are inside *CommonSubexpressionElemination* namespace in CSE.h file. The core system interacts with this module by only two functions for checking enabling and disabling of CSE. If CSE is enabled then the code changes required to be applied in the code are prepared by this module also and they are stored in a file which is later read by the module which actually changes the code. Same procedure is followed for de-optimization of CSE. At first we would concentrate on the functions related to enabling of CSE and then we would discuss about the functions related to de-optimization of CSE.

A.1 CSE enabling

Algorithm for checking if CSE is enabled directly follows the CSE specification in SpeLO language in Figure 5.4. A function returns true value if CSE is enabled and otherwise it returns false. Core system invokes this function and on having true as return value it invokes the module to do the code changes. There are two functions - one for checking the code pattern condition in the CSE specification and another for checking dependence condition checking. Algorithm for checking the code pattern is described in algorithm 10. This algorithm basically finds the congruent expressions in the program.

Algorithm 10 find_commonSubExpression

```
1: Input: Expression($e$) whose congruent expression is to be found, concerned
   function($f$)
2: Output: Set of expressions in $f$ that are congruent to $e$
3: for each Basic Block($BB$) in function $f$ do
4:     for each instruction($I$) in the $BB$ do
5:         if $I$ is of type BinaryOperator instruction then
6:             if operands of $e$ matches operands of $I$ then
7:                 Add $I$ to the output set
8:             end if
9:         end if
10:    end for
11: end for
12: return output set
```

Dependence condition are checked using the algorithm 11. After the system decides that CSE is enabled, it prepares the code change specification to carry out the optimization in the code. The algorithm used for preparing the code changes directly follows from the action part in the CSE specification in SpeLO language described in Figure 5.4. Code changes are prepared as shown in algorithm 12. This code change specification is later read by the concerned module in the system to change the code.

A.2 CSE disabling

For checking if any previously performed CSE optimization is disabled it needed to consult the old optimization information. There we can find the specification of save statements, delete statements and insert statements for all previously performed CSE optimizations in the code. Save statements, delete statements and insert statements are represented as sets of unique ids. Checking for disabling of CSE is done using by the algorithm 13. If the system decides that a previously performed CSE is to be disabled then it prepares the code changes to de-optimize. The algorithm to prepare this code change is described in algorithm 14.

Algorithm 11 check_depend_conditions

```
1: Input: Set of common sub expressions, dependence relation in the program
2: Output: Return true if CSE is enabled and code change is prepared
3: Let $BeginS$ is the expression which precedes the other among two common sub-expressions
4: Let $EndS$ is the other expression
5: Let $AntiDEPSet$ is the set of expressions which are anti dependent on $BeginS$
6: for each expression($E$) in $AntiDEPSet$ do
7:     Let $FlowDEPSet$ is the set of expressions flow dependent on $E$
8:     if $FlowDEPSet$ contains $EndS$ then
9:         Anti-dependence condition is not satisfied, CSE is not enabled
10:        return false
11:    end if
12: end for
13: if $BeginS$ dominates $EndS$ then
14:     All path condition is also satisfied, CSE is enabled
15:     Call function get_code_change_for_CSE to prepare the code change spec for CSE
16:     return true
17: end if
18: return false
```

Algorithm 12 get_code_change_for_CSE

```
1: Input: Set of common sub expressions ($ESet$)
2: Output: Prepares the code change specification and writes them to specific file
3: Creates the name of the temporary variable to be used. Let us say it to be *temp*
4: Create a *insert* code change at the line number of the first statement with common sub-expression (*save statement*)
5: Convert the common sub-expression to post-fix notation ($Postfix1$)
6: for each common sub-expression $E$ in the set $ESet$ do
7:     Get the statement($S$) in which $E$ resides
8:     Get Source line number($l$) of $S$
9:     Create a *delete* code change at $l$ (*delete statement*)
10:    Convert $S$ to post-fix notation ($Postfix2$)
11:    Replace every occurrence of $Postfix1$ in $Postfix2$ with *temp* to get *ReplacedStmt*
12:    Create a *insert* code change at $l$ with source line as *ReplacedStmt* (*insert statement*)
13: end for
14: Write the code changes to a specific file
```

Algorithm 13 isDisabledCSE

```
1: Input: Save statement($SaveS$), Insert statement set($InsertSet$), delete statement
   set($DeleteSet$) and dependence relation of the program
2: Output: Returns true if CSE specified by the input argument is disabled or otherwise
   returns false
3: for each statement($S$) in the set $InsertSet$ do
4:   if $S$ is not flow dependent on $SaveS$ then
5:     CSE is disabled
6:     return true
7:   end if
8:   Let $AntiDEPSet$ is the set of statements that are anti dependendent on $SaveS$
9:   for each statement($E$) in $AntiDEPSet$ do
10:    Let $FlowDEPSet$ is the set of expressions flow dependent on $E$
11:    if $FlowDEPSet$ contains any statement from $InsertS$ then
12:      CSE is disabled
13:      return true
14:    end if
15:  end for
16:  return false
17: end for
```

Algorithm 14 get_code_change_for_deoptimization

```
1: Input: Save statement($SaveS$), Insert statement set($InsertSet$), delete statement
   set($DeleteSet$)
2: Output: Determines the code changes to apply for de-optimization and writes them
   to a file
3: for each statement($S$) in $SaveS$ do
4:   Let $l$ be the line number in optimizer view of the statement $S$
5:   Create a delete code change at $l$
6: end for
7: for each statement($S$) in $InsertS$ do
8:   Let $l$ be the line number in optimizer view of the statement $S$ (Calculated from
     map_opt_view_to_logical)
9:   Create a delete code change at $l$
10: end for
11: for each statement($S$) in $DeleteS$ do
12:   Let $src\_line$ be the source line corresponding to $S$ (Got from the logical program
      view)
13:   Let $l$ be the line number corresponding replacement statement of this statement
14:   Create a insert code change at $l$ with source line as $src\_line$
15: end for
16: Write the code changes to a specific file
```