# A Framework for Exploring Optimization Properties

Min Zhao[1], Bruce R. Childers[2], and Mary Lou Soffa[3]

[1] Hewlett Packard
min.zhao@hp.com
[2] University of Pittsburgh
childers@cs.pitt.edu
[3] University of Virginia
soffa@cs.virginia.edu

**Abstract.** Important challenges for compiler optimization include determining what optimizations to apply, where to apply them and what is a good sequence in which to apply them. To address these challenges, an understanding of optimization properties is needed. We present a model-based framework, FOP, to determine how optimizations enable and disable one another. We combine the interaction and profitability properties to determine a "best" sequence for applying optimizations. FOP has three components: (1) a code model for the program, (2) optimization models that capture when optimizations are applicable and their actions, and (3) a resource model that expresses the hardware resources affected by optimizations. FOP determines interactions by comparing the create- and destroy-conditions of each optimization with the post conditions of other optimizations. We develop a technique with FOP to construct code-specific optimization sequences. Experimentally, we demonstrate that our approach achieves similarly good code quality as empirical techniques with less compile-time.

## 1 Introduction

The field of code optimization has been extremely successful over the past 40 years. Various reports from research and commercial projects indicate that the performance of software can be improved by 20% to 40% with aggressive optimization [1][2]. However, it has long been known that there are issues with the application of optimizations. First, optimizations may degrade performance in certain circumstances. For example, Briggs and Cooper reported improvements ranging from +49% to –12% for their algebraic re-association optimization [3]. Second, optimizations enable and disable other optimizations so the order of applying optimizations can have an impact on performance [1][18][21][10], which is known as the *phase ordering problem*. Finally, optimization configurations can impact the effectiveness of optimizations (e.g., how many times to unroll a loop or the tile size) [15][4][8]. These problems are compounded when different hardware platforms are considered. Due to these problems, optimizing compilers are not achieving their full potential. To systematically tackle these problems, we need to identify and study the properties of optimizations, especially those that target the application of optimizations. For example, to selectively

O. de Moor and M. Schwartzbach (Eds.): CC 2009, LNCS 5501, pp. 32–47, 2009.

apply only beneficial optimizations, we need to determine the impact of applying an optimization at a particular code point given the resources of the targeted platform (i.e., the *profitability property*). To efficiently determine a code-specific optimization sequence, we also need to detect the disabling and enabling interferences among optimizations (i.e., the *interaction property*) at code points.

There are two general approaches to exploring optimization properties. The first one uses formal techniques. The formal approach has been used to prove the soundness and correctness of optimizations [14][13][16]. Work also has been done to automatically generate the implementation of optimizations [20][21][9][12] from a formal specification. Another approach uses experimental techniques. That is, after performing optimizations, the properties are experimentally determined (e.g., the code is executed to evaluate performance for determining profitability). The empirical approach has been used to determine the correctness of an optimizer [6]. It has also been used to determine profitability and interactions for finding good optimization sequences and configurations [1][8][18][11]. A disadvantage of the experimental approach is its cost and scalability, as the execution of the program is required. It may take hours, or even days, to find a good optimization sequence for a complex program [10]. Ideally, we need a systematic way to address the application of optimizations, which is practical, effective and scalable [22].

Our approach is to develop a model-based framework that applies optimizations based on their properties which are automatically derived from models of the code, the optimizations themselves, and machine resources. These properties guide the compiler in the application of the optimizations. In prior work, we showed how to determine the profitability property from analytic models for code, optimizations and machine resources [23]. Using the models, the profitability of an optimization was determined to avoid the circumstances where an optimization can degrade performance.

This paper presents a **F**ramework for determining **O**ptimization **P**roperties, **FOP** and shows it can be used to determine the *interaction property*, caused by optimizations enabling and disabling other optimizations. We combine the interaction property with the previously studied profitability property to efficiently find a good code-specific optimization sequence. FOP includes code and optimization models. The code model, automatically constructed from the source, captures characteristics about the code related to the pre-conditions of an optimization. The optimization model, constructed by the optimizer engineer, captures the pre-conditions and actions (i.e., code changes) of optimizations. FOP also has a resource model but it is not needed to determine the interaction property.

We present an algorithm that derives the enabling and disabling interaction property for a set of optimizations. The key idea is to determine the **code changes** needed to meet the **pre-conditions** of an optimization, using the **post-conditions** of other optimizations. We also give a novel technique that automatically constructs code-specific optimization sequences using knowledge about the interaction and profitability properties at each code point. The sequences are used to guide the compiler in the actual application of optimizations.

We implemented FOP and used it to find code-specific sequences for a set of optimizations, including copy propagation(CPP), constant folding(CTF), dead code elimination(DCE), partial redundancy elimination(PRE), loop invariant code motion(LICM), global value numbering(GVN), branch elimination(BRE), branch

chaining(BRC), and register allocation(RA). We compared our technique with a fixed-order approach and an empirical approach. The results show that our approach achieves better program performance than the fixed-order technique and determines similarly good sequences as the empirical approach with up to a 43 times reduction in compile-time. Our technique scales to large programs because it does not need to execute the program.

The contributions of this paper include: a formalization of optimization application and the interaction property; a framework to determine enabling and disabling interactions among optimizations; a technique to determine optimization order from the interaction and profitability properties; and, a study that demonstrates the usefulness of FOP in addressing the phase ordering problem.

## 2 Model-Driven Optimization

In this section, we formally define the interaction property. We start with basic definitions needed to express disabling and enabling conditions for an optimization. To apply an optimization, we must ensure that the semantics of a program are not changed by the optimization. Thus, a set of **pre-conditions** (both text and dependencies) is needed for an optimization to be applicable. When an optimization is applicable in some context, it can cause code conditions to change so that another optimization is **enabled** or **disabled**. We define how optimizations enable and disable one another. We begin with the definition of a Boolean operator ~ that returns true when the conditions D are met in a code segment C.

