2008

Model-driven search-based loop fusion optimization for handwritten code

Pamela Bhattacharya
Louisiana State University and Agricultural and Mechanical College

MODEL-DRIVEN SEARCH-BASED LOOP FUSION OPTIMIZATION FOR HANDWRITTEN CODE

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Systems Science

in

The Department of Computer Science

by

Pamela Bhattacharya
B.Tech. in Information Technology, West Bengal University of Technology, India, 2006

August 2008

Dedicated to Ma, Babai, Mamon, Taun, Shonai, Dadu and Sudipto

Acknowledgements

At times our own light goes out and is rekindled by a spark from another person.
Saying thank you to that person is more than good manners. It is good spirituality.

My parents, uncle, aunt and my grandparents are the ones who encouraged me to study. Without their constant mental support, love, and blessings, I am pretty sure I would have never made it. Thank you Ma, Baba, Mamon, Taun, Shonai and Dadu. Sudipto is an angel in my life without whose constant inspiration, affection and patience, achieving anything in this far-off land would have been next to impossible for me. My sisters, Dona, Pom and Tuklu have all been such wonderful siblings to have. I have learnt so much from you guys. I love you so much. Thank you too for being there. Dr. Gerald Baumgartner has been both an advisor and guardian to me. He is a patient teacher with a gentle character. He astounds me with his diverse knowledge in almost any field along with his thorough and consistent analysis of our research. Without his support, it would have been really difficult for me to achieve my goals. Thank you so much Dr. Baumgartner. I would also like to thank Dr. J. Ramanujam and Dr. Rahul Shah for serving on my committee. I would like to thank my all friends at Baton Rouge. I would like to specially express my gratitude towards Santanu Da and Sahana Di, for their caring attitude and immense help, which made my life in this alien land so much easier.

Table of Contents

List of Tables

List of Figures

Abstract

The *Tensor Contraction Engine* (TCE) is a compiler that translates high-level, mathematical tensor contraction expressions into efficient, parallel Fortran code. A pair of optimizations in the TCE, the fusion and tiling optimizations, have proven successful for minimizing disk-to-memory traffic for dense tensor computations. While other optimizations are specific to tensor contraction expressions, these two model-driven search-based optimization algorithms could also be useful for optimizing handwritten dense array computations to minimize disk to memory traffic. In this thesis, we show how to apply the loop fusion algorithm to handwritten code in a procedural language.

While in the TCE the loop fusion algorithm operated on high-level expression trees, in a standard compiler it needs to operate on abstract syntax trees. For simplicity, we use the fusion algorithm only for memory minimization instead of for minimizing disk-to-memory traffic. Also, we limit ourselves to handwritten, dense array computations in which loop bounds expressions are constant, subscript expressions are simple loop variables, and there are no common subexpressions.

After type-checking, we canonicalize the abstract syntax tree to move side effects and loop-invariant code out of larger expressions. Using dataflow analysis, we then compute reaching definitions and add use-def chains to the abstract syntax tree. After undoing any partial loop fusion, a generalized loop fusion algorithm traverses the abstract syntax tree together with the use-def chains. Finally, the abstract syntax tree is rewritten to reflect the loop structure found by the loop fusion algorithm.

We outline how the constraints on loop bounds expressions and array index expressions could be removed in the future using an algebraic cost model and an analysis of the iteration space using a polyhedral model.

1. Introduction

In computing, optimization is the process of modifying a system to make some aspect of it work more efficiently or use fewer resources. It also aims at making the existing methods more computationally efficient while retaining the same functionality. Compiler optimization is the process of modifying the output of a compiler to minimize or maximize some attribute of an executable program. In general, the process of compiler optimization are traditional heuristics which can be *empirical run-time search* or *model-driven compile-time search*. Empirical program optimizers estimate the values of key optimization parameters by generating different program versions and running them on the actual hardware to determine which values give the best performance. In contrast, conventional compilers at best uses models of programs and machines to choose these parameters, which is referred to as the Model-driven approach.

Our research focusses on developing a model-driven and search-based approach for optimizations for handwritten code using loop fusion techniques. Search-based optimizations are techniques that work by selecting the method amongst many, such that the chosen one is more computationally effectual. Loop fusion, as discussed later in details in Section 4.1, is a high-level program transformation for compiling efficient code for modern architectures and we use it to minimizing intermediate storage or disk to memory traffic.

The Tensor Contraction Engine (TCE) project developed a program synthesis system to facilitate the rapid development of high-performance parallel programs for a class of scientific computations. These computationally intensive components are expressible as a set of tensor contractions and are often encountered in electronic structure calculations in chemistry and physics. These computations also require a large amount of storage space, which can be beyond the capacity of a single computer's memory. To overcome this difficulty, the TCE applied the fusion module, which uses of loop fusion optimization to find an optimal evaluation for the tensor expression. This module also made evaluation decisions about: the structure of *for* loops, the array placement in memory or on disk, the array distribution on nodes and the recomputation of intermediate values to reduce the memory needs.

Similar to quantum computations, which can require immense storage space, general-purpose

languages should be able to handle codes which demand equally massive storage space. One of the major challenges in this respect, is to ease programming by providing the programmer an abstraction of memory as an infinite store. This is especially helpful for dense linear-algebra code and other dense array computations. Since search-based optimization algorithms were successfully employed in the TCE for memory minimization, we generalize these ideas to handwritten code structures that entail large amounts of storage space.

The fusion module in the TCE was applied on tensor expressions after they were minimized using a set of algebraic transformations, such as associativity and distributivity, to minimize the number of operations needed to compute the expression. On the other hand, we apply our fusion transformations on the Abstract Syntax Tree (Section 4.1), generated from the user's code. Our algorithm for memory minimization is applied on the information we have regarding the loop constraints and dependencies after our Data Flow Analysis (Section 4.1) across the entire Abstract Syntax Tree.

We do have both assumptions and limitations in our algorithms and those are discussed in the Algorithm (Chapter 5) and possible future work sections (Chapter 7) of the thesis, respectively. The thesis is structured in the following way. Chapter 2 covers previous work related to the fusion algorithms in compiler optimizations. In Chapter 3 we define in details the problem we are looking at and the major challenges. Chapter 4 discusses definitions and explains terminologies related to our study. The algorithmic specifics and our approach to the solution of the problem discussed in Chapter 3 are dealt with in Chapter 5. With the help of several examples in Chapter 6, we illustrate our algorithm for a better understanding of the solution we provide. We conclude in Chapter 7 with some remarks about of the significance of the work presented in this thesis and with an outlook at potential extensions of our work in near future.

2. Related Work

Enhancing parallelism and locality in computationally intensive programs involving arrays to speed up target programs often running on multiprocessor systems has been a significant topic of study amongst the computer scientists whose concentration is broadly in the field of compiler optimization. As discussed in the Introduction, Chapter 1 of this thesis, our study is mostly an extension and generalization of existing memory minimization work in the TCE. The TCE was inclined more towards applications in quantum chemistry, while ours is focussed more on applications in general-purpose languages.

