RULE-BASED OPTIMIZATION WITH
K-FRAMEWORK

BY
SEOKJE SEO

THESIS
Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Advisor:
Professor David Padua
Abstract

The thesis discusses pre-compiler optimization using rule-based rewriting. Our goal is to facilitate the proof of correctness of the process of program optimization. A source-to-source optimizer based on the proposed strategy can be a preprocessor to a certified compiler such as CompCert and this way it will facilitate the process of certification of a sophisticated compiler. In fact, CompCert is highly assured but it trades off the assurance with efficiency. Unlike other compilers like Intel C compiler or GNU C compiler, CompCert does not optimize aggressively. The paper will discuss optimization rules implemented using the K-framework and how much efficiency improvement achieved by use of our optimization rules.
Acknowledgments

I owe my deepest gratitude to my supervisor, David Padua, for guidance and encouragement from the beginning of the research and finishing the thesis. Also I would like to show my gratitude to Grigore Rosu and Andrei Stefanescu. They helped me to understand K-Framework and gave advices to overcome challenges facing with K-Framework. I am grateful to Jeremy Johnson and Franz Franchetti. It was great opportunity working with them. I would like to thank my friends Sanguk and Hyungjoo helped me to do proof read. Lastly, I cannot say thank enough to my parents support me more than 6 years of education at the University of Illinois.
# Table of Contents

List of Tables ........................................................................... v

List of Figures .......................................................................... vi

Chapter 1 Introduction ............................................................ 1

Chapter 2 Background ............................................................. 3
  2.1 The K-Framework ............................................................ 3
  2.2 Define Optimization Rules ................................................. 3

Chapter 3 Optimization Rules .................................................. 6
  3.1 Optimizations .................................................................. 6
  3.2 Loop Invariant Removal .................................................. 6
  3.3 Loop Unrolling ............................................................... 8
    3.3.1 Loop Unroll .............................................................. 8
    3.3.2 Unroll and Jam ......................................................... 11
  3.4 Forward Substitution ...................................................... 13
  3.5 Constant Propagation and Folding ...................................... 14
  3.6 Dead-code Elimination ...................................................... 16

Chapter 4 Evaluation .............................................................. 21
  4.1 Evaluation ..................................................................... 21
  4.2 Heuristic Search for Optimal Optimization Pass ................... 21
  4.3 Matrix-Matrix Multiplications ........................................... 22
  4.4 Vector Operations .......................................................... 26
  4.5 Dynamic Window Monitor ............................................... 27

Chapter 5 Conclusion .............................................................. 30

References .............................................................................. 31
List of Tables

4.1 Running time of difference combination of passes of Matrix-Matrix Multiplication : Naive Algorithm ....................................................... 23
4.2 Running time of difference combination of passes of Matrix-Matrix Multiplication : Switched Loop Algorithm ....................................................... 24
4.3 Result speed-ups for each vector operations ....................................................... 26
4.4 Running time of difference combination of passes of Dynamic Window Monitor ................................. 29
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example use of Syntax and Rule: number increment</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Rewriting Steps</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>Loop Invariant Removal Rules</td>
<td>7</td>
</tr>
<tr>
<td>3.2</td>
<td>Loop Unroll Rules (1)</td>
<td>9</td>
</tr>
<tr>
<td>3.3</td>
<td>Loop Unroll Rules (2)</td>
<td>10</td>
</tr>
<tr>
<td>3.4</td>
<td>Unroll and Jam Rules</td>
<td>12</td>
</tr>
<tr>
<td>3.5</td>
<td>Forward Substitution Rules</td>
<td>13</td>
</tr>
<tr>
<td>3.6</td>
<td>Constant Propagation Rules</td>
<td>14</td>
</tr>
<tr>
<td>3.7</td>
<td>Constant Folding Rules</td>
<td>15</td>
</tr>
<tr>
<td>3.8</td>
<td>Deadcode Elimination Rules (1)</td>
<td>17</td>
</tr>
<tr>
<td>3.9</td>
<td>Deadcode Elimination Rules (2)</td>
<td>18</td>
</tr>
<tr>
<td>3.10</td>
<td>Deadcode Elimination Rules (3)</td>
<td>19</td>
</tr>
<tr>
<td>3.11</td>
<td>Deadcode Elimination Rules (4)</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Naive and Switched Loop Matrix-Matrix Multiplication</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>Performance improvement by Rule-based optimization: Matrix-Matrix Multiplication</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Dynamic Window Monitor source code.</td>
<td>27</td>
</tr>
<tr>
<td>4.4</td>
<td>Dynamic Monitor Window: Optimized</td>
<td>28</td>
</tr>
<tr>
<td>4.5</td>
<td>Running time comparison between Compilers: Dynamic Window Monitor</td>
<td>28</td>
</tr>
</tbody>
</table>
Assurance and Performance are two important characteristics of a software product. They are good measures of software quality. Higher assurance can provide more accurate and confident result. This is critical for software that implements infrastructures used by hundreds of people on a daily basis and vehicles that can cause serious accidents. Higher performance can offer faster results and in this way give the system more solutions from where to choose.

Performance and assurance are particularly important for software that processes real-time data continuously and whose result impacts system’s reliability. For instance, sensor processing of a robot. A robot continuously collects sensor data to detect its current status. Then it uses this data to determine the best (or optimal) solution to achieve its goal. Since the environment around the robot keeps changing, it must find the best solution every time. If performance is not enough to process data in real-time, it is not able to cope with sudden events and this can preclude the robot from achieving its goal. Furthermore, if the solution is not to be trusted, it also reduces the likelihood of the robot achieving its goal.

Measuring performance of software is simple. All that is needed to execute the code and measure on how fast it gets a result and how much data it can be processed per unit of time. However, assurance cannot be done in the same way. From the design of program to the generation of the executable binary, a program goes through many steps. It is possible that a program’s algorithm has a problem although its implementation and execution are perfect. On the contrary, a perfect algorithm can be implemented poorly. Hence, each steps requires formal proofs that they are correct and reliable.

CompCert[3] is a C compiler that is formally verified to avoid the introduction of bugs and to make compiled executable to run as designed. However, CompCert generated executables lose performance in return for assurance because it is harder to apply aggressive optimization as compared with other compilers that are not verified.

The idea of this paper is to apply rule-based optimization on C source code before compiling it with CompCert compiler to reduce performance degradation. Rule-based optimization facilitates formal proof of the optimization. This thesis reports on several basic optimizations using the K-Framework[2][6] and ensure
the improvements they deliver.

This thesis is organized as follows: Section 2 describes basic concepts of the K-Framework and how it is used to implement rewriting optimization rules. Section 3 lists rules defined in K and explanations of Loop Invariant Removal, Loop Unrolling, Forward Substitution, Constant Propagation & Folding, and Dead-code elimination. Section 4 gives details of experiments and results on matrix-matrix multiplication, vector-vector operations, and a dynamic window monitor code used in real-time processing. The conclusions are presented in Section 5.
Chapter 2

Background

2.1 The K-Framework

The re-write rules in this paper are implemented using the K-Framework.[2] The implemented system parse C programs and re-write to using pre-defined rules of simple optimizations. This section will briefly describe the K-Framework which is a rewrite-based executable semantic framework that can be used to define and execute programming languages, type systems and formal analysis tool. The K-Framework has 3 main parts: Configurations, Computations, and Rules. Configurations organizes cells that hold the state of system. Cells can be nested and labelled so that construct the base system. Computations carry computational meanings through syntax. From this, a language to use can be defined including its grammar and computational operations. For this research, the language syntax is based on a CIL-syntax for the K-framework written by Andrei Stefanescu. The syntax was slightly modified to work with C.