**Def. 1:** D ~ C if the code conditions D are true in code C; /~ is the negation of ~.

We express an optimization O as $[O^{Pre}, O^{Act}]$, where $O^{Pre}$ represents the pre-conditions needed before the actions (i.e., code changes) $O^{Act}$ are applied for semantic correctness. We express the application of an optimization as:

$$(C)\ [O^{Pre}, O^{Act}]_s\ \langle R\rangle \Rightarrow (C')\ [O^{properties}]_s$$

where C is a code segment with a statement point, S, at which the optimization is applied. If $O^{Pre}$ ~ C, the optimization is applicable. $O^{Act}$ is applied in this case to C and C' is produced ($\Rightarrow$). R is the machine resources upon which the code segment is executed. $O^{properties}$ represents the different optimization properties that can be derived, such as interaction and profitability.

As can be seen, optimization properties depend on code context C, the optimization O and the machine resources R. We model each one of these components and use these models to analyze optimization properties; that is, $C_M\ O_M\ R_M \Rightarrow O^{properties}$ where $C_M$ is the code model, $O_M$ is an optimization model and $R_M$ is the resource model (the subscript "M" refers to a model, rather than a specific optimization, code sequence or resource).

Instead of applying the optimizations, we use models to express the results of the optimizations and analyze the results to determine the properties. In addition, unlike actually applying optimizations, we do not apply a data flow algorithm after each optimization to detect data flow changes. We do this by analyzing the code model.

An example of our technique is shown in Figures 1 and 2. We give a brief discussion to motivate the definition of the interaction property. The example describes the

determination of enabling interactions for copy propagation (CPP) and dead code elimination (DCE). A small source program is provided in Figure 1(a). From the source, FOP automatically generates the dependences needed for the code model as shown in Figure 1(b). A dependence is expressed as <$S_i$, $S_j$, type, dir, pos>. For example, there is a *flow* dependence between $S_1$ and $S_2$ which has equal direction for the first operand. Thus, the dependence is <$S_1$, $S_2$, flow, =, 1>. Figure 1 shows the optimization specification for DCE and CPP in (c) and (d).

We next define the enabling and disabling conditions for optimizations. Then, we present an efficient technique to compute the interaction property.

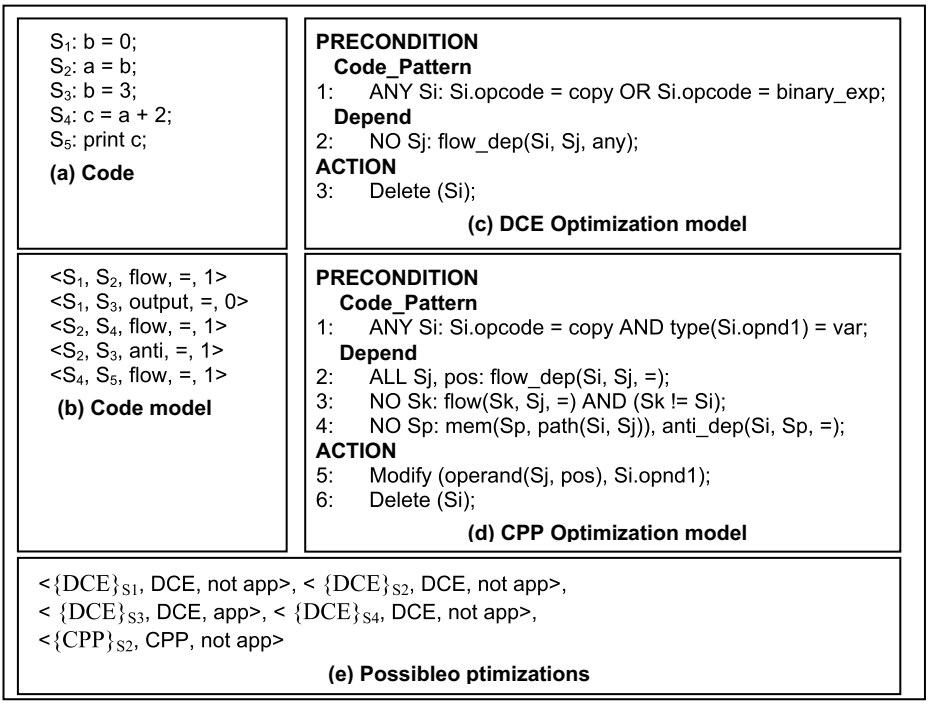


**Fig. 1.** An Example of Determining Interaction

**Def. 2:** Given a code segment, C, an optimization $O_i$ **enables** an optimization $O_k$ if the application of $O_i$ creates the pre-conditions of $O_k$, expressed as:

$$O_i \text{ enables } O_k \text{ if } (C)\,[O_i^{Pre}, O_i^{Act}]_S \Rightarrow (C') \wedge [O_k^{Pre}]\ /\sim C \wedge [O_k^{Pre}] \sim C'$$

**Def. 3:** An optimization $O_i$ **disables** $O_k$ if the application of $O_i$ destroys the pre-conditions of $O_k$:

$$O_i \text{ disables } O_k \text{ if } (C)\,[O_i^{Pre}, O_i^{Act}]_S \Rightarrow (C') \wedge [O_k^{Pre}] \sim C \wedge [O_k^{Pre}]\ /\sim C'$$

Intuitively, to determine the enabling and disabling interaction property between $O_i$ and $O_k$, we need to analyze the **code changes** caused by applying $O_i$ and the **code changes** that can **create** or **destroy** the **pre-condition** of $O_k$.