Before the inception of the TCE, there were various advancements in developing high-performance parallel software for computationally intensive scientific applications. Parallel languages like ZPL [45] and parallel extensions to general-purpose sequential languages like C [11], Java [49] and Fortran [39] are worth mention in this respect. Parallel libraries and problem solving environments like SCALAPACK [12], PLAPACK [4], UHFFT [36], Global Arrays [38], OVERTURE [10], Cactus [42], PETSc [5], Broadway [23] and domain-specific synthesis from high-level specification, such as SPIRAL for the signal processing domain [37].

Since our work is an builds on the TCE, it is important at this point to discuss their study, which broadly formed the skeleton to our research. An overview of the TCE as a program synthesis tool was presented in three different papers by Gerald Baumgartner et al. in [6–8]. The TCE program synthesis tool contains a suite of optimization algorithms. Fusion algorithms were addressed by the team of TCE researches in several of their papers. A static memory minimization algorithm was introduced in [14, 28]. While the first step in the TCE towards memory minimization was the Fusion Module, the second step towards finding an optimal evaluation was the Tiling Module. This module makes evaluation decisions about: tiling arrays and selecting tile sizes and the ordering of statements in the evaluation. Based on a realistic cost model, the tiling module is responsible for selecting the optimal evaluation for the given tensor expression. The tiling algorithms are presented in [26, 27]. The memory minimization was further extended for the space-time trade-off optimization in [15]. In the space-time trade-off approach, the fusion optimization and tiling optimization are decoupled in the sense that the fusion optimization does not include any tiling or

disk access information in the cost model. The data-locality algorithms were addressed, focusing mainly on the minimization of the amount of data transferred between cache and memory in [16,17]. The integrated fusion and tiling approach were further studied in [9], where a slow, optimal and a fast, efficient algorithms were proposed. Another integrated approach, where tiling and disk access information is included in the cost model was referred to in [16].

Much work has been done on improving locality and parallelism by loop fusion in [25, 33, 44]. The contraction of arrays into scalars through loop fusion is studied in [21] but is motivated by data locality enhancement and not memory reduction. Loop fusion in the context of delayed evaluation of array expressions in APL (synonym for *A P*rogramming *L*anguage is an array programming language) programs is discussed in [22], but their work is also not aimed at minimizing array sizes; in addition, they consider loop fusion without considering any loop reordering.

Strout et al. [46] present a technique for determining the minimum amount of memory required for executing a perfectly nested loop with a set of constant-distance dependence vectors. Fraboulet et al. [20] use loop alignment to reduce memory requirement between adjacent loops by formulating the one-dimensional version of the problem as a network flow problem.

Pike and Hilfinger [40] apply tiling and fusion to a set of consecutive perfectly nested loops (each containing one statement) of the same nesting depth.

Considerable research on loop transformations for locality in nested loops has been reported in the literature [18, 19, 34, 47].

Frameworks for handling imperfectly nested loops have been presented in [2] and [32]. Ahmed et al. [2] have developed a framework that embeds an arbitrary collection of loops into an equivalent perfectly nested loop that can be tiled; this allows a cleaner treatment of imperfectly nested loops.

Lim et al. developed a framework based on affine partitioning and blocking to reduce synchronization and improve data locality in [32]. Specific issues of locality enhancement, I/O placement and optimization, and automatic tile size selection have not been addressed in the works that can handle imperfectly nested loops [2], [32]. The approach undertaken in this project bears similarities to some projects in other domains, such as the SPIRAL project, which is aimed at the design of a system to generate efficient libraries for digital signal processing algorithms [24], [41], [48].

All these efforts use search-based approaches for performance tuning of codes. A comparison of model-based and search-based approaches for matrix-matrix multiplication is reported in [50]. In addition, motivated by the difficulty of detecting and optimizing matrix operations hidden in array subscript expressions within loop nests, several projects have worked on efficient code generation

from high-level languages such as MATLAB and Maple [13], [43], [35], [1].

While our effort shares some common goals with several of the projects mentioned above, there are also significant differences in both the problems we look at and in the solution we provide thereof.

3. Problem Statement

As mentioned in earlier chapters, over the last few years, a collaborative project between computer scientists and quantum chemists enabled the development of a program transformation system, called the Tensor Contraction Engine (TCE), to automatically transform from a high-level specification of computations (expressed as complex tensor contractions) into optimized parallel programs. Apart from the TCE, there have been significant advances in recent years in developing frameworks for compiler optimizations for parallelism and locality as discussed in detail in the Related Work chapter (Chapter 2).

A pair of optimizations in the TCE, the fusion and tiling optimizations, have proven successful for minimizing disk to memory traffic for dense tensor computations. While other optimizations are specific to tensor contraction expressions, these two optimizations could also be useful for optimizing handwritten dense array computations.

The fusion and tiling optimizations can be used for different purposes. E.g., the fusion optimization by itself can be used for minimizing memory requirements for intermediate results. With a different cost model and together with the tiling optimization, these optimizations can be used for minimizing disk to memory traffic. The loop fusion optimization finds optimal loop structures that minimize both disk I/O and memory requirements using a fairly course cost model. The result of the fusion optimization is a set of candidate loop structures in which intermediates that needs to be stored to disk are identified. The tiling optimization then uses a very fine-grained, precise cost model to tile the candidate loop structures to find the code that minimizes disk I/O while staying within the constraints of the available memory.

Let us consider an example to help us understand the problem we address in this thesis. Suppose a programmer codes in a general purpose language to find solution to the summation represented below [31]:

$$S_{abij} = \sum_{cdefkl} A_{acik} \times B_{befl} \times C_{dfjk} \times D_{cdel} \tag{3.1}$$

The equation used above is a common of its type in tensor contractions. The typical index ranges are on the order of tens to a few thousands. If this expression is directly translated to code (with ten nested loops, for indices a - l), the total number of arithmetic operations required will be $4 \times N^{10}$, if the range of each index a-l is N. Instead, the same expression can be rewritten by using associative and distributive laws as follows [31]:

$$S_{abij} = \sum_{ck} \left(\sum_{df} \left(\sum_{el} B_{befl} \times D_{cdel} \right) \times C_{dfjk} \right) \times A_{acik} \tag{3.2}$$