2.2 Define Optimization Rules

For better understanding of the optimization rules in this paper, it is necessary to know the main concepts and components of the K-Framework. The K-Framework has many built-in functions and primitive types to help the rewriting process. As a general base type, the K-Framework provides $K$ type. The $K$ type can be used to represent any type. Also, the $KList$ type represents lists of objects of type $K$. Once the framework reads a source code, the code is converted into an association list. However, such an association list was not suitable to find pattern contains arbitrary types. For example, an if-statement can have multiple nested statements. Finding patterns involving if statement require matching arbitrary nested structures. However the framework cannot search them with an association list. To overcome this challenge, the source code in association list is transformed into a cons list with list of $K$ in $KList$. And now it is possible to find such a pattern by matching KList for arbitrary types.

Syntax and Rule are essential components to write the optimization rules. Syntax of k-framework defines a syntax of a input language. Also K-framework syntax can be used to define internal syntaxes supporting
Syntax:

| NumberType ::= Int | Float |
| K ::= NumberType "+++
function"
| k ::= "Protector" "(///" K ")///"[function]

Rule:

| NumberType+++
⇒
Protector(/// NumberType+++)/// |
| Var:Int+++)
⇒
Protector(/// Var+Int1\n)/// |
| Var:Float+++)
⇒
Protector(/// Float2Int(Var)+++\n)/// |
| K)
⇒
K |

Figure 2.1: Example use of Syntax and Rule: number increment

built-in and user-defined functionalities. For the optimization rules, Syntax is used to defines additional patterns other than regular C language to help apply each optimizations. Rules rewrite matching patterns defined by syntax conditionally with built-in data structure and functions such as a map, a type cast and etc. All of the rule-based optimizations are implemented as user-defined functions with combination of syntax and rule. Some optimizations requires multiple steps of rewriting and it is needed to make sure no other optimization intervenes. Also even in one optimization, if the optimization has multiple rewrite rules, they need to be prioritized. This challenge can be solved by combination of protecting syntax and ordered rule.

2.1 is a simple example shows how the combination works. Above rules and syntaxes will cast float constants to integer constants and increase the number by 1 when Int+++ or Float+++ is presented. When K-Framework found the pattern, it will rewrite with wrapper "Protector(***" and ")***. The wrapper needs to be unique and different from C or an unexpected pattern will be discovered. Once the target pattern is rewritten with the wrapper, only rules contain the wrapper can rewrite now. In current version of K-Framework, rules are ordered in sequence of their declarations. Thus the last rule which unwrap K will not rewrite until there is no rule above to rewrite. Following 2.2 shows how a matching pattern is rewritten.
8.5+++  
⇒ the 1st rule is applied  
Protector ///8.5+++///  
⇒ the 3rd rule is applied  
Protector ///8+++///  
⇒ the 2nd rule is applied  
Protector ///9///  
⇒ the last rule applied  
9

Figure 2.2: Rewriting Steps

The next chapter will describe the rules of each optimization.
Chapter 3
Optimization Rules

3.1 Optimizations

In this section, the rules and conditions of each optimizations. There are six optimizations implemented and each optimizations consists of collections of K-Framework rules. To be sure of correctness of each optimizations, the conditions for each rules are strict and the optimizations cover strict cases only.

The optimizations assumed the target source is in static single assignment form. The decision is made to keep rewriting rules simpler. However, it turns out that static single assignment form makes some rules more complex. This will be discussed in following sections.

The re-writing rules in each optimization is implemented as function in K. So a chain of the rules will be applied until there is no matching pattern to rewrite. In most cases, one optimization function defined by K syntax can have multiple matching pattern defined by K rule. The priority between these patterns are determined by serial occurrence in K code. If there are more than two K rules define re-writing, the rule written prior to the following rules will be considered to be applied first.

3.2 Loop Invariant Removal

Loop Invariant removal optimizes a code by moving invariants out of a loop. If the loop has an assignment statement that does assign the same value in all over iterations, the statement is unnecessary after the first iteration. By moving the statement out of the loop, it is possible to avoid the overhead. However, it it not always correct to move out a statement. To be sure, it should satisfy several conditions. The code in 3.1 is actual code for the rule in the K-framework. The rule is looking for is an assignment statement in a for-loop which ending of the loop is explicit. The code has three conditions to be satisfied.

1. The first condition checks dependencies of the assignment statement in the loop. $IsVariablesChangedOverIterationOnK$ will accept a expression of the assignment statement and a loop-body in K-List. $Index = 0$, $K-List$ will include index of the loop to the test. The function will extract all variable changed over iteration and check if the assigning expression has the changing variables.
2. The second condition checks if the statement is nested or not. If the statement is in other loop-statement or if-statement, relocation of the statement may lead to incorrect code.

3. The third condition checks if the loop will actually run or not. If not, the statement should not be moved out of the loop.

At the point that the K-framework cannot find any match in the code. The second rule of RemoveLoopInvariant will be applied and return the optimized code.

```plaintext
Figure 3.1: Loop Invariant Removal Rules

```

```
```
3.3 Loop Unrolling

Loop unroll optimization can lead to solid advantages. First, it can reduce number of branches in each iteration so reduce number of cycles. Second, it can help to achieve spatial locality of memory. At the beginning of this study it seemed that unroll transformations are easy to implement. However, while implementing the rules, more and more edge cases came out. Assumption of static single assignment helps to implementing other optimizations such as constant propagation. However, the assumption actually brought too many limitations to rules and difficulties to actual implementation.

One of the obstacles is a remainder loop with variable required to be initialized. Because of static single assignment restriction, it is not possible to duplicate the loop body. It requires extra operation to deal with dependency between iterations. For example, suppose that there is a loop accumulate integer values in an array. It uses a variable name `sum` . Because static single assignment does not allow to assign sum twice in the code, the code is forced to use if-statement to initialize sum and to accumulate it in a single assignment. Current version of static single assignment enforce function of loop unrolling can only deal dependency in a single loop but the dependency between an unrolled loop and its remainder. Because of the previous limitation, the rules must detect if unroll optimization on a for-loop will require to have a remainder loop to make sure correctness. Thus, current rule covers for-loop without conditions that includes variables. Relaxing static single assignment assumption will let rules cover more.

3.3.1 Loop Unroll

Rules for loop unroll covers two cases.

1. A for-loop without any variable in conditions that will run more than number of unrolls.
2. A for-loop without any variable in conditions that will run less than number of unrolls.

3.3 is rules that covers the cases. This optimization is consist of three major rules: `LoopUnroll`, `Static-SingleAssignment`, and `DuplicateCode`. `LoopUnroll` finds pattern that matches the cases above and rewrites it to unrolled for-loop. It checks three conditions. First, it checks if the pattern found is the inner-most loop or not. Second, it checks if there is any possible violation of static single assignment. For example, when there is an array in the loop-body, if the index of array does not contains iterator of the loop, there will be multiple assignments on the same variable in the array. Third, it checks if the unrolled loop will not generate a remainder loop or not.