**Def. 4:** The **post-condition** of O, $[O^{\text{Post-C}}]_S$, is the set of the code changes, $C_\Delta$, after applying O at statement S in code segment C. We use ● to indicate the inclusion of the changes that are made to C by the optimization; that is, C' = C ● $C_\Delta$.

$$[O^{\text{Post-C}}]_S = \{ C_\Delta \mid (C)[O^{\text{Act}}]_S \Rightarrow C \bullet C_\Delta \}$$

We also define a set of code changes that are needed to create or destroy an opportunity for an optimization, O.

**Def. 5:** The **create-condition** of O, $\{[O^{\text{Create-C}}]_S\}$, is the set of code changes, $C_\Delta^i$ that make O applicable at statement S in code segment C.

$$\{[O^{\text{Create-C}}]_S\} = \{\{ C_\Delta^i \} \mid [O^{\text{Pre}}]_S \not\sim C \wedge [O^{\text{Pre}}]_S \sim C \bullet C_\Delta^i \}$$

**Def. 6:** The **destroy-condition** of O, $\{[O^{\text{Destroy-C}}]_S\}$, is the set of code changes, $C_\Delta^i$ that make O not applicable at statement S in code segment C.

$$\{[O^{\text{Destory-C}}]_S\} = \{\{ C_\Delta^i \} \mid [O^{\text{Pre}}]_S \sim C \wedge [O^{\text{Pre}}]_S \not\sim C \bullet C_\Delta^i \}$$

To detect enabling and disabling interactions, we compute the code changes that enable an optimization $O_k$ by comparing the post-conditions of other optimizations, say $O_i$, against the create and destroy-conditions for $O_k$.

**Theorem 1:** An optimization $O_i$ **enables** $O_k$ if there exists a $C_\Delta^i$ in $\{[O_k^{Create-C}]_S\}$ such that $C_\Delta^i \subseteq [O_i^{\text{Post-C}}]_S$.

**Proof:** Straightforward, based on the Definitions 2 and 5.

**Theorem 2:** An optimization $O_i$ **disables** $O_k$ if exists a $C_\Delta^i$ in $\{[O_k^{Destroy-C}]_S\}$ such that $C_\Delta^i \subseteq [O_i^{\text{Post-C}}]_S$.

**Proof:** Based on Definitions 3 and 6.

We develop a new algorithm to determine the enabling and disabling interactions of optimizations at the per-statement level. For a statement in the program, the interaction algorithm determines how a set of optimizations interacts with one another. We now give a high-level overview of the interaction algorithm, which is discussed in detail in Section 3.3. The algorithm has three steps.

**Step 1:** For each O ∈ ***O*** and each S∈ C, compute the code changes needed to create or destroy an optimization opportunity.

**Step 2:** For each O ∈ ***O*** and each S∈ C, compute the post conditions after applying O at point S in C.

**Step 3:** For each O ∈ ***O*** and each S∈ C, compare create- and destroy- conditions with post conditions of all optimizations to determine enabling and disabling properties.

Returning to Figures 1 and 2, when the interaction algorithm starts, it generates the specific post-, create-, and destroy- conditions for every possible optimization

opportunity in the code. Figure 1e shows the possible optimizations. We show the details for two optimizations, $\{DCE\}_{S3}$ and $\{CPP\}_{S2,}$ in Figures 2(a) and (b).

$\{DCE\}_{S3}$ is a dead code elimination that operates on $S_3$ and is applicable. Thus, there is only one create- condition for $\{DCE\}_{S3}$ which is simply "true". There are three destroy-conditions for $\{DCE\}_{S3}$. The first one is deleting $S_3$. The second one is modifying $S_3$'s operation. The third one is inserting a flow dependence that has $S_3$ as the source. The post-conditions for $\{DCE\}_{S3}$ show how it changes the code model, which includes deleting $S_3$, deleting the anti-dependence between $S_2$ and $S_3$ and deleting the output dependence between $S_1$ and $S_3$. Similarly, the create-, destroy- and post-conditions are generated for $\{CPP\}_{S2}$ from the CPP optimization model.

In the last step, the interaction algorithm compares the create- and destroy-conditions with the post-condition of other optimizations and determines the interactions. For example, there is only one condition needed for $\{CPP\}_{S2}$ to be applicable; i.e., <delete_dep, anti,$S_2$,$S_3$,=>. When the interaction algorithm checks $\{DCE\}_{S3}$'s post-conditions, it finds that $\{DCE\}_{S3}$ changes the dependency by deleting the anti-dependence between $S_2$ and $S_3$. This condition matches with the enabling expression of $\{CPP\}_{S2}$. Thus, $\{DCE\}_{S3}$ **enables** $\{CPP\}_{S2}$.

**<{DCE}$_{S3}$, DCE, applicable>**
**<Create-conditions,** true>
**<Destroy-conditions,** <delete $S_3$> ∨ <modify_opcode, $S_3$, ≠, copy/binary_arith>
∨ <insert_dep, flow, $S_3$, any, any> >
**<Postcondition,** <delete $S_3$> ∧ <delete_dep, anti, $S_2$, $S_3$, =,>
∧ <delete_dep, output, $S_1$, $S_3$, =,>>

**(a) Detailed conditions for {DCE}$_{s3}$**

**<{CPP}$_{S2}$, CPP, not applicable>**
**<Create-conditions,** <delete_dep,anti,$S_2$,$S_3$, =,>>
**<Destroy-conditions,** <delete $S_2$> ∨ <modify_opcode, $S_2$, ≠, copy>
∨ <modify_opnd, $S_2$, dst, ≠, var > ∨ <modify_opnd, $S_2$, opnd1, ≠, var>
∨ <delete_dep, flow, $S_2$, $S_4$, =,> ∨ <insert_dep, flow, $S_2$, any, =,>*
∨ <insert_dep, flow, any, $S_4$, =, any ≠ $S_2$>
∨ <insert_dep, anti, $S_2$, any, =, in_any_path($S_2$, $S_4$)>>
**<Postcondition,** <delete $S_2$> ∧ <delete_dep, flow, $S_1$, $S_2$, =,>
∧ <modify_opnd, $S_4$, opnd1, $S_2$.opnd1> ∧ <delete_dep, flow, $S_2$, $S_4$, =,>
∧ <insert_dep, flow, $S_1$, $S_4$, =,>>