For the above computation using a computer program, we require temporary arrays as follows [31]:

$$T1_{bcdf} = \sum_{el} B_{befl} \times D_{cdel} \tag{3.3}$$

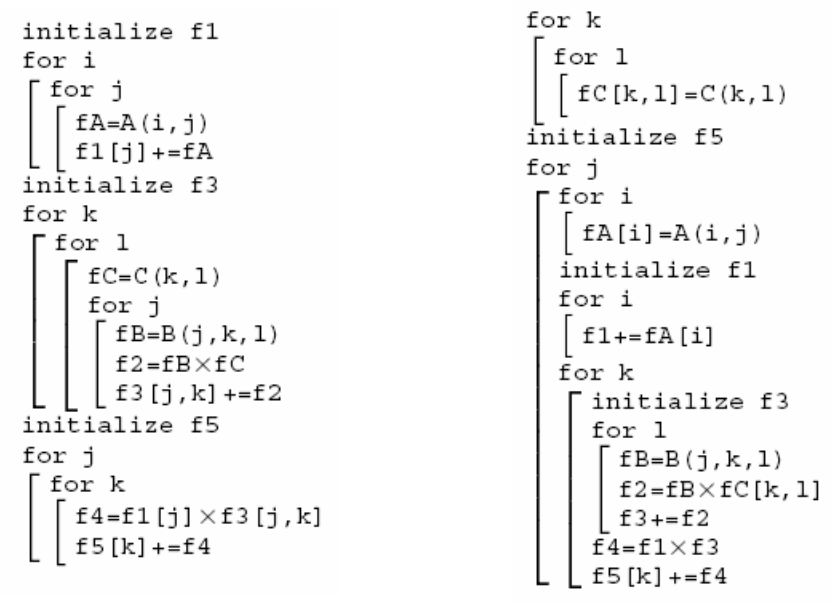
$$T2_{bcjk} = \sum_{df} T1_{bcdf} \times C_{dfjk} \tag{3.4}$$

$$S_{abij} = \sum_{ck} T2_{bcjk} \times A_{acik} \tag{3.5}$$

For the unfused version of the code for the above example will require additional space to store the temporary arrays $T1$ and $T2$. Often the space requirements for the temporary arrays poses serious problems. It has also been found that these intermediate arrays become so large that they even do not fit on disk.

This problem of storage was studied by C. Lam et al. in [28–30] and suggested an efficient way to reduce memory requirement for the computation in terms of potential loop fusions. The idea used was that when one loop nest produces an intermediate array which is consumed by another loop nest, fusing the two loop nests allows the dimension corresponding to the fused loop to be eliminated in the array. This results in a smaller intermediate array and, thus, reduces the memory requirements. For the example considered, the application of fusion is illustrated in Fig. 3.1. This

demonstrates how $T1$ can be reduced to a scalar and $T2$ to a two-dimensional (2-D) array, without changing the number of operations.



Partially Fused Code (a)

Maximally Fused Code (b)

FIGURE 3.1: Partially and Maximally Fused Code

We essentially use the same memory minimization algorithm and generalize it for our use. The following instances are what makes our work more challenging:

- The input for the TCE is a simple tensor expression with just binary and unary operations, in our case it is a piece of code written in general purpose language which has more to it. It includes assignments, loops of various types, namely, *for* and *while*, conditional blocks other than binary and unary operations.

- An expression tree in the TCE is at a much higher level of abstraction and mathematically much simpler, it is easy for the compiler to have more information about potential fusion. This not only makes optimization easier but also removes the need for reconstruction of the provided input. In order to apply fusion algorithm we need to discover relevant information, which in our case we do by employing standard compiler Data Flow Analysis techniques. This is important because handwritten code has much more constraints than the TCE input tensor expressions had, which again makes optimization harder.

- A cost model is used to measure the memory usage as a result of fusion at a specific node. This is helpful when we need to choose the most optimized solution amongst different fusion solutions, each of which have different memory usage. Developing cost models for handwritten code written in general purpose languages is not always possible, and our goal is to develop cost models that cover a large range of code reasonable accurately.

- Since handwritten code do not follow any strict rules and can use arbitrary loop structures, very often loop dependencies are encountered. This prohibits loop interchanges, loop fusion, or tiling and this calls for more perfect designs in the memory minimization algorithm so that it can handle it all.

As a first step towards this goal of running the memory minimization algorithm on an arbitrary code structure, we concentrate on loop fusion optimization for a simplified problem. This thesis demonstrates that it is possible to use search-based fusion algorithm already employed in the TCE to handwritten codes, or in other words on Abstract Syntax Tree. The primary benefit of this to a programmer is that it gives the abstraction of working with intermediate arrays of arbitrary size. We do have a few limitations and assumptions and the details are discussed in the conclusion chapter of this thesis (Chapter 7).

4. Definitions

In order to explain the algorithm and for a better understanding of the preliminaries of our optimization techniques, it is important at this point of the thesis, to provide a deeper understanding of the used terminologies and how they are being used in our work.

As our memory minimization algorithm is concentrated on handwritten code in general purpose languages, we first explain a few important ideas that are used in the context of general compilation of handwritten code, a snapshot of common compiler optimizations and how we have employed them.

After we have furnished the above concepts of compilers and conventional optimization techniques, we elucidate the terminologies explicitly used for our research and had been primarily used by C. Lam in [28] and is pertinent in our algorithms.

4.1 Definitions Related to Compiler Optimization in General

- *Expression Trees:*

  Expression Trees are inherent tree-like structures used to represent algebraic expressions. As an example let us consider the following expression:

  $$a \div b + (c + d) * e$$

  The expression tree of the above expression is represented as follows in Figure 4.1:

- *Abstract Syntax Tree (AST):*

  It is well known, that during compilation, after the code is passed through the Lexical and Syntax Analyzer, an abstract tree is constructed, such that each independent statement forms a child, such that the parent is the operator and the leaf nodes are operators.

  An abstract syntax tree is a finite, labeled, directed tree, where each interior node represents a programming language construct and the children of that node represent meaningful components of the construct. Internal nodes are labeled by operators, and the leaf nodes

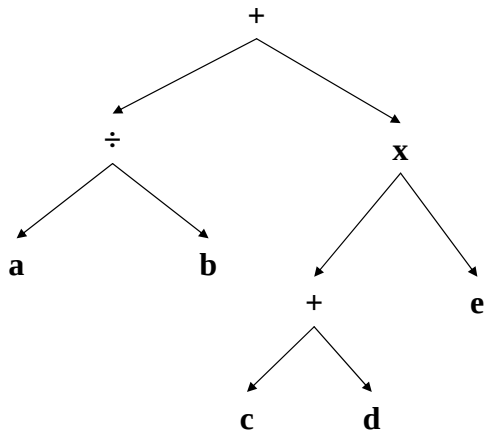


FIGURE 4.1: Example of An Expression Tree

represent the operands of the operators. Thus, the leaf nodes are NULL operators and only represent variables or constants. An Abstract Syntax Tree differs from a Concrete Parse Tree by omitting nodes and edges for syntax rules that do not affect the semantics of the program.

The source code when parsed to produce an abstract syntax tree, a lexer (like flex) is used to recognize tokens (Sequences of characters that make words in the language) and using a parser (like bison) the words are grouped structurally to produce AST.