`DuplicateCode` simply duplicates loop body and substitutes its loop-iterator to loop-iterator + increment. Variables that violate static single assignment will be solved by a following rule.
StaticSingleAssignment rule rewrites chunk of block to follow static single assignment principal. It collects all types of variables from their declaration and rewrite any duplicated variable assignment in the loop-body using collected data.

```
syntax KList ::= "LoopUnroll" "(" KList "," Int ")" [function]
syntax KList ::= "DuplicateCode" "(" KList "," CId "," Int ")" [function]
syntax KList ::= "StaticSingleAssignment" "(" KList ")" [function]
syntax KList ::= "StaticSingleAssignmentForLoopRemoved" "(" KList ")" [function]
syntax Int ::= "Ceil" "(" Int "," Int ")" [function]

// array in the inner most loop should contain loop index as a index rule
LoopUnroll (  
Code1:KList ,  
FT:Type FN:CId(Ps:Params) "//function", {/// FunctionIndex:Int 0,,  
CodeInFunction1:KList,,  
for (Index:Cld = Init:Int; Index < Limit:Int; Index = Index + Increment:Int)  
"//loop",  
{/// LoopIndex1:Int LoopIndex2:Int,,  
CodeInLoop:KList,,  
}/// LoopIndex1 LoopIndex2,,  
CodeInFunction2:KList,,  
}/// FunctionIndex 0,,  
Code2:KList, NumberOfUnroll:Int  
)  
=>  
LoopUnroll (  
Code1,,  
StaticSingleAssignment (  
FT FN(Ps) "//function", {/// FunctionIndex 0,,  
CodeInFunction1,,  
for (Index = Init; Index < Limit; Index = Index + NumberOfUnroll * Increment) "//unroll",  
{/// LoopIndex1 LoopIndex2,,  
DuplicateCode(CodeInLoop, Index, NumberOfUnroll -Int 1),,,  
}/// LoopIndex1 LoopIndex2,,  
CodeInFunction2,,  
}/// FunctionIndex 0  
)  
,,  
Code2, NumberOfUnroll  
)  
when (SearchKLabelOnKList(CodeInLoop, 'for'(';'(';')') /=K true) andBool (Limit >= Int NumberOfUnroll)  
andBool (notBool(StaticSingleAssignmentViolation(CodeInLoop, Index)))  
andBool (((Limit -Int Init) %Int (Increment *Int NumberOfUnroll)) ==Int 0)
```

Figure 3.2: Loop Unroll Rules (1)
rule
LoopUnroll (  
    Code1:KList,,  
    FT:Type FN:Clid(Ps:Params) /*function*/ ,, {/// FunctionIndex:Int 0,,  
    CodeInFunction1:KList,,

        for (Index:Clid = Init:Int; Index < Limit:Int; Index = Index + Increment:Int)
            "//loop",,
            {/// LoopIndex1:Int LoopIndex2:Int,,  
            CodeInLoop:KList,,
            }/// LoopIndex1 LoopIndex2,,

            CodeInFunction2:KList,,
            }/// FunctionIndex 0,,  
    Code2:KList
    ,
    NumberOfUnroll:Int
    )
) =>
LoopUnroll (  
    Code1,,  
    StaticSingleAssignmentForLoopRemoved (  
        FT FN(Ps) /*function*/ ,, {/// FunctionIndex 0,,  
        CodeInFunction1,,
        Index = 0;,  
        DuplicateCode(CodeInLoop, Index, Ceil((Limit −Int Init), Increment) −Int 1),,,
        CodeInFunction2,,
    )/// FunctionIndex 0
    )
    ,,  
    Code2,,  
    NumberOfUnroll
    )
when (SearchKLabelOnKList(CodeInLoop, 'for' (';' ';') ')') /=K true)
    andBool ((Limit −Int Init) <Int NumberOfUnroll)
    andBool (notBool(StaticSingleAssignmentViolation(CodeInLoop, Index)))

Figure 3.3: Loop Unroll Rules (2)
3.3.2 Unroll and Jam

The loop unroll optimization above cannot be applied to a for-loop that contains an array without a loop-iterator that is being unrolled due to static single assignment. For those cases, unroll and jam can be applied. When there are two loops and one is nested to the other, the outer-loop can be unrolled and duplicated nested loop can be jammed to one loop. The conditions for this rule are following,

1. An inner-loop does not contain any other for-loop which is the loop is the inner-most loop.
2. An outer loop iterates enough to be unrolled.
3. There is no static single assignment violation when loop body is duplicated.
4. When arrays are used in a body of inner loop, the arrays only can use variables alone. For example, a[i] can be used but a[i + 1] cannot.
5. An outer-loop does not make remainder loop after unrolling.
syntax KList ::= "UnrollAndJam" "(" KList "," Int ")" [function]
syntax K ::= "SearchArrayIndexOnK" "(" KList "," CId ")" [function]
rulen
UnrollAndJam(
  Code1:KList , ,
  FT:Type FN:CId(Ps:Params) "//function" , ,
  {// FunctionIndex:Int 0 , ,
    CodeInFunction1:KList , ,
    for (OuterLoopIndex:CId = OuterLoopInit:Int ; OuterLoopIndex < OuterLoopLimit:
      Int : OuterLoopIndex = OuterLoopIndex + OuterLoopIncrement:Int ) "//loop" , ,
    {// OuterLoopIndex1:Int OuterLoopIndex2:Int , ,
      for (InnerLoopIndex:CId = InnerLoopInit:Int ; InnerLoopIndex < InnerLoopLimit
        :Int : InnerLoopIndex = InnerLoopIndex + InnerLoopIncrement) "//loop" , ,
      }// InnerLoopIndex1: Int InnerLoopIndex2: Int , ,
    }// OuterLoopIndex1 OuterLoopIndex2 , ,
  }// FunctionIndex 0 , ,
  Code2:KList , ,
  NumberOfUnroll: Int)
  =>
  Code1 , ,
  FT FN(Ps) "//function" , ,
  {// FunctionIndex 0 , ,
    CodeInFunction1 , ,
    for (OuterLoopIndex = OuterLoopInit ; OuterLoopIndex < OuterLoopLimit;
      Int : OuterLoopIndex = OuterLoopIndex + NumberUnroll * Int OuterLoopIncrement)
      "//unroll" , ,
    {// OuterLoopIndex1 OuterLoopIndex2 , ,
      for (InnerLoopIndex = InnerLoopInit ; InnerLoopIndex < InnerLoopLimit;
        Int : InnerLoopIndex = InnerLoopIndex + InnerLoopIncrement) "//unroll" , ,
      }// InnerLoopIndex1 InnerLoopIndex2 , ,
    }// OuterLoopIndex1 OuterLoopIndex2 , ,
  }// FunctionIndex 0

, ,
  Code2
when
  (SearchKLabelOnKList(CodeInInnerLoop, 'for '(_;_;_') _ =/=K true) andBool ((
    OuterLoopLimit -Int OuterLoopInit) >=Int NumberUnroll)
  andBool (notBool(StaticSingleAssignmentViolation(CodeInInnerLoop, OuterLoopIndex))
      )
  andBool (ArrayIndexCheckForUnrollAndJamInSet(FindAllArraysInKList(CodeInInnerLoop)
    ))
  andBool (((OuterLoopLimit -Int OuterLoopInit) %Int (OuterLoopIncrement *Int
    NumberUnroll)) ==Int 0)
  )