**(b) Detailed conditions for {CPP}$_{S2}$**

**Fig. 2.** Determining Interaction

## 3 FOP Components

To determine the enabling and disabling interactions, FOP uses models for both code and optimizations. The **code model** expresses the code context that is needed in determining the create-conditions, destroy-conditions and post-condition of an optimization. We use the control flow graph (CFG) as the basic code model and identify a

distinguished code point, S, (i.e., statement) where an optimization may be applied. The general form of the statement is three-address code. We use dependencies to represent data flow information. A dependence is represented with the tuple $< Ss, Sd, type, dir, pos >$. There are four *types* of dependencies: flow, anti-, output, and control dependencies [5]. The *dir* element records the direction of the dependence, which can be forward, backward or equivalent, represented by <, >, or =, respectively. The direction is needed in loop optimizations. The *pos* element records the position of the operand dependence between $S_s$ and $S_d$. An **optimization model** expresses the pre-conditions $O^{Pre}$ and the actions $O^{Act}$ of an optimization. We developed an optimization specification language, SpeLO, based on Gospel that specifies a class of scalar and loop optimizations [21]. SpeLO extends Gospel to a larger class of optimizations, including path-based ones (e.g., PRE). A compiler engineer uses SpeLO to describe the optimization model for FOP.

## 3.1 Optimization Models

### 3.1.1 SpeLO

The structure of a SpeLO specification is shown in Fig. 3. The PRECONDITION section specifies the conditions, $O^{Pre}$, under which the optimization is safe to apply. There are two parts in the pre-condition section: code patterns, $O^{Pattern}$ and dependencies, $O^{Depend}$.

```
OptName
PRECONDITION
    Code_Pattern
        [Quantifier ElementId: mem_list, element_format_list;]+
    Depend
        [Quantifier ElementId [, pos]: mem_list, condition_list;]*
ACTION
    [primitive_operation;]*
```

**Fig. 3.** The Format of a SpeLO Specification

***Code Pattern.*** The code pattern gives the generic code structure that must be satisfied for the optimization to be applicable. The code pattern identifies program elements such as a statement or loop, which represent the distinguished code point where the optimization can be applied. If an element is a statement, then the code pattern expresses what statement operator and operands are needed for the optimization to be applicable. A quantifier includes **ANY** referring to any matching element, **ALL** referring to all matching elements, and **NO** indicating that there are no matching elements. *mem_list* specifies a set to which an element belongs, such as a path or a loop. Format expressions are used to give the specific format of the code element, *element_format_list*. Multiple expressions can be combined with "AND" and "OR". To standardize the format (without losing generality), SpeLO uses disjunctive normal form (DNF) to express the combination of multiple expressions.

***Depend.*** The second part of the PRECONDITION section gives the generic control and data dependence relationships that must be satisfied for the optimization to be applicable. The *condition_list* consists of the relations combined by AND and OR

operators in DNF. A relation can be a dependence relation in the form of *type_of_dependence* (*Ss*, *Sd*, *dir*). The dependence's type and direction are the same as in the code model. A position tag, *pos*, can also be given in a dependence relation to indicate the position of the dependence should be checked.

The ACTION section describes the modifications to the code or code properties (e.g., value number of a statement) that would result from applying the optimization. We decompose these effects into four primitive operations on the code: *move*, *add*, *delete* and *modify*. The semantics of the primitive operations are typical edit operations; they can be used to express the actions of optimizations [21].

### 3.1.2 Partial Redundancy Elimination (PRE) Optimization Model

Figure 4 gives the optimization model for PRE, a path-specific optimization. Line 1 shows that when a statement $S_i$ is a binary expression, there is a possible PRE opportunity. All the same expressions $S_j$, executed on a path to $S_i$ without a redefinition between them are found (lines 2-3). The definitions $S_p$ of this statement are selected, where there is a path that does not include the collected same expressions (line 4). The immediate predecessors of the statement on the path that does not include the same expression are saved. These are insertion points where the computation should be added. At the same time, it is required that at these insertion points, the expression is anticipated, as shown on line 5. When applying PRE, the computation is added at the insertion points and before the same expressions $S_j$. The same expressions $S_j$ and statement $S_i$ are replaced with the assignment on lines 6-9.

```
PRECONDITION
   Code_Pattern
1:    ANY Si: Si.opcode = binary_exp;
2:    ALL Sj:  mem(path(Entry, Si)), Sj.opcode = Si.opcode
                AND Sj.opnd1= Si.opnd1 AND Sj.opnd2 = Si.opnd2;
   Depend
3:    NO Sk: anti_dep(Sj, Sk, =) AND flow_dep(Sk, Si, =);
4:    ALL Sp: flow_dep(Sp, Si, =) AND ¬ in_every_path(Sj, Sp, Si, save pred(Si))  AND
                ¬ in_any_path(pred(Si), Sj, Si) to Bq)
5:    NO Bl: mem(Bq), ¬post_dom(B(Si), Bl);
ACTION
6:    Add ((new_temp= Si.opnd1 Si.opcode Si.opnd2), Bq);
7:    Add (new_temp=Si.opnd1 Si.opcode Si.opnd2), Sj);
8:    Modify (Sj, (Sj.dst = new_temp));
9:    Modify (Si, (Si.dst = new_temp));
```

**Fig. 4.** PRE Optimization Model

## 3.2 Interaction Algorithm

Given an optimization and a program point, the interaction algorithm determines the enabling and disabling interactions by comparing its post-condition with create- and destroy-conditions of other optimizations. The algorithm first considers every statement in the code segment and every optimization in a set of optimizations to determine the create-conditions and destroy-conditions for each optimization opportunity. In the second step, the algorithm generates the post-conditions for each optimization

opportunity. In the last step, each optimization's create and destroy-conditions are compared with post-conditions of other optimizations to compute the enabling and disabling interaction.