- *Symbol Tables:*

  A symbol table is broadly defined as a compile-time data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and sometimes its location. It is not used during run time by statically typed languages.

  Specifically, a symbol table stores the following usually:

  - For each *Type Name*, its type definition (e.g., for the C type declaration typedef int* mytype, it maps the name mytype to a data structure that represents the type int*).
  - For each *Variable Name*, its type. If the variable is an array. it also stores dimension information. It may also store storage class, offset in activation record etc.
  - For each *Constant Name*, its type and value.

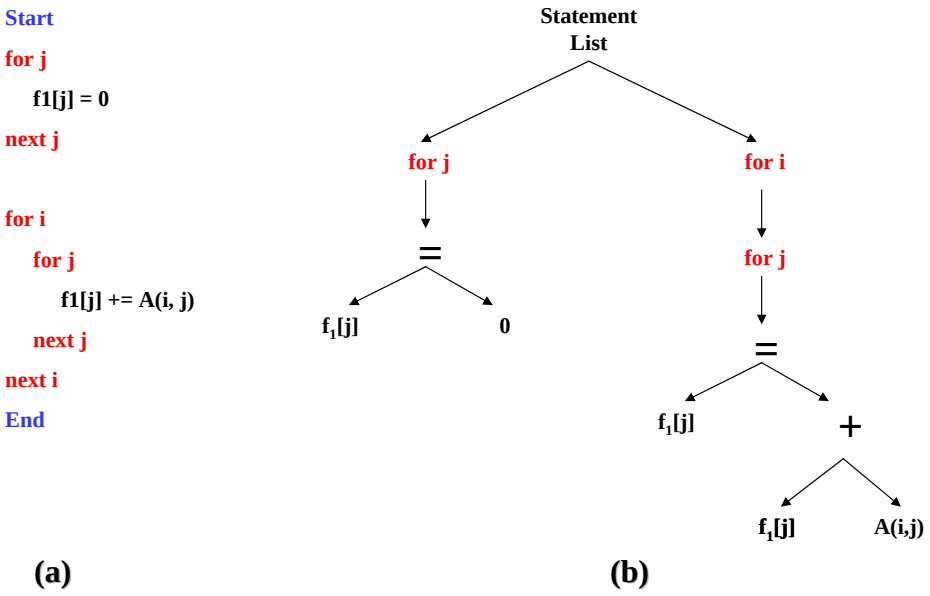


FIGURE 4.2: Abstract Tree Construction from Code Fragment

– For each *Function and Procedure*, its formal parameter list and its output type. Each formal parameter must have name, type, type of passing (by-reference or by-value), etc.

- *Dependency Graph:*

  A Dependency Graph is an attribute b at a node in a parse tree depends on an attribute c, then the semantic rule for b at that node must be evaluated after the semantic rule that defines c. The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called a dependency graph. In other words, a dependency graph depicts the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first in needed to compute the second.

- *Directed Acyclic Graphs (DAG):*

  A Directed Acyclic Graph (DAG) for an expression identifies the common subexpressions in the expression.Like a syntax tree,a DAG has a node for every subexpression of the expression; an interior node represents an operator and its children represent its operands. The difference is that a node in a DAG representing a common subexpression has more than one parent in a syntax tree, common subexpression would be represented as a duplicated subtree.

For example, let us consider the following expression [3]:

$$a + a + a * (b - c) + (b - c) * d$$

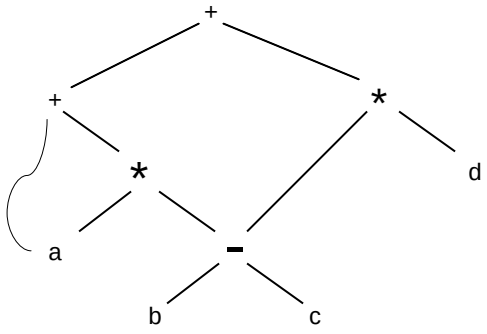


FIGURE 4.3: DAG for Expression a + a + a ∗ (b − c) + (b − c) ∗ d

In Figure 4.3, leaf a has two parents, because a appears twice in the expression. The two occurrences of the sub-expression $b-c$ are represented by one node, the node labeled -. That node has two parents, representing its two uses in the subexpressions $a * (b - c)$ and $(b - c) * d$. Even though, b and c appear twice in the complete expression, their nodes each have one parent, since both uses are in the common subexpression $b-c$.

- *Compiler Optimization:*

  Compiler optimization is the process of tuning the output of a compiler to minimize or maximize some attribute of an executable program. The most common requirement is to minimize the time taken to execute a program; a less common one is to minimize the amount of memory occupied, and the growth of portable computers has created a market for minimizing the power consumed by a program.

  Following are the main ideas in Compiler Optimization:

  - *Avoid redundancy:*

    Reuse results that are already computed and store them for use later, instead of recomputing them.

  - *Less code:*

    Remove unnecessary computations and intermediate values. Less work for the CPU, cache, and memory is usually faster.

- *Straight line code, fewer jumps:*

  Less complicated code. Jumps interfere with the prefetching of instructions, thus slowing down code.

- *Locality:*

  Code and data that are accessed closely together in time should be placed close together in memory to increase spatial locality of reference.

- *Exploit the memory hierarchy:*

  Accesses to memory are increasingly more expensive for each level of the memory hierarchy, so place the most commonly used items in registers first, then caches, then main memory, before going to disk.

- *Parallelize:*

  Reorder operations to allow multiple computations to happen in parallel, either at the instruction, memory, or thread level.

- *Code Optimization:*

  In this technique, the initial code written by the user is modified so that the system performs better.

  Code Optimizations that are performed automatically by a compiler or manually by the programmer, can be classified by various characteristics.

  - *The scope of the optimization:*

    * *Local optimizations* — Performed in a part of one procedure.

      · *Common sub-expression elimination:* (e.g. those occurring when translating array indices to memory addresses.

      Consider the following expression:

      $$(\text{a} + \text{b}) - (\text{a} + \text{b}) \text{ a} + \text{b}) \div 4$$

      In the above example, *common subexpression* refers to the duplicated $(a + b)$. In order to avoid redundancy, the value of $(a + b)$ is calculated only once.

      · *Using registers for temporary results, and if possible for variables.*

      · *Replacing multiplication and division by shift and add operations.*

    * *Global optimizations — Performed with the help of data flow analysis and split-lifetime analysis.*

        - *Code motion (hoisting) outside of loops,*
        - *Value propagation, and*
        - *Strength reductions*
    - *Inter-procedural optimizations*
- *What is improved in the optimization:*
    - *Space optimizations* — Reduces the size of the executable/object.
        - *Constant pooling,*
        - *Dead-code elimination.*
    - *Speed optimizations* — Most optimizations belong to this category.