Figure 3.4: Unroll and Jam Rules
3.4 Forward Substitution

Forward substitution removes unnecessary variable declaration and variable assignment from the code. This will improve performance as well as readability. However, to remove edge cases, current rule only does the substitution on base-level code that is not nested at all. The conditions are following,
1. The variable that will be substituted only occurs once or it may induce unnecessary operations.
2. The variable that will be substituted is a scalar variable. This condition is forced to prevent an expression assigned to array or pointers are substituted. Since they might need to hold some value out of the function optimized.
3. The variable that will be substituted and an expression contains the variable are not nested by any other statements to prevent edge cases such that a variable is substituted into if-statement that might not be executed.

```plaintext
syntax KList ::= "ForwardSubstitute" "(" KList ")" [function]
syntax Bool ::= "IsStatementOnBaseLevel" "(" KList ")" [function]
syntax Bool ::= "IsStatementOnBaseLevelHelp" "(" BraketIndex ")" [function]
rule ForwardSubstitute(
  Code1:KList,,
  FT:Type FN:Cld ( Ps:Params ) "//function",
  {// FunctionIndex:Int 0,,
    CodeInFunction1:KList,,
    LHS:LVal = RHS:Exp;,,
    CodeInFunction2:KList,,
  }// FunctionIndex 0,,
  Code2:KList
)
=> ForwardSubstitute(
  Code1,,
  FT FN (Ps) "//function",,
  {// FunctionIndex 0,,
    CodeInFunction1,,
    SubstituteOnKList(CodeInFunction2, LHS, RHS),,
  }// FunctionIndex 0,,
  Code2
)
when (OccuranceSearchOnKList(CodeInFunction2, LHS) ==Int 1)
  andBool (IsScalarVariable(LHS))
  andBool (IsStatementOnBaseLevel(CodeInFunction1))
rule ForwardSubstitute(Code:KList)
=> Code
```

Figure 3.5: Forward Substitution Rules
3.5 Constant Propagation and Folding

Constant Propagation and Constant Folding are called recursively to apply the rules as much as possible. Constant Propagation attests two conditions,

1. It checks if the variable is a scalar variable or not. As discussed, scalar variable is safe to remove assignment and substitute its occurrence with constant value. (Assumed the variable is not passed by reference)
2. It checks if the assignment statement is actually executed or not. The rule checks if the statement is nested or not. When the statement is in if-statement, execution is not deterministic and not safe to propagate. If the statement is in for-loop it will be taken care by loop invariant removal optimization.

Constant folding goes through all codes and finds expression that can be evaluated at the point of rewriting such as expressions that only have constant values.

```plaintext
syntax KList ::= "PropagateConstant" "(" KList ")" [function]

rule
  PropagateConstant (Code1:KList , ,
     FT:Type F:CId(Ps:Params) "//function",
     {/// FunctionIndex:Int 0 , ,
        CodeInFunction1:KList , ,
        LV:LVal = C:Const; , ,
        CodeInFunction2:KList , ,
    }/// FunctionIndex 0 , ,
     Code2:KList
  )
  =>
  PropagateConstant (FoldConstant (Code1 , ,
     FT F(Ps) "//function",
     {/// FunctionIndex 0 , ,
        CodeInFunction1 , ,
        SubstituteOnKList (CodeInFunction2 , LV , C) , ,
    }/// FunctionIndex 0 , ,
     Code2
  )
)

when IsScalarVariable(LV)

  andBool (notBool(SearchKLabelOnKList(CodeInFunction1 , ' '))

rule
  PropagateConstant (Code:KList)
  =>
  FoldConstant (Code)
```

Figure 3.6: Constant Propagation Rules
Figure 3.7: Constant Folding Rules
3.6 Dead-code Elimination

Dead-code elimination optimization deals with several cases that the part of code will not be executed at all. The rules cover following cases,

1. An empty for-statement
2. For-statement that will not be executed
3. Unnecessary if-statement generated by loop-unroll
4. Unnecessary declaration of variable
5. Scalar variable declared in function and assigned but not used anywhere.
6. if-statement with constant condition.
rule EliminateDeadCode(
    Code1: KList,
    for (I1: Instr; E: Exp; I2: Instr) S: String,
    
    /// LoopIndex1: Int LoopIndex2: Int,
    }/// LoopIndex1 LoopIndex2,
    Code2: KList
)
⇒
EliminateDeadCode(
    Code1,
    Code2
)

rule EliminateDeadCode(
    Code1: KList,
    for (Index: CId = IndexInit: Int; E: Exp; I2: Instr) S: String,
    
    /// LoopIndex1: Int LoopIndex2: Int,
    CodeInLoop1: KList,
    }/// LoopIndex1 LoopIndex2,
    Code2: KList
)
⇒
EliminateDeadCode(
    Code1,
    Code2
)
when
    BinaryOperation(SubstituteOnK(E, Index, Init)) = K 0

rule EliminateDeadCode(
    Code1: KList,
    for (Index: CId = IndexInit: Int; Condition: Exp; I: Instr) S: String,
    
    /// LoopIndex1: Int LoopIndex2: Int,
    CodeInLoop1: KList,
    C: CId = E1: Exp == IndexInit ? E2: Exp : E3: Exp,
    CodeInLoop2: KList,
    }/// LoopIndex1 LoopIndex2,
    Code2: KList
)
⇒
EliminateDeadCode(
    Code1,
    for (Index = IndexInit; Condition; I) S,
    
    /// LoopIndex1 LoopIndex2,
    CodeInLoop1,
    C = E3,
    CodeInLoop2,
    }/// LoopIndex1 LoopIndex2,
    Code2
)
when
    (SearchKLabelOnKList(CodeInLoop1, 'for' ('(';';';') ') =/= K true)
    andBool notBool ConstantFoldOnK(SubstituteOnK(E1, Index, IndexInit)) = Int
    IndexInit

Figure 3.8: Deadcode Elimination Rules (1)
Figure 3.9: Deadcode Elimination Rules (2)
rule
EliminateDeadCode(
    Code1: KList,
    if (B: Int) "//thenBlock" else "//elseBlock",
    {/// BlockIndex1: Int Parent:Int,
     CodeInThenBlock:KList,
    }/// BlockIndex1 Parent,
    {/// BlockIndex2: Int Parent,
     CodeInElseBlock:KList,
    }/// BlockIndex2 Parent,
    Code2:KList
)
⇒
EliminateDeadCode(
    Code1,
    CodeInThenBlock,
    Code2
)
when B /= Int 0

rule
EliminateDeadCode(
    Code1: KList,
    if (B: Int) "//thenBlock" else "//elseBlock",
    {/// BlockIndex1: Int Parent:Int,
     CodeInThenBlock:KList,
    }/// BlockIndex1 Parent,
    {/// BlockIndex2: Int Parent,
     CodeInElseBlock:KList,
    }/// BlockIndex2 Parent,
    Code2:KList
)
⇒
EliminateDeadCode(
    Code1,
    CodeInElseBlock,
    Code2
)
when B == Int 0

rule
EliminateDeadCode(
    Code1: KList,
    if (B: Int) "//thenBlock",
    {/// BlockIndex1: Int Parent:Int,
     CodeInThenBlock:KList,
    }/// BlockIndex1 Parent,
    Code2:KList
)
⇒
EliminateDeadCode(
    Code1,
    CodeInThenBlock,
    Code2
)
when B /= Int 0

Figure 3.10: Deadcode Elimination Rules (3)
Figure 3.11: Deadcode Elimination Rules (4)
Chapter 4

Evaluation

4.1 Evaluation

The performance improvement resulting from the application of Rule-Based Optimization implemented in the K-framework is discussed this section. The evaluation is done using a few simple vector-vector operations, matrix-matrix multiplications, and a sample code generated by Spiral[1] for the LandShark project.