### 3.2.1 Step 1: Generating Create- and Destroy-Conditions

For each optimization, O, and a code point, S, the interaction algorithm compares $O^{Pre}$ with the create- and destroy-conditions for other optimizations. That is, for each $O \in \boldsymbol{O}$ and each $S \in C$, we compute the create- and destroy-conditions using the pre-conditions and C: $(C)[O^{Pre}]_S \Rightarrow \{[O^{Create\text{-}C}]_S\}$ and $(C)[O^{Pre}]_S \Rightarrow \{[O^{Destroy\text{-}C}]_S\}$.

To find these code changes, the PRECONDITION of each optimization model is compared with the code model. The Code Pattern and Depend parts of the PRECONDITION section are consider separately; that is $[O^{Pre}] = [O^{Pattern} \wedge O^{Depend}]$. For example, consider the form, **A AND B OR D**, where A, B and D are basic expressions. The code pattern expression for CPP is

$$\text{Si.opcode} = \text{copy AND type (Si.opnd1)} = \text{var}$$

Thus, A is "Si.opcode = copy" and B is "type(Si.opnd1)=var". D is not given. For Depend, the second dependence expression of CPP is "flow(Sk, Sj, =) AND (Sk != Si)". In this case, A is "flow(Sk, Sj, =)" and B is "Sk != Si".

***Code Pattern.*** When the code model (i.e., the state at code point S) is compared to the code pattern, two cases are possible. When the code *matches* Code Pattern, the create-conditions are true. Thus, step 1 of the interaction algorithm needs only to determine the destroy-conditions, $\{[O^{Destroy\text{-}C}]_S\}$. When the code does *not match* the pre-conditions, the destroy-conditions are true and the algorithm determines only create-conditions, $\{[O^{Create\text{-}C}]_S\}$.

**Case 1: $[O^{Pattern}]$ ~ C:** The first case happens when the current statement S in C matches the code pattern. The destroy-conditions are generated. Suppose, the code pattern expression is A AND B OR D, the destroy-conditions are created as:

$$[O^{Destroy\text{-}Pattern\text{-}C}]_S = (\text{delete S}) \vee (\neg A \wedge \neg D) \vee (\neg B \wedge \neg D)$$

For example, the destroy-conditions for $CPP_i$ are:

$$(\text{delete } S_{Stmtid}) \vee (\text{modify\_opnd}, S_{Stmtid}.\text{opcode} \neq \text{copy}) \vee (\text{modify\_opcode}, \text{type}(S_{Stmtid}.\text{opnd1}) \neq \text{var})$$

**Case 2: $[O^{Pattern}]_S$ /~ C:** Another case occurs when the current statement S does not match the code pattern. The interaction algorithm generates the create-conditions, considering only the legal code changes that can be made by other optimizations. For example, constant folding requires that both operands are constant. Even if a statement has a variable operand, it is possible to perform constant folding when the statement's variable operand can be changed to a constant by other optimizations (e.g., constant propagation).

$$[O^{Create\text{-}Pattern\text{-}C}]_S = (\text{if } (A \wedge \neg B) \text{ insert B}) \vee \text{if } (\neg A \wedge B) \text{ insert A}) \vee (\text{if } (\neg D) \text{ insert D})$$

When it is impossible for any code change made by another optimization to match Code Pattern, an optimization opportunity is not created.

**Depend.** After determining create- and destroy-conditions for the code pattern, the create- and destroy-conditions are generated for the Depend specification. For each quantifier ANY, ALL and NO, there are two cases, corresponding to a match and no match between $O^{Depend}$ and C. Again, assume the dependence rules are in the form of A AND B OR D.

**Case 1:** For the ALL quantifier, if there is a match $[O^{Depend}]_S \sim C$, then the create-condition is true and the destroy-conditions are generated as below. "Alldep" represents the All quantifier.

$$\{[O^{Destroy\text{-}alldep\text{-}C}]_S\} = (\text{delete S1}) \wedge ... \wedge (\text{delete Sn}) \vee (\text{insert A\^{}B})^* \vee (\text{insert D})^*$$

$\{[O^{Destroy\text{-}alldep\text{-}C}]_S\}$ shows that if all of the matching statements are deleted, then the optimization opportunity is destroyed. It also includes insertion of a dependence that matches the dependence rule, (insert $A \wedge B$)* or (insert D).

**Case 2:** For the ALL quantifier, when the code model does not match the dependence rule $[O^{Depend}]_S \not\sim C$, the create-conditions are generated as:

$$\{[O^{Create\text{-}alldep\text{-}C}]_S\} = (\text{insert A\^{}B})^* \vee (\text{insert D})^*$$

Similarly, the interaction algorithm generates create and destroy-conditions for the ANY and NO quantifiers.

### 3.2.2 Step 2: Generating Post-conditions

The post-conditions of O are the code changes after applying the actions of O. In its second step, the interaction algorithm generates the post-conditions for each optimization opportunity according to the actions of the optimization.

**Step 2:** For each O in ***O*** and S, $(C)[O^{Act}]_S \Rightarrow (C) \bullet [O^{Post}]_S$

The primitive operations in the ACTION section specify the code modifications made by the optimization. The actions are decomposed into individual modifications during generation of the specific post-conditions.