- *Intermediate Representation (IR):*

  In the process of translating a source program to a target code, a compiler usually constructs one or more Internediate Representations (IR), which usually has varied forms. Syntax trees are a form of IR commonly used during syntax and semantic analysis. Machine dependent code-optimization is carried out in the intermediate code for production of better code.

- *Canonical Trees:*

  For the purposes of code optimization, the compiler often evaluate parts of the IR in a different fashion compared to the initial one, to test which order maximizes optimization.One disadvantage of this process, is that subexpressions (parts of the IR tree) may have side effects and different order of evaluating functions can have end effects.

  In order to overcome this shortcoming, the IR tree is rewritten as (broken down into) an equivalent list of canonical trees. Thus a Canonical Tree is a tree where any subtree can be evaluated in any order (informal definition). Rewriting (transformation) the IR tree into multiple Canonical Trees usually consists of:

  - *Subexpression Extraction*
  - *Subexpression insertion*
  - *Subexpression commutation, if possible:*

    $$(A + B) + C = A + (B + C)$$

  - *Moving procedure calls to the root of the canonical tree*

- *Data Flow Analysis (DFA):*

  Data-Flow Analysis refers to a body of techniques that derive information about the flow of data along program execution paths. This requires associating with every point in the program, a *data-flow value* that represents an abstraction of the set of all possible program states that can be observed for that point.

  The four most important Data-flow values considered for code optimization are Successor, Predecessor, In and Out for a statement. For a statement s, the data flow values are defined as follows:

  - *succ(s)*: Set of Immediate successor statements of s
  - *pred(s)*: Set of Immediate predecessor statements of s
  - *In(s)*: Program point just before executing s
  - *Out(s)*: Program point just after executing s

  In set notation, In and Out can be representated as follows:

  - *Out(s)*: *Gen(s)* $\bigcup$ (*In(s)* - *Kill(s)*)

- *Basic Blocks:*

  A Basic Block is a piece of code that has one entry point (i.e., no code within it is the destination of a jump instruction), one exit point and no jump instructions contained within it. The start of a basic block may be jumped to from more than one location. The end of a basic block may be a jump instruction or the statement before the destination of a jump instruction. In our case, each subtree in the abstract tree are basic blocks.

  In a Basic Block, the IN and OUT sets keep track of what values are passed on to the block and what values are passed out after the computation steps are completed within the block.

- *Reaching Definitions:*

  Reaching Definitions is a data-flow analysis which statically determines which definitions may reach a given point in the code. It is defined as follows by Aho, Ullman and Sethi as "A definition d reaches a point p if there exists a path from the point immediately following d to p such that d is not killed (overwritten) along that path."

  A definition of a variable x is a statement that assigns, or may assign, a value to x. When a

variable is defines, it becomes a set of the Generated values of a basic block, often represented as *GEN*.

A definition of a variable, say x, is said to have been *killed* if there is another definition of x along the path. All variables that are *killed* forms a part of the *KILL* set of the basic block.

During data flow analysis, each basic block, maintains a set of all variables that are being Generated and Killed in the block. An example of *GEN* and *KILL* for a simple block of code is shown in Figure 4.4.

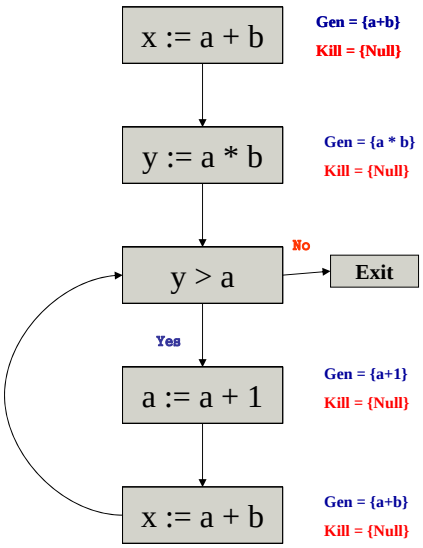


FIGURE 4.4: Example of Computation of *GEN* and *KILL*

- *Loop Optimizations:*

  Loop transformations or optimization, plays an important role in improving cache performance and effective use of parallel processing capabilities. Loop fusion or loop combining is one of those techniques which mainly attempts to reduce loop overhead. Loop fusion approach combines two adjacent loops that would iterate the same number of times (whether or not that number is known at compile time) by confirming the fact that they make no reference to each other's data.

  In our research, we make use mainly of two kinds of loop optimizations, namely:

  – *Loop Fusion:*

    Loop fusion approach combines two adjacent loops that would iterate the same number

of times (whether or not that number is known at compile time), by confirming the fact that their bodies do not make reference to each other's data.

- *Loop Tiling:*

  Loop tiling partitions a loop's iteration space into smaller chunks or blocks, so as to help ensure data used in a loop stays in the cache until it is reused. The partitioning of loop iteration space leads to partitioning of large array into smaller blocks, thus fitting accessed array elements into cache size, enhancing cache reuse and eliminating cache size requirements.

- *Definition of a Variable:*

  When a variable, v, is on the Left-Hand Side of an assignment statement, such as *s(j)*, then *s(j)* is a definition of v. Every variable (v) has at least one definition by its declaration (V) (or initialization).

- *Use of a Variable:*

  If variable, v, is on the Right-Hand Side of statement *s(j)*, there is a statement, s(i) with $i < j$ and *min(j-i)*, that it is a definition of v and it has a use at *s(j)* (or, in short, when a variable, v, is on the Right-Hand Side of a statement *s(j)*, then v has a use at statement *s(j)*).

- *Use-Definition Chain (UD Chain):*

  This is a data structure that consists of a use, U, a variable, and all the definitions, D, of that variable that can reach that use without any other intervening definitions. A definition can have many forms, but is generally taken to mean the assignment of some value to a variable (which is different from the use of the term that refers to the language construct involving a data type and allocating storage).

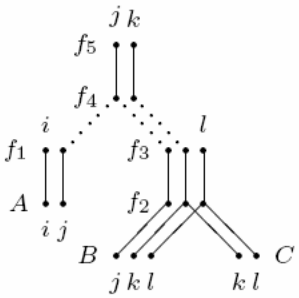
- *Definition-Use Chain (DU Chain):*

  This data structure is the counterpart of a UD Chain is a which consists of a definition, D, of a variable and all the uses, U, reachable from that definition without any other intervening definitions.

```
initialize f1
for i
  for j
    fA=A(i,j)
    f1[j]+=fA
initialize f3
for k
  for l
    fC=C(k,l)
    for j
      fB=B(j,k,l)
      f2=fB×fC
      f3[j,k]+=f2
initialize f5
for j
  for k
    f4=f1[j]×f3[j,k]
    f5[k]+=f4
```

Partially Unfused Code

(a)



Fusion Graph for Partially Unfused Code

(b)

FIGURE 4.5: Illustration of Loop Fusion for Memory Minimization

4.2 Preliminary Definitions of Terminologies Specially Used in Our Algorithms

- *Scope and Fusion Scope of a Loop:*

  The *scope of a loop* of index i in a subtree rooted at v, denoted $scope(i, v)$, is defined in the usual sense as the set of nodes in the subtree that the fused loop spans. That is, if the i-loop is fused, $scope(i, v) = scope(c) \cap subtree(v)$, where c is a fusion chain for the i-loop with $v \in scope(c)$. If the i-loop of v is not fused, then $scope(i, v) = \emptyset$. We also define the *fusion scope of an i-loop* in a subtree rooted at v as $fscope(i, v) = scope(i, v)$ if the i-loop is fused between v and its parent; otherwise $fscope(i, v) = \emptyset$. As an example, for the fusion graph in Figure 4.5(b) [31],

  $scope(j, f_3) = \{B, f_2, f_3\}$, but $fscope(j, f_3) = \emptyset$.