The evaluation is performed on a workstation that runs on Ubuntu 15.04 that is installed on 256GB Solid State Drive. The workstation has Intel Ivy Bridge i7-3770 runs 3.4Ghz with 16 GB of RAM. CompCert 2.1 was used to compile the source codes. For comparison, we consider Intel ICC [5] (Composer XE 2013 update 3) and GNU GCC [4] (GCC 4.9.2). Since CompCert 2.1 only can compile in 32bit, we also used 32 bit mode in the other compilers. Each single target source codes is a single function that called by the benchmark program. To reduce impact of outliers, we ran the target code at least 50 millions times and averaged the running time.

4.2 Heuristic Search for Optimal Optimization Pass

In the previous chapter, we discussed optimizations. The question now is how to combine those optimizations to improve the running time. We also need to consider selection of parameters for optimizations like Loop Unroll and Unroll and Jam. To find out a combination that gives faster running time, our system automatically tries out various combination of optimizations and degrees of unrolling. The program itself is compiled by CompCert and takes object files compiled by three compilers: CompCert, ICC, and GCC.
There are five optimization passes can be chosen for a combined optimization.

0. **No Optimization applied**

1. **Loop Invariant removal pass**

2. **Loop Unrolling pass**

3. **Unroll and Jam pass**

4. **Constant Propagation and Constant Folding pass**

5. **Forward Substitution pass**

6. **Dead-code Elimination**

In result tables in following pages, 0 in passes column indicates no optimization is applied. Each time one optimization rewrite the code. Dead Code Elimination optimization is applied to get rid of unnecessary declarations and statements. However, dead code elimination is not showed in the tables.