**Table 1.** Generating Post-Conditions

| Action | (Pattern) Code Modifications | (Depend) Dependence Modifications |
|---|---|---|
| **Move** | delete (S)<br>insert (NewS, AfterS) | delete_dep (type, stat, S, dir)<br>insert_dep (type, stat, NewS, dir)<br>insert_dep (type, NewS, stat, dir) |
| **Add** | insert(S, AfterS) | insert_dep (type, S, stat, dir)<br>insert_dep (type, stat, S, dir) |
| **Delete** | delete (S) | delete_dep (type, stat, S, dir) |
| **Modify** | modify_opnd(S, opnd, new_opnd) | delete_dep (type, stat, S, dir)<br>where dep_position = opnd<br>insert_dep (type, stat, S, dir)<br>where dep_position= new_opnd<br>insert_dep (type, S, stat, dir)<br>where dep_position= new_opnd |
| | modify_opcode(S, new_opcode) | -- |

Table 1 shows how to generate post-conditions for each primitive action in the ACTION section. A row corresponds to an action, given in the first table column. The second and third columns give the changes to the code model. For example, the move operation deletes a statement from its original location and inserts a new one at a new location. When a statement is deleted, its dependences must also be deleted. When a new statement is inserted, dependences are inserted at the new location.

### 3.2.3 Step 3: Computing the Interactions

In this final step, the algorithm determines the interactions among optimizations by matching the create- and destroy-conditions of each optimization with the post-conditions of other optimizations.

**Step 3:** Compare conditions in $\{[O_k^{Destroy-C}]_S\}$ and $\{[O_k^{Create-C}]_S\}$ with $[O_i^{Post-C}]_S$:

If there exists a $C_\Delta^i$ in $\{[O_k^{Destroy-C}]_S\}$ such that $C_\Delta^i \subseteq [O_i^{Post-C}]_S$ then $O_i$ disables $O_k$

and if there exists a $C_\Delta^i$ in $\{[O_k^{Create-C}]_S\}$ such that $C_\Delta^i \subseteq [O_i^{Post-C}]_S$ then $O_i$ enables $O_k$

The process for matching $O_i$'s create- and destroy-conditions with the post-conditions is as follows. For each optimization opportunity, the algorithm tries to match the post-conditions of other optimizations. It finds the optimizations whose post-condition matches the condition. Next, it tries to match the set of optimizations whose post-conditions match conditions to enable/disable $O_i$ together. The optimization whose post-conditions match condition C enables/disables $O_i$. The condition action (i.e., *delete*, *insert*, *delete_dep*, *insert_dep*, *modify_opnd*, or *modify_opcode*) and the object (e.g., statement or dependence) are compared. For example, if A is <*delete* $S_3$>, an optimization whose post-condition deletes $S_3$ matches A. If A is <*delete_dep, type,* $S_i$*,* $S_j$*, dir, other_condition*>, the post-condition has to match all parts of condition A. The post-condition has the same type of dependence between $S_i$ and $S_j$, direction, and the other conditions are satisfied.

## 4 Optimization Ordering Using Properties

Typically, compilers apply optimizations in a predetermined order, perhaps guided by a compiler writer's expertise. In our approach, we use the profitability and interaction properties to determine the optimization order at the statement level.

**Def. 7:** The Profit of an optimization O, $O^{profit}$, is the performance difference after applying O. Performance can be defined as execution time, dynamic instruction count or other metrics. Suppose (C) $[O^{Pre}, O^{Act}]_S$ <R> → (C')$[O^{profit}]$, then

$$O^{profit} = \text{Performance}(C', R) - \text{Performance}(C, R)$$

We determine optimization ordering based on properties, expressed as: **$O_i$ before $O_j$**

$$\text{If } (O_i \text{ enables } O_j) \text{ OR } (O_j \text{ disables } O_i)$$

$$\text{OR } (O_i, \text{ no interaction } O_j \text{ AND Profit}(O_i)) > \text{Profit}(O_j))$$

Fig. 5 shows our algorithm to determine phase ordering. A working set, app, tracks which optimizations to consider. A list, seq, holds the optimization sequence determined by the algorithm. app is initialized to all applicable optimizations and seq

is initialized to the empty sequence. The algorithm iterates until the working set is empty (line 3). The algorithm evaluates the profit of optimizations in list, *Profit(O)*, on line 4. The profitability of an optimization is computed analytically [23]. The algorithm selects the optimization $O_k$ with the largest *Profit* as the next optimization in the sequence. $O_k$ is added to seq on line 6. On line 7, the algorithm updates app according to what optimizations are disabled and enabled by $O_k$. We require that when $O_k$ along with other optimizations disables $O_m$ and all the other optimizations are already in the sequence, then we remove $O_m$ from app. For the enabling interaction, we also require that optimizations already in seq do not disable $O_m$, and then we can add $O_m$ to app. We evaluate app until it is empty to achieve the sequence that maximizes the evaluation function.

Although we use a single optimization in the discussion, FOP can determine the properties for a series of optimizations, i.e., the combination of optimizations. In this case, (C)[ $O_{i...k}^{\text{Pre}}$ , $O_{i...k}^{\text{Act}}$ ] <R> $\Rightarrow$ (C´) [ $O_{i...k}^{\text{Properties}}$ ]. The phase ordering among combinations of optimizations can also be determined using optimization properties.

```
1:  app = {all applicable optimization instances};
2:  seq = {};
3:  while (app ≠ empty) {
4:    Evaluate_Profit (list);
5:    select Ok that Profit(Ok) is the best;
6:    seq = seq + { Ok };
7:    app = app – { Ok }
8:    app = app – { Om | Disable({Ok, …}, Om) ∧ {Ok, …} ⊆ seq }
9:    app = app + { Om | Enable({Ok, …}, Om) ∧ {Ok, …} ⊆ seq
                  ∧ ¬∃ (Op ∈ seq ∧ Disable(Op, Om))}
10:   app = app + { Om1Om2 | Enable({Ok, …}, Om1Om2) ∧ {Ok, …} ⊆ seq };  }
```

**Fig. 5.** Algorithm to determine a Good Optimization Sequence

## 5 Experiments

To evaluate FOP, we compared three approaches to applying optimizations: a fixed-order approach, an empirical approach that uses a genetic algorithm (GA) to search for effective optimization sequences [1] and our approach. We run experiments on an Intel Pentium IV 2.4GHz machine, with 512MB of memory and RedHat Linux.