- *Indexset Sequence:*

  To describe the relative scopes of a set of fused loops, we introduce the notion of an *indexset sequence*, which is defined as an ordered list of disjoint, non-empty sets of loop indices. For example, $f = \langle\{i, k\}, \{j\}\rangle$ is an indexset sequence. For simplicity, we write each indexset in an indexset sequence as a string. Thus, f is written as $\langle ik, j\rangle$. Let g and g' be indexset sequences. We denote by $|g|$ the number of indexsets in g, $g[r]$ the r-th indexset in g, and

$Set(g)$ the union of all indexsets in g, i.e., $Set(g) = \bigcup_{1 \le r \le |g|} g[r]$. For instance, $|f| = 2$, $f[1] = \{i, k\}$, and $Set(f) = Set(\langle j, i, k \rangle) = \{i, j, k\}$. We say that g' is a *prefix* of g if $|g'| \le |g|$, $g'[|g'|] \subseteq g[|g'|]$, and for all $1 \le r < |g'|$, $g'[r] = g[r]$. We write this relation as $prefix(g', g)$. So, $\langle\rangle$, $\langle i \rangle$, $\langle k \rangle$, $\langle ik \rangle$, $\langle ik, j \rangle$ are prefixes of f, but $\langle i, j \rangle$ is not. The *concatenation* of g and an indexset x, denoted $g + x$, is defined as the indexset sequence g'' such that if $x \neq \emptyset$, then $|g''| = |g| + 1$, $g''[|g''|] = x$, and for all $1 \le r < |g''|$, $g''[r] = g[r]$; otherwise, $g'' = g$.

  - *Special Indexset Sequence:*

    $\langle * \rangle$, represents the constant *any*, and it denotes the maximal fusion for Array and Constant nodes. It indicates that any index of the parent can be fused between the leaf node and the parent, and not only indices of the leaf node.

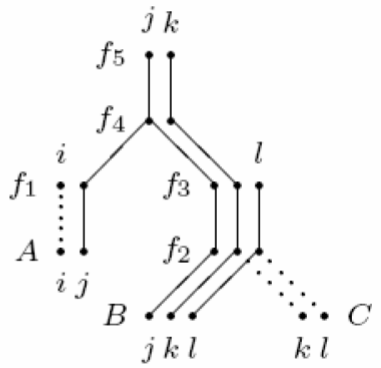
- *Fusion:*

  We use the notion of an indexset sequence to define a *fusion*. Intuitively, the loops fused between a node and its parent are ranked by their fusion scopes in the subtree from largest to smallest; two loops with the same fusion scope have the same rank (i.e., are in the same indexset). For example, in Figure 4.6(b) [31], the fusion between f_2 and f_3 is $\langle jkl \rangle$ and the

```
for k
  for l
    fC[k,l]=C(k,l)
initialize f5
for j
  for i
    fA[i]=A(i,j)
  initialize f1
  for i
    f1+=fA[i]
  for k
    initialize f3
    for l
      fB=B(j,k,l)
      f2=fB×fC[k,l]
      f3+=f2
    f4=f1×f3
    f5[k]+=f4
```

Maximally Fused Code

(a)



Fusion Graph for Maximally Fused Code

(b)

FIGURE 4.6: Illustration of Maximal Fusion for Memory Minimization

fusion between f_4 and f_5 is $\langle j, k \rangle$ (because the fused j-loop covers two more nodes, A and

f_1). Formally, a fusion between a node v and $v.parent$ is an indexset sequence f such that

1. $Set(f) \subseteq v.indices \cap v.parent.indices$,
2. for all $i \in Set(f)$, the i-loop is fused between v and $v.parent$, and
3. for all $i \in f[r]$ and $i' \in f[r']$,
   (a) $r = r'$ iff $fscope(i, v) = fscope(i', v)$, and
   (b) $r < r'$ iff $fscope(i, v) \supset fscope(i', v)$.

- *Nesting:*

  Similarly, a *nesting* of the loops at a node v can be defined as an indexset sequence. Intuitively, the loops at a node are ranked by their scopes in the subtree; two loops have the same rank (i.e., are in the same indexset) if they have the same scope. For example, in Figure 4.5(b), the loop nesting at f_3 is $\langle kl, j\rangle$, at f_4 is $\langle jk\rangle$, and at B is $\langle jkl\rangle$. Formally, a nesting of the loops at a node v is an indexset sequence h such that

  1. $Set(h) = v.indices$ and
  2. for all $i \in h[r]$ and $i' \in h[r']$,
     (a) $r = r'$ iff $scope(i, v) = scope(i', v)$, and
     (b) $r < r'$ iff $scope(i, v) \supset scope(i', v)$.

  By definition, the loop nesting at a leaf node v must be $\langle v.indices\rangle$ because all loops at v have empty scope.

- *Legal Fusion:*

  A legal fusion graph (corresponding to a loop fusion configuration) for an expression tree T can be built up in a bottom-up manner by extending and merging legal fusion graphs for the subtrees of T. For a given node v, the nesting h at v summarizes the fusion graph for the subtree rooted at v and determines what fusions are allowed between v and its parent. A fusion f is legal for a nesting h at v if $prefix(f, h)$ and $\text{set}(f) \subseteq v.parent.indices$. This is because, to keep the fusion graph legal, loops with larger scopes must be fused before fusing those with smaller scopes, and only loops common to both v and its parent may be fused. For example, consider the fusion graph for the subtree rooted at f_2 in Figure 4.5(e). Since the nesting at f_2 is $\langle kl, j\rangle$ and $f_3.indices = \{j, k, l\}$, the legal fusions between f_2 and f_3 are $\langle\rangle$,

$\langle k\rangle$, $\langle l\rangle$, $\langle kl\rangle$, and $\langle kl, j\rangle$. Notice that all legal fusions for a node v are prefixes of a *maximal legal fusion*, which can be expressed as

In Figure 4.5(b), the maximal legal fusion for C is $\langle kl\rangle$, and for f_2 is $\langle kl, j\rangle$.