### 4.3 Matrix-Matrix Multiplications

Matrix-Matrix Multiplication is one of popular algorithm that gets benefit from loop unroll optimization. And the algorithm as several variants to improves performance. In this evaluation, naive algorithm and switch loop algorithm.

```c
void naive(float **A, float **B, float **C)
{
    int i, j, k;
    for (i = 0; i < 200; i++) {
        for (j = 0; j < 200; j++) {
            for (k = 0; k < 200; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void switch(float **A, float **B, float **C)
{
    int i, j, k;
    for (i = 0; i < 200; i++) {
        for (k = 0; k < 200; k++) {
            for (j = 0; j < 200; j++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Figure 4.1: Naive and Switched Loop Matrix-Matrix Multiplication

Naive algorithm has triple nested loop with one instruction adding multiplication of two elements from
each matrices to the result matrix.

The Table 4.1 is a result obtained from the search program with three different compilers mentioned earlier.

<table>
<thead>
<tr>
<th>Passes</th>
<th>CompCert</th>
<th>Time(ms)</th>
<th>Spd-up</th>
<th>ICC</th>
<th>Time(ms)</th>
<th>Spd-up</th>
<th>GCC O3</th>
<th>Time(ms)</th>
<th>Spd-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>0[0]</td>
<td>30721.1</td>
<td>1.00</td>
<td></td>
<td>0[0]</td>
<td>18039.2</td>
<td>1.00</td>
<td>0[0]</td>
<td>6794.19</td>
<td>1.00</td>
</tr>
<tr>
<td>14[1]</td>
<td>30646.8</td>
<td>1.00</td>
<td></td>
<td>14[1]</td>
<td>18153</td>
<td>0.99</td>
<td>14[1]</td>
<td>6864.01</td>
<td>0.99</td>
</tr>
<tr>
<td>145[1]</td>
<td>30636.7</td>
<td>1.00</td>
<td></td>
<td>145[1]</td>
<td>18217.4</td>
<td>0.99</td>
<td>145[1]</td>
<td>6791.56</td>
<td>1.00</td>
</tr>
<tr>
<td>124[8]</td>
<td>30640.4</td>
<td>1.00</td>
<td></td>
<td>124[8]</td>
<td>18048.4</td>
<td>1.00</td>
<td>124[8]</td>
<td>6859.77</td>
<td>0.99</td>
</tr>
<tr>
<td>134[8]</td>
<td>13819.8</td>
<td>2.22</td>
<td></td>
<td>134[8]</td>
<td>9087.98</td>
<td>1.98</td>
<td>134[8]</td>
<td>6257.5</td>
<td>1.09</td>
</tr>
<tr>
<td>1345[8]</td>
<td>15981.2</td>
<td>1.92</td>
<td></td>
<td>1345[8]</td>
<td>9901.67</td>
<td>1.82</td>
<td>1345[8]</td>
<td>6314.5</td>
<td>1.08</td>
</tr>
<tr>
<td>124[10]</td>
<td>30640.4</td>
<td>1.00</td>
<td></td>
<td>124[10]</td>
<td>18048.4</td>
<td>1.00</td>
<td>124[10]</td>
<td>6859.77</td>
<td>0.99</td>
</tr>
<tr>
<td>1345[10]</td>
<td>15981.2</td>
<td>1.92</td>
<td></td>
<td>1345[10]</td>
<td>9901.67</td>
<td>1.82</td>
<td>1345[10]</td>
<td>6314.5</td>
<td>1.08</td>
</tr>
<tr>
<td>124[16]</td>
<td>30640.4</td>
<td>1.00</td>
<td></td>
<td>124[16]</td>
<td>18048.4</td>
<td>1.00</td>
<td>124[16]</td>
<td>6859.77</td>
<td>0.99</td>
</tr>
<tr>
<td>134[16]</td>
<td>13819.8</td>
<td>2.22</td>
<td></td>
<td>134[16]</td>
<td>9087.98</td>
<td>1.98</td>
<td>134[16]</td>
<td>6257.5</td>
<td>1.09</td>
</tr>
<tr>
<td>1234[16]</td>
<td>13753.1</td>
<td>2.23</td>
<td></td>
<td>1234[16]</td>
<td>9156.39</td>
<td>1.97</td>
<td>1234[16]</td>
<td>6489.15</td>
<td>1.05</td>
</tr>
</tbody>
</table>

Table 4.1: Running time of difference combination of passes of Matrix-Matrix Multiplication: Naive Algorithm

In table 4.1, the digits outside the brackets in the Passes column indicates which optimization passes were applied and the number within the bracket is the degree of unrolling.

As shown in the table, optimized source code gets a performance improvement factor of 2+. Also all the passes in five best performing codes contains 3 which is unroll and jam optimization. Because of static single assignment constraint, loop unrolling cannot be applied to the naive algorithm.

Another variant of MMM is the switched loop shown in 4.1. The only difference between these two
Table 4.2: Running time of difference combination of passes of Matrix-Matrix Multiplication: Switched Loop Algorithm.

<table>
<thead>
<tr>
<th>Passes</th>
<th>CompCert</th>
<th></th>
<th>Passes</th>
<th>ICC</th>
<th></th>
<th>Passes</th>
<th>GCC O3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time(ms)</td>
<td>Spd-up</td>
<td></td>
<td>Time(ms)</td>
<td>Spd-up</td>
<td></td>
<td>Time(ms)</td>
</tr>
<tr>
<td>0</td>
<td>14892.3</td>
<td>1.00</td>
<td>0</td>
<td>18039.2</td>
<td>1.01</td>
<td>0</td>
<td>6427.04</td>
</tr>
<tr>
<td>14</td>
<td>14865.9</td>
<td>1.00</td>
<td>14</td>
<td>18153</td>
<td>1.00</td>
<td>14</td>
<td>6380.49</td>
</tr>
<tr>
<td>145</td>
<td>14941.2</td>
<td>0.99</td>
<td>145</td>
<td>18217.4</td>
<td>1.00</td>
<td>145</td>
<td>6338.43</td>
</tr>
<tr>
<td>124</td>
<td>12536.4</td>
<td>1.19</td>
<td>124</td>
<td>18149.1</td>
<td>1.00</td>
<td>124</td>
<td>5218.53</td>
</tr>
<tr>
<td>134</td>
<td>14882.4</td>
<td>1.00</td>
<td>134</td>
<td>9858.55</td>
<td>1.84</td>
<td>134</td>
<td>6359.05</td>
</tr>
<tr>
<td>1234</td>
<td>12582.9</td>
<td>1.18</td>
<td>1234</td>
<td>10076</td>
<td>1.80</td>
<td>1234</td>
<td>5252.24</td>
</tr>
<tr>
<td>1324</td>
<td>12720.5</td>
<td>1.17</td>
<td>1324</td>
<td>9763.2</td>
<td>1.86</td>
<td>1324</td>
<td>5361.32</td>
</tr>
<tr>
<td>1245</td>
<td>12588</td>
<td>1.18</td>
<td>1245</td>
<td>18125.4</td>
<td>1.00</td>
<td>1245</td>
<td>5695.02</td>
</tr>
<tr>
<td>1345</td>
<td>15078.1</td>
<td>0.99</td>
<td>1345</td>
<td>9016.7</td>
<td>1.83</td>
<td>1345</td>
<td>6333.13</td>
</tr>
<tr>
<td>12452</td>
<td>12473.9</td>
<td>1.19</td>
<td>12452</td>
<td>9778.76</td>
<td>1.86</td>
<td>12452</td>
<td>5339.37</td>
</tr>
<tr>
<td>13245</td>
<td>12527.8</td>
<td>1.19</td>
<td>13245</td>
<td>9773.06</td>
<td>1.86</td>
<td>13245</td>
<td>5327.61</td>
</tr>
<tr>
<td>1244</td>
<td>13313.6</td>
<td>1.12</td>
<td>1244</td>
<td>18048.4</td>
<td>1.01</td>
<td>1244</td>
<td>4438.36</td>
</tr>
<tr>
<td>1344</td>
<td>14953.2</td>
<td>0.99</td>
<td>1344</td>
<td>9087.98</td>
<td>2.00</td>
<td>1344</td>
<td>6332.61</td>
</tr>
<tr>
<td>12344</td>
<td>13254.3</td>
<td>1.12</td>
<td>12344</td>
<td>9156.39</td>
<td>1.98</td>
<td>12344</td>
<td>4391.27</td>
</tr>
<tr>
<td>13244</td>
<td>13235.5</td>
<td>1.12</td>
<td>13244</td>
<td>9215.28</td>
<td>1.97</td>
<td>13244</td>