We consider nine optimizations: CPP, CTF, DCE, PRE, LICM, GVN, BRC, BRE, and RA. The optimizations are incorporated into the MachSUIF optimizer [17] for the Intel IA-32 instruction set. The fixed-order sequence is "GVN, BRC, BRE, CPP, CTF, DCE, PRE, LICM, GVN, BRC, BRE, CPP, CTF, DCE, PRE, LICM, RA". The selection of the fixed order was based on a past study of interactions among these optimizations [21]. In all cases, register allocation is done as the last optimization.

The empirical approach (GA) has the same configuration as in [1]. We performed a search for each *function* in a program with 10 generations. Each generation had a population of 20 sequences. Every sequence had 16 optimization passes, picked from the possible optimizations. At each generation, the best 10% of the sequences survive without any change. The remaining part of the new generation is created with a

crossover operation, which is followed by character-by-character mutation (5% mutation rate). We tried more generations but there was no further improvement.

### 5.1 Compile-Time Comparison

The empirical approach both applies optimizations and executes the code to evaluate profitability. For the SPEC benchmarks, the test inputs were used to execute the code. FOP uses the interaction and the profitability properties to determine optimization order. Table 2 shows the compile-time overhead of the approaches.

From the table, the compile-time for the fixed-order is small. It varies from 0.05 to 6.11 minutes. Because the empirical approach (GA) executes the application to determine profitability and to apply optimizations, and it recomputes the data flow to detect interactions, its compile-time is large, varying from 5 minutes to 43.6 hours. Each function is compiled for 200 sequences and evaluated by executing the code. Its total compile-time is related to the compile-time and execution time for each function. For example, there are 106 functions in *gzip*. The average compile-time for a function is 0.8 seconds. The execution time for the test input is 2.4 seconds. Considering the GA search time, it took 1181 minutes to find code-specific sequences for *gzip*.

With our approach, the compile-time is reduced: It varies from 0.7 to 82 minutes. Our approach needs to determine the interaction property among optimizations and to predict the profitability property. Its compile-time depends on the time to determine the interaction property and the time to predict profitability. For example, in *mpeg* the average compile-time to determine the interaction property is about 20 seconds and the compile-time to determine profitability is about 6 seconds. Thus, it took 82.24 minutes for our approach to determine good optimization sequences for *mpeg*.

**Table 2.** Compile-time of Three Approaches (minutes)

| Benchmarks | Fixed-order | Empirical | FOP |
|---|---|---|---|
| adpcm.rawcaudio | 0.05 | 5.41 | 1.14 |
| mpeg2.enc | 1.92 | 726.67 | 82.24 |
| bitcount | 0.15 | 18.97 | 1.66 |
| dijkstra.large | 0.05 | 11.63 | 0.68 |
| FFT | 0.11 | 13.20 | 1.81 |
| 164.gzip | 1.52 | 1180.67 | 53.82 |
| 175.vpr | 6.11 | 1469.23 | 61.23 |
| 181.mcf | 0.53 | 74.64 | 19.54 |
| 197.parser | 5.4 | 976.34 | 49.23 |
| 256.bzip2 | 2.34 | 2618.79 | 58.68 |

### 5.2 Performance Comparison

We compared the performance of the three approaches, as shown in Fig. 6. The figure shows the improvement of the empirical and the model-driven approaches over the fixed-order approach. Performance is measured with dynamic instruction count.

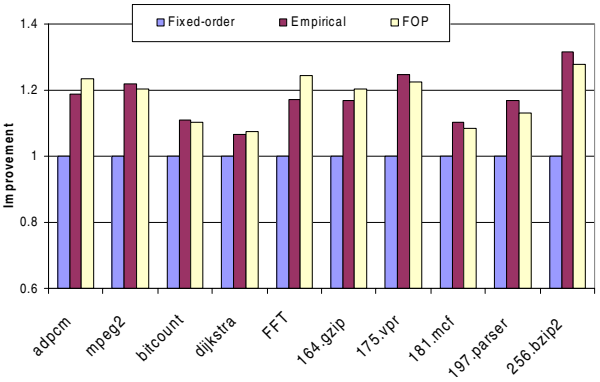


**Fig. 6.** Performance Comparison of Three Approaches

As the figure shows, the empirical and model-driven approaches improve performance more than the fixed-order approach. In *256.bzip2*, the improvement with the empirical and model-driven approaches over the fixed-order sequence is 32% and 28% respectively. In most cases, the model-driven approach has similar performance improvement as the empirical approach. However, the performance of the model-driven approach is better in a few cases. For example, in *adpcm*, the improvement is 19% with the empirical technique and 23% with our model-driven approach. This higher improvement happens because an optimization instance is not applied if it is predicted to be unprofitable.

In terms of memory, FOP uses 51KB to 723KB (average) to store data and control dependence information. The maximum memory requirements ranged from 106KB to 9815KB. On today's machines, this amount of memory is reasonable.

Our experiments show that optimization properties are useful in finding code-specific optimization sequences. Our techniques show it is practical to *analytically model optimizations and compute interactions to find a good code-specific order* in which to apply optimizations.