- *Resulting Nesting:*

  Let u be the parent of a node v. If v is the only child of u, then the loop nesting at u as a result of a fusion f between u and v can be obtained by the function

  $$\mathit{ExtNesting}(f, u) = f + (u.\mathit{indices} - \mathit{Set}(f))$$

  For example, in Figure 4.6(b), if the fusion between f_2 and f_3 is $\langle kl\rangle$, then the nesting at f_3 would be $\langle kl, j\rangle$.

- *Compatible Nestings:*

  Suppose v has a sibling v', f is the fusion between u and v, and f' is the fusion between u and v'. For the fusion graph for the subtree rooted at u (which is merged from those of v and v') to be legal, $h = \mathit{ExtNesting}(f, u)$ and $h' = \mathit{ExtNesting}(f', u)$ must be *compatible* according to the condition:

  > for all $i \in h[r]$ and $j \in h[s]$,
  > if $r < s$ and $i \in h'[r']$ and $j \in h'[s']$, then $r' \leq s'$.

  This requirement ensures an i-loop that has a larger scope than a j-loop in one subtree will not have a smaller scope than the j-loop in the other subtree. If h and h' are compatible, the resulting loop nesting at u (as merged from h and h') is h'' such that

  > for all $i \in h''[r'']$ and $j \in h''[s'']$,
  > if $i \in h[r]$, $i \in h'[r']$, $j \in h[s]$, and $j \in h'[s']$, then
  >
  > 1. $r'' = s''$ implies $r = s$ and $r' = s$, and
  > 2. $r'' \leq s''$ implies $r \leq s$ and $r' \leq s'$.

  Effectively, the loops at u are re-ranked by their combined scopes in the two subtrees to form h''. As an example, in Figure 4.6(b), if the fusion between f_1 and f_4 is $f = \langle j\rangle$ and the fusion between f_3 and f_4 is $f' = \langle k\rangle$, then $h = \mathit{ExtNesting}(f, f_4) = \langle j, k\rangle$ and $h' = \mathit{ExtNesting}(f', f_4) = \langle k, j\rangle$ would be incompatible. But if f is changed to $\langle\rangle$, then

$h = \mathit{ExtNesting}(f, f_4) = \langle jk \rangle$ would be compatible with h', and the resulting nesting at f_4 would be $\langle k, j \rangle$. A procedure for checking if h and h' are compatible and forming h'' from h and h' is provided in Chapter 5.

- *The "more-constraining" relation on Nestings:*

  A nesting h at a node v is said to be *more or equally constraining than* another nesting h' at the same node, denoted $h \sqsubseteq h'$, if for all legal fusion graph G for T in which the nesting at v is h, there exists a legal fusion graph G' for T in which the nesting at v is h' such that the subgraphs of G and G' induced by $T - \mathit{subtree}(v)$ are identical. In other words, $h \sqsubseteq h'$ means that any loop fusion configuration for the rest of the expression tree that works with h also works with h'. This relation allows us to do effective pruning among the large number of loop fusion configurations for a subtree in Section 5. It can be proved that the necessary and sufficient condition for $h \sqsubseteq h'$ is that

  for all $i \in m[r]$ and $j \in m[s]$, there exist r', s' such that

  1. $i \in m'[r']$ and $j \in m'[s']$,
  2. $r = s$ implies $r' = s'$, and
  3. $r < s$ implies $r' \leq s'$

  where $m = \mathit{MaxFusion}(h, v)$ and $m' = \mathit{MaxFusion}(h', v)$. Comparing the nesting at f_3 between Figure 4.5(b) and (f), the nesting $\langle kl, j \rangle$ in (e) is more constraining than the nesting $\langle jkl \rangle$ in (f).

5. Algorithm for Memory Minimization

5.1 Overview

Handwritten code in a general purpose language has to go through various stages of compilation before it is translated into the target language as shown in Figure 5.1.

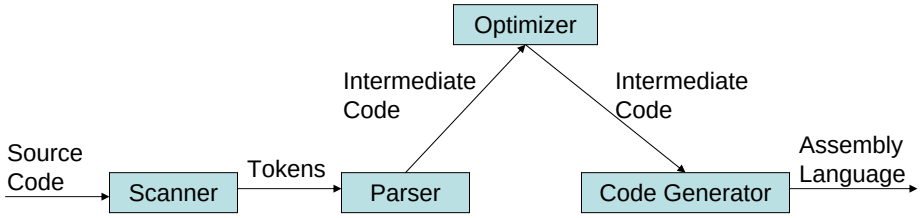


FIGURE 5.1: Phases of Compilation

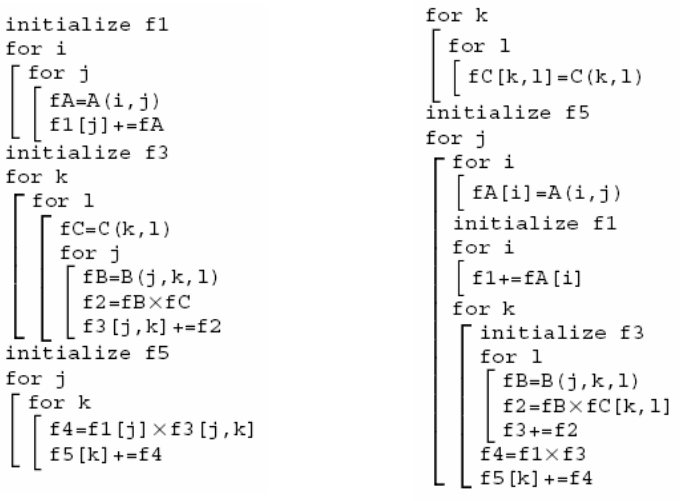
Machine Independent Optimization tunes the output of a compiler to minimize or maximize some attribute of an executable program is carried out to minimize the time taken to execute a program and the amount of memory occupied. Because high-level language constructs often introduces substantial run-time overhead, code optimization techniques are hence often used to eliminate unnecessary instructions or replace set of instructions by better and faster set of instructions that essentially does the same thing.

There exists various optimization techniques, namely, Loop optimizations (Section 4.1), Data Flow optimization (Section 4.1) and Single Statement Assignment (SSA) optimization. In our research, we employ only the first two to attain maximum memory minimization. Our memory minimization approach seeks to perform loop fusion transformations to reduce the memory requirements. The main difference between existing memory minimization algorithm in the TCE and the one that will be discussed in this thesis is that, that all extant ones apply on expression trees

(Section 4.1) while ours features the application of a generalized version of the same applied on handwritten code.