<td>4412.07</td>
</tr>
<tr>
<td>12454</td>
<td>13515.1</td>
<td>1.12</td>
<td>12454</td>
<td>18151.4</td>
<td>1.00</td>
<td>12454</td>
<td>4348.23</td>
</tr>
<tr>
<td>13454</td>
<td>15105.6</td>
<td>0.98</td>
<td>13454</td>
<td>9738.56</td>
<td>1.86</td>
<td>13454</td>
<td>6351.27</td>
</tr>
<tr>
<td>123454</td>
<td>13273.4</td>
<td>1.12</td>
<td>123454</td>
<td>9466.51</td>
<td>1.92</td>
<td>123454</td>
<td>4481.95</td>
</tr>
<tr>
<td>132454</td>
<td>13341.7</td>
<td>1.12</td>
<td>132454</td>
<td>9115.77</td>
<td>1.99</td>
<td>132454</td>
<td>4324.12</td>
</tr>
<tr>
<td>1248</td>
<td>13808</td>
<td>1.08</td>
<td>1248</td>
<td>18068.6</td>
<td>1.00</td>
<td>1248</td>
<td>4151.67</td>
</tr>
<tr>
<td>1384</td>
<td>15027.2</td>
<td>0.99</td>
<td>1384</td>
<td>9055.74</td>
<td>2.00</td>
<td>1384</td>
<td>6360.99</td>
</tr>
<tr>
<td>12348</td>
<td>13789</td>
<td>1.09</td>
<td>12348</td>
<td>9071.18</td>
<td>2.00</td>
<td>12348</td>
<td>4182.18</td>
</tr>
<tr>
<td>13248</td>
<td>13618.9</td>
<td>1.09</td>
<td>13248</td>
<td>9088.29</td>
<td>2.00</td>
<td>13248</td>
<td>4154.17</td>
</tr>
<tr>
<td>12458</td>
<td>13656.2</td>
<td>1.09</td>
<td>12458</td>
<td>18031.4</td>
<td>1.01</td>
<td>12458</td>
<td>4156.01</td>
</tr>
<tr>
<td>13458</td>
<td>14915.2</td>
<td>1.00</td>
<td>13458</td>
<td>9079.51</td>
<td>2.00</td>
<td>13458</td>
<td>6447.37</td>
</tr>
<tr>
<td>123458</td>
<td>13930.9</td>
<td>1.07</td>
<td>123458</td>
<td>9132.63</td>
<td>1.99</td>
<td>123458</td>
<td>4162.12</td>
</tr>
<tr>
<td>132458</td>
<td>13671.9</td>
<td>1.09</td>
<td>132458</td>
<td>9104.68</td>
<td>1.99</td>
<td>132458</td>
<td>4243.23</td>
</tr>
<tr>
<td>12410</td>
<td>13703.1</td>
<td>1.08</td>
<td>12410</td>
<td>18048.4</td>
<td>1.01</td>
<td>12410</td>
<td>4157.6</td>
</tr>
<tr>
<td>13410</td>
<td>14890.8</td>
<td>1.00</td>
<td>13410</td>
<td>9257.96</td>
<td>1.96</td>
<td>13410</td>
<td>6592.98</td>
</tr>
<tr>
<td>123410</td>
<td>13722</td>
<td>1.08</td>
<td>123410</td>
<td>9238.85</td>
<td>1.96</td>
<td>123410</td>
<td>4313.29</td>
</tr>
<tr>
<td>132410</td>
<td>13779.2</td>
<td>1.08</td>
<td>132410</td>
<td>9486.88</td>
<td>1.91</td>
<td>132410</td>
<td>4157.01</td>
</tr>
<tr>
<td>124510</td>
<td>13703.5</td>
<td>1.08</td>
<td>124510</td>
<td>18031.4</td>
<td>1.01</td>
<td>124510</td>
<td>4218.37</td>
</tr>
<tr>
<td>134510</td>
<td>14893.6</td>
<td>1.00</td>
<td>134510</td>
<td>9228.36</td>
<td>1.97</td>
<td>134510</td>
<td>6728.3</td>
</tr>
<tr>
<td>1234510</td>
<td>13704.8</td>
<td>1.08</td>
<td>1234510</td>
<td>9331.42</td>
<td>1.95</td>
<td>1234510</td>
<td>4252.05</td>
</tr>
<tr>
<td>1324510</td>
<td>13869.2</td>
<td>1.07</td>
<td>1324510</td>
<td>10461.2</td>
<td>1.74</td>
<td>1324510</td>
<td>4163.47</td>
</tr>
</tbody>
</table>

algorithms is the order of the loops. While naive algorithm has i-j-k loops, switched loop algorithm has i-k-j loops. This algorithm runs faster than naive due to better memory locality that benefits cache hits. This improves performance about 200% without additional optimization.

In the search results, the speed up is 1.2 against the original source code. Because MMM with switched loop is already 2 times faster than naive MMM, the speed up is not as dramatic as previous one. However, the running time of optimized code from naive algorithm is close to the running time of optimized switched algorithm.

One thing that noticeable is that unroll with degree two is faster than unroll with higher degrees. It turns out small number of degree performs better. Following degree two, degree four and degree eight comes
Comparing optimization to other compiler, ICC compiled binary get speed up more than 50% on naive MMM and only 1 2% of speed up on switched loop MMM. GCC compiled binaries run even faster than ICC. It runs more than 2 times faster than ICC and CompCert. And GCC compiled binaries performed almost the same on both naive and switched loop algorithm. Figure 4.2 shows a comparison of running time between different compilers. Regardless which compiler is used, it was able to see performance improvement was achieved from rule-based optimization especially by loop unrolling.
Vector operations can get benefits from the optimizations. Since it has similar memory locality as Matrix-Matrix Multiplication, it should achieve improvement from loop unrolling optimization. As the results from previous section, loop unroll optimization is the most important it. Several basic vector operations were used for the evaluation: vector addition, scatter assignment, dot product, and multiple vector addition/dot product. Since the target codes only consist of loops, only optimization effective should be loop unrolling in this case. However, it is still worth to search and try different degree of unrolling.

As expected unroll optimization impact performance on the operation. For CompCert compiled binaries, higher degree gives more performance in the most cases. ICC and GCC compiled binaries mostly perform better than CompCert. However, ICC didn’t get any speed up for some cases. It seems that unrolled code interrupts its compiler optimization. The performance improvement result from optimization of vector operations are listed in the table from Figure 4.3. Since only loop unroll optimization achieves better performance. Only the best running time improvement achieved are listed below. Depends on a vector operation, the speed-ups vary from factor of 1.23 to factor of 1.63 for CompCert.
4.5 Dynamic Window Monitor

Dynamic Window Monitor (DWMonitor) is a code that my optimizations are most focus on to improve. DWMonitor is generated by Spiral [1] to process digital signal from sensors. Since the original code generated is not SSA, the code is transformed to SSA. Figure 4.3 are two version of DWMonitor. 4.3b has extra lines of code to enforce static single assignment. Hence, it is expected to have performance degradation. After all of six optimization explained in previous section is applied, the code resulted is much more efficient and does not contain any loop as showed in Figure 4.4.