## 6 Related Work

Formal and empirical approaches have been used to determine the order to apply optimizations. Knuth and Bendix proposed a solution to express optimizations as a set of rewriting rules [9]. Their algorithm detects potential conflicts and resolves them by introducing new rewriting rules, derived from the existing set. However, the procedure is difficult to generalize. Whitfield and Soffa described a framework that enables the exploration, both analytically and experimentally, of properties of optimizations, including the interaction property [20], [21]. They proposed Gospel to express the pre-condition and post-conditions of optimizations. They studied the optimization interactions with proofs or examples. However, their approach can not automatically detect the interactions among optimizations based on code context.

Another approach uses heuristic-driven search algorithms to find a good optimization sequence. Almagor et al. performed a large experimental study on the space of optimization sequences [1]. Although their approach can produce efficient code, it can

be slow. Kulkarni et al. proposed an interactive compilation system, VISTA, which used a genetic algorithm, performance information and user input to select an effective optimization sequence. They further proposed two approaches to improve search performance [7], [10], [11]. Triantifyllis et al. recognized the benefit of finding good optimization sequences [18], [19]. To limit compile-time, their system used a fixed set of optimization sequences and obtained the best result with the set. However, the selection of the sequences should be considered.

## 7 Conclusion

This paper presented a framework (FOP) which can automatically determine optimization properties. We use FOP to determine enabling and disabling interactions among optimizations without actually applying the optimizations. An application of FOP is to find a good code-specific order in which to apply optimizations. This paper presented an algorithm that constructs an effective optimization sequence using the interaction and profitability properties. We implemented FOP and experimentally found good code-specific orders. The results showed that we obtain sequences that have similar performance as an empirical approach with less compile-time. Our work demonstrates that an analytic approach can be used for optimization properties, which are useful in addressing phase ordering.

## References

1. Almagor, L., Cooper, K., Grosul, A., Harvey, T., Reeves, S., Subramanian, D., Torczon, L., Waterman, T.: Finding Effective Compilation Sequences. In: Conf. On Languages, Compilers, and Tools for Embedded Systems (2004)
2. Bodík, R., Gupta, R., Soffa, M.L.: Complete removal of redundant expressions. SIGPLAN Not. 39(4) (April 2004)
3. Briggs, P., Cooper, K.D.: Effective Partial Redundancy Elimination. In: Conf. on Programming Language Design and Implementation (1994)
4. Coleman, S., McKinley, K.S.: Tile Size Selection Using Cache Organization and Data Layout. In: Conf. on Programming Language Design and Implementation (1995)
5. Ferrante, J., Ottenstein, K., Warren, J.: The program Dependence Graph and Its Use in Optimization. ACM Trans. on Programming Languages 9(3) (1987)
6. Jaramillo, C., Gupta, R., Soffa, M.L.: Comparison checking: An approach to avoid debugging of optimized code. In: Nierstrasz, O., Lemoine, M. (eds.) ESEC 1999 and ESEC-FSE 1999. LNCS, vol. 1687, p. 268. Springer, Heidelberg (1999)
7. Kulkarni, P., Hines, S., Hiser, J., Whalley, D., Davidson, J., Jones, D.: Fast Searches for Effective Optimization Phase Sequences. In: Conf. on Programming Language Design and Implementation (2004)
8. Kisuki, T., Knijnenburg, P.M.W., O'Boyle, M.F.P.: Combined Selection of Tile Size and Unroll Factors Using Iterative Compilation. In: Int'l. Conf. on Parallel Architectures and Compilation Techniques (2000)
9. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational problems in abstract algebra. Pergamon Press, Oxford (1970)
10. Kulkarni, P., Whalley, D.B., Tyson, G.S., Davidson, J.W.: Exhaustive Optimization Phase Order Space Exploration. In: Int'l. Symp. on Code Generation and Optimization (2006)

11. Kulkarni, P., Whalley, D.B., Tyson, G.S., Davidson, J.W.: Evaluating Heuristic Optimization Phase Order Search Algorithms. In: Int'l. Symp. on Code Generation and Optimization (2007)
12. Lacey, D.: Program Transformation using Temporal Logic Specifications. PhD dissertation, Univ. of Oxford (August 2003)
13. Lerner, S., Millstein, T., Chambers, C.: Automatically Proving the Correctness of compiler optimizations. In: Conf. on Programming Language Design and Implementation (2003)
14. Lacey, D., Jones, N., Wyk, E., Frederiksen, C.: Proving correctness of compiler optimizations by temporal logic. In: Symp. on Principles of Programming Languages (2002)
15. McKinley, K., Carr, S., Tseng, C.: Improving Data Locality with Loop Transformations. ACM Trans. on Programming Languages and Systems 18(4), 424–453 (1996)
16. Necula, G.C.: Translation validation for an optimizing compiler. In: Conf. on Programming Language Design and Implementation (2000)
17. Smith, M.D., Holloway, G.: An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization
18. Triantafyllis, S., Vachharajani, M., Vachharajani, N., August, D.I.: Compiler Optimization-space Exploration. In: Int'l. Symp. on Code Generation and Optimization (2003)
19. Triantafyllis, S., Vachharajani, M., August, D.I.: Compiler Optimization-space Exploration. Journal of Instruction-Level Parallelism (2005)
20. Whitfield, D., Soffa, M.L.: An Approach to Ordering optimizing transformations. In: Symp. on Principles and Practice of Parallel Programming (1990)
21. Whitfield, D., Soffa, M.L.: An Approach for Exploring Code Improving Transformations. ACM Trans. on Programming Languages and Systems 19(6), 1053–1084 (1997)
22. Yotov, K., Li, X., Ren, G., Cibulskis, M.: A Comparison of Empirical and Model-driven optimization. In: Conf. on Programming Language Design and Implementation (2003)
23. Zhao, M., Childers, B.R., Soffa, M.L.: A Model-based Framework: an Approach for Profit-driven Optimization. In: Int'l. Symp. on Code Generation and Optimization (2005)