```c
int dwmonitor(double *X, double *D) {
    double s5, s8, s7, s6, q3, q4, w1, s4, s1;
    int i5, i3;

    s5 = 0.0;
    s8 = X[0];
    s7 = 1.0;
    for (i5 = 0; i5 < 2; i5++) {
        s6 = (s7*D[i5]);
        s5 = (s5 + s6);
        s7 = (s7*s8);
    }
    s1 = 0.0;
    for (i3 = 0; i3 < 1; i3++) {
        q3 = X[(i3 + 1)];
        q4 = X[(3 + i3)];
        w1 = (q3 - q4);
        s4 = (((w1 >= 0)) ? (w1) : (-w1));
        s1 = (((s1 >= s4)) ? (s1) : (s4));
    }
    return ((s1 >= s5));
}
```

(a) Original

```c
int dwmonitor(double *X, double *D) {
    double q3, q4, s1, s4, s5, s6, s7, s8, w1, z1, z5, z7;
    int w2, i5, i3;
    s8 = X[0];
    for (i5 = 0; i5 <= 2; i5++) {
        z5 = (i5 == 0) ? 0.0 : s5;
        z7 = (i5 == 0) ? 1.0 : s7;
        s4 = (z7 * D[i5]);
        s5 = (z5 + s4);
        s7 = (z7 * s8);
    }
    for (i3 = 0; i3 < 1; i3++) {
        z1 = (i3 == 0) ? 0.0 : s1;
        q3 = X[(i3 + 1)];
        q4 = X[(i3 + 3)];
        w1 = (q3 - q4);
        s6 = (((w1 >= 0)) ? (w1) : (-w1));
        s1 = (((s1 >= s6)) ? (z1) : (s6));
    }
    w2 = ((s1 >= s5));
    return w2;
}
```

(b) Single Static Assignment Form

Figure 4.3: Dynamic Window Monitor source code.
```c
int dwmonitor ( double * X, double * D ) {
    double s618 ;
    double w117 ;
    double s1 ;
    double s6 ;
    double w1 ;
    s6 = (((w1 >= 0) ? w1 : (− w1)) ;
    s1 = (((s1 >= s6) ? 0.0 : s6) ;
    s618 = (((w117 >= 0) ? w117 : (− w117)) ;
    return (((s1 >= s618) ? s1 : s618) >= ((D [ 0 ] + (X [ 0 ] * D [ 1 ])) + ((X [ 0 ] * X [ 0 ] ) * D [ 2 ])))) ;
}
```

Figure 4.4: Dynamic Monitor Window: Optimized

Table 4.4 is a benchmark result of DWMonitor. The outcome from binaries compiled by CompCert was closed to expectation. However, binaries from ICC and GCC were different. For both cases, transforming to SSA form benefits the most and the impact from the optimization was negligible. It seems that single static assignment form helps ICC and GCC to optimize themselves. On the other hand, the optimization improved performance for CompCert. Although, transforming to SSA form influence performance, improvement from the optimization overcome the degradation. In case of CompCert binaries, SSA version is 25% slower than original code generated from Spiral but the optimized binaries are about 20% faster than the original. The figure 4.5 shows the running time of DWMonitor of each compilers.

Figure 4.5: Running time comparison between Compilers: Dynamic Window Monitor

28
<table>
<thead>
<tr>
<th>Passes</th>
<th>CompCert</th>
<th>Spd-up</th>
<th>ICC</th>
<th>Spd-up</th>
<th>GCC O3</th>
<th>Spd-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>0[0]</td>
<td>8.42943</td>
<td>0.75</td>
<td>3.23041</td>
<td>1.14</td>
<td>5.9892</td>
<td>1.09</td>
</tr>
<tr>
<td>14[1]</td>
<td>8.08528</td>
<td>0.78</td>
<td>3.24486</td>
<td>1.14</td>
<td>5.9671</td>
<td>1.10</td>
</tr>
<tr>
<td>145[1]</td>
<td>9.10771</td>
<td>0.69</td>
<td>3.19835</td>
<td>1.16</td>
<td>6.0036</td>
<td>1.09</td>
</tr>
<tr>
<td>124[2]</td>
<td>7.45083</td>
<td>0.85</td>
<td>3.19484</td>
<td>1.16</td>
<td>6.0064</td>
<td>1.09</td>
</tr>
<tr>
<td>134[2]</td>
<td>8.17355</td>
<td>0.77</td>
<td>3.1939</td>
<td>1.16</td>
<td>6.0301</td>
<td>1.09</td>
</tr>
<tr>
<td>1234[2]</td>
<td>7.28461</td>
<td>0.87</td>
<td>3.41879</td>
<td>1.08</td>
<td>5.9633</td>
<td>1.10</td>
</tr>
<tr>
<td>1245[2]</td>
<td>8.01483</td>
<td>0.79</td>
<td>3.45166</td>
<td>1.07</td>
<td>5.9747</td>
<td>1.10</td>
</tr>
<tr>
<td>1345[2]</td>
<td>9.44808</td>
<td>0.67</td>
<td>3.22016</td>
<td>1.15</td>
<td>6.0050</td>
<td>1.09</td>
</tr>
<tr>
<td>12345[2]</td>
<td>8.02866</td>
<td>0.79</td>
<td>3.19723</td>
<td>1.16</td>
<td>5.9810</td>
<td>1.10</td>
</tr>
<tr>
<td>13245[2]</td>
<td>8.06633</td>
<td>0.78</td>
<td>3.19901</td>
<td>1.15</td>
<td>6.001</td>
<td>1.09</td>
</tr>
<tr>
<td>124[4]</td>
<td>5.00875</td>
<td>1.26</td>
<td>3.20245</td>
<td>1.15</td>
<td>5.9225</td>
<td>1.11</td>
</tr>
<tr>
<td>134[4]</td>
<td>7.99487</td>
<td>0.79</td>
<td>3.20331</td>
<td>1.15</td>
<td>5.9966</td>
<td>1.09</td>
</tr>
<tr>
<td>1234[4]</td>
<td>5.00703</td>
<td>1.26</td>
<td>3.21341</td>
<td>1.15</td>
<td>6.3924</td>
<td>1.03</td>
</tr>
<tr>
<td>1324[4]</td>
<td>5.22657</td>
<td>1.21</td>
<td>3.24897</td>
<td>1.14</td>
<td>5.9368</td>
<td>1.11</td>
</tr>
<tr>
<td>1245[4]</td>
<td>5.15368</td>
<td>1.23</td>
<td>3.19712</td>
<td>1.16</td>
<td>6.8421</td>
<td>0.96</td>
</tr>
<tr>
<td>1345[4]</td>
<td>9.44041</td>
<td>0.67</td>
<td>3.22107</td>
<td>1.15</td>
<td>6.0043</td>
<td>1.09</td>
</tr>
<tr>
<td>12345[4]</td>
<td>5.15021</td>
<td>1.23</td>
<td>3.21041</td>
<td>1.15</td>
<td>6.3840</td>
<td>1.03</td>
</tr>
<tr>
<td>124[8]</td>
<td>5.015</td>
<td>1.26</td>
<td>3.20293</td>
<td>1.15</td>
<td>6.3728</td>
<td>1.03</td>
</tr>
<tr>
<td>134[8]</td>
<td>7.9945</td>
<td>0.79</td>
<td>3.21803</td>
<td>1.15</td>
<td>5.9643</td>
<td>1.10</td>
</tr>
<tr>
<td>1234[8]</td>
<td>5.06191</td>
<td>1.25</td>
<td>3.1973</td>
<td>1.16</td>
<td>6.0998</td>
<td>1.08</td>
</tr>
<tr>
<td>1324[8]</td>
<td>5.01671</td>
<td>1.26</td>
<td>3.2</td>
<td>1.15</td>
<td>5.9305</td>
<td>1.11</td>
</tr>
<tr>
<td>1245[8]</td>
<td>5.36209</td>
<td>1.18</td>
<td>3.23816</td>
<td>1.14</td>
<td>6.5631</td>
<td>1.00</td>
</tr>
<tr>
<td>1345[8]</td>
<td>9.06273</td>
<td>0.70</td>
<td>3.19741</td>
<td>1.16</td>
<td>6.0853</td>
<td>1.08</td>
</tr>
<tr>
<td>12345[8]</td>
<td>5.65989</td>
<td>1.12</td>
<td>3.60122</td>
<td>1.03</td>
<td>6.3404</td>
<td>1.04</td>
</tr>
<tr>
<td>13245[8]</td>
<td>5.1859</td>
<td>1.22</td>
<td>3.20893</td>
<td>1.15</td>
<td>7.2853</td>
<td>0.90</td>
</tr>
<tr>
<td>124[10]</td>
<td>5.01196</td>
<td>1.26</td>
<td>3.40805</td>
<td>1.08</td>
<td>5.9985</td>
<td>1.09</td>
</tr>
<tr>
<td>134[10]</td>
<td>8.03207</td>
<td>0.79</td>
<td>3.64521</td>
<td>1.01</td>
<td>6.1498</td>
<td>1.07</td>
</tr>
<tr>
<td>1234[10]</td>
<td>5.4057</td>
<td>1.17</td>
<td>3.19812</td>
<td>1.16</td>
<td>6.5760</td>
<td>1.00</td>
</tr>
<tr>
<td>1324[10]</td>
<td>5.32703</td>
<td>1.19</td>
<td>3.20917</td>
<td>1.15</td>
<td>6.6330</td>
<td>0.99</td>
</tr>
<tr>
<td>1245[10]</td>
<td>5.17842</td>
<td>1.22</td>
<td>3.44303</td>
<td>1.07</td>
<td>6.3598</td>
<td>1.03</td>
</tr>
<tr>
<td>1345[10]</td>
<td>9.11192</td>
<td>0.69</td>
<td>3.19283</td>
<td>1.16</td>
<td>6.2158</td>
<td>1.06</td>
</tr>
<tr>
<td>12345[10]</td>
<td>5.15145</td>
<td>1.23</td>
<td>3.2508</td>
<td>1.14</td>
<td>6.3563</td>
<td>1.03</td>
</tr>
<tr>
<td>13245[10]</td>
<td>5.16251</td>
<td>1.22</td>
<td>3.19656</td>
<td>1.16</td>
<td>6.4152</td>
<td>1.02</td>
</tr>
<tr>
<td>Raw</td>
<td>6.31796</td>
<td>1.00</td>
<td>3.69458</td>
<td>1.00</td>
<td>6.5633</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 4.4: Running time of difference combination of passes of Dynamic Window Monitor.
Chapter 5

Conclusion

This thesis, I presented a rule-based optimization that rewrites C source code. Also it described how the optimizations are written in the K-Framework, and then how to make it to combine Syntaxes and Rules. Then this strategy is evaluated over simple matrix operations and vector operation in single static assignment form as well as more real-life dynamic windows monitor code generated by Spiral in both with and without SSA. The evaluation shows that the optimization can greatly improve performance on matrix and vector operations. In the case of DWMonitor, it improves 104% of performance comparing to the native source code without SSA. However, it achieves more than 1.5 times speed-up from SSA source code.

The rules presented work for single static assignment. However, a source code in SSA form can be inefficient than a native code. In the future, it would be good idea to investigate on rewrite optimization on codes in non-SSA form. Then it could be applied to more codes. Also not discussed here and left for future work are formal proofs of the rules.
References