In earlier work on the TCE project, loop fusion techniques have been employed successfully for minimizing memory for large algebraic expressions specifically targeting a class of scientific computations encountered in chemistry and physicselectronic structure calculations, where many computationally intensive components are expressible as a set of tensor contractions. Loop fusion merges loop nests with common outer loops into larger imperfectly nested loops. When one loop nest produces an intermediate array that is consumed by another loop nest, fusing the two loop nests allows the dimension corresponding to the fused loop to be eliminated in the array. This results in a smaller intermediate array and, thus, lowers the memory requirement. The use of loop fusion can be seen to result in significant potential reduction to the total memory requirement. For a computation composed of a number of nested loops, there will generally be a number of fusion choices that are not all mutually compatible. This is because different fusion choices could require different loops to be made the outermost.

In this thesis, we generalize those techniques for handwritten code by using common standard machine-independent optimization techniques before the code generation phase is reached during compilation. Let us consider the following example as given in Figure 5.2 [31]. The brackets indicate



Partially Fused Code (a)

Maximally Fused Code (b)

FIGURE 5.2: Example of A Partially Fused Code and Maximally Fused Version of the Same

the scope of the loops. The size of the arrays are reduced in the fused code by use of loop fusions

as shown. In order to achieve this, we represent the unfused code in the form of an abstract syntax tree (Section 4.1). In the following sections, we explain the various stages of the minimization process.

5.2 Tree Canonicalization

In order to look for potential optimization, we first build the abstract syntax tree for the given code. We then build the symbol table and populate it with the variable names, types, and pointers to all nodes in the AST that uses or defines the variable. The steps we follow to do this is shown in Algorithm 1.

Algorithm 1 *Traversal 1: Builds and Populates the Symbol Table*

```
for each node v in some top down traversal of T do
  if v.nchildren = 2 then
    /* Consider the Left Child */
    /* If Variable DONOT exist in Symbol Table, i.e, variable occurs first in Tree*/
    if v.child1 DoNot Exist in SymTab then
      if v.parent is an Assignment then
        /* If Parent of v is an assignment, save the variable name in SymTab, Store the pointer to the
        assignment node in AssignOpNodTab and in the Pointer List in SymTab, Build the Reaching
        definitions graph */
        SymTab[row++].VarName ← VarName
        Assign to set of Pointers in SymTab[row++].PointerSet ← Pointer to v.child1
      end if
      /* If NOT an assignment Node, just save the Variable Name in SymTab*/
    else
      SymTab[row++].VarName ← VarName
    end if
    if v.child2 DoNot Exist in SymTab AND v.child2 is NOT Static then
      SymTab[row++].VarName ← VarName
    end if
    if v.child2 EXISTS in SymTab then
      Locate v.child2 in SymTab
      In the row of SymTab where v.child2 is Found
      APPEND to the same set of pointers in SymTab[row].PointerSet ← Pointer to v.child2
    end if
    if v.child1 EXISTS in SymTab AND v.parent is an Assignment then
      Locate v.child1 in SymTab
      In the row of SymTab where v.child1 is Found
      APPEND to the same set of pointers in SymTab[row].PointerSet ← Pointer to v.child1
    end if
  end if
end for
```

After we have populated the symbol table, the abstract syntax tree is again traversed top-down and we find out all subtrees that can be potentially converted to canonical trees (Section 4.1). This helps us to filter parts of the code which can be probably be affected by change in code

organization.

5.3 Reaching Definitions

In order to apply loop fusion to our algorithm, we need to have sufficient knowledge of flow of data through the blocks of code or subtrees in the abstract syntax tree. We use standard Reaching Definitions algorithm, one of the standard data flow analysis technique [3].

After the tree is canonicalized, the tree is free of all portions of code that may affect the results of the loop fusion, we compute the *GEN* and *KILL* sets for each node of the AST. This information is important for data flow analysis and in calculating the reaching definition later on. We evaluate the *GEN* and *KILL* in a third traversal (first being traversal for building and filling the symbol table and the second being filtering canonical trees from the AST as discussed in the preceding paragraph). This traversal is a top-down traversal and we store the values we compute along with the nodes itself, so that it is easy to retrieve all information about the node all at once. The *GEN* set, the acronym for Generated values, is computed such that, whenever we come across an assignment statement in the AST, we assign the pointer to the node of the left child of the assignment node to the *GEN* set. The *KILL* set on the other hand, contains set of all variables which are killed during an assignment. A variable is said to have been killed, when a variable is redefined in the tree. A new definition of a variable kills its earlier value. The moment, we include a pointer to a variable node into the *GEN* set, we simultaneously include all pointers to the same variable, if it is defined earlier, to the *KILL* set.

The steps we follow to construct our GEN and KILL sets are shown in Algorithm 2.

Algorithm 2 *Traversal 2: Compute Gen, Kill, In, Out*

```
row=0
for each node v in some top down traversal of T do
   /* Identify an assignment node */
   if v=Assignment Operator then
      /* Find out the left child of assignment node in the Symbol Table */
      Locate v.child1 in SymTab
      /* Variable ChildRow points to the row in Symbol Table which contains v.child1 as VarName */
      ChildRow ← Row in which v.child1 exists in SymTab
      v.GEN ← Pointer to node v.child1
      v.KILL ← Set of all Pointers in SymTab[ChildRow].PointerSet - Pointer to node v.child1
      v.IN ← NULL
      v.OUT ← NULL
      row++
   end if
end for
```

Algorithm 3 *Iterative Algorithm to compute Reaching Definitions*

```
OUT[ENTRY] = Empty

for each Basic Block other than ENTRY do
   OUT[B] = Empty
end for
while changes any to OUT occur do
   /* P is the set of predecessor blocks of current Block B */
   IN[B] → ⋃ OUT[P]

   OUT[B] → GEN[B] ⋃ (IN[B] - KILL[B])
end while
```

Once we have the *GEN* and *KILL* sets computed for all the assignment nodes, we need to determine the values for the *IN* and *OUT*. We use the Reaching Definition algorithm, for each and every assignment nodes to compute the value of *IN* and *OUT* sets. The iterative algorithm to compute reaching definitions used by us is the one described by Aho, Ullman and Sethi. It starts with the estimate that *OUT[B] = Empty* for all Basic Blocks (Section 4.1) *B*, and converge to the desired values of *IN* and *OUT*. The algorithm continues till the *IN's* and *OUT's* converge.

Before applying the above algorithm, we traverse the AST once again top-down. Whenever we reach a node, since the values of *IN-OUT* sets depends on the node type, each kind of node is treated separately for computation of the same. We only consider for the time being: Assignment Node, Loops (For and While), and Conditional Statements (If-Then-Else).

- *Assignment Node:*

  For an assignment node, the IN set contains whatever is passed on as the OUT set by the parent of the node. The OUT set contains all the items, that has been generated (i.e., part of the *GEN* statement) and has not been *KILLed* from the *IN* set.

- *While Loop Node:*

  The *IN* set of the while loop node is what is passed on as the OUT set of its parent node. The body of the while loop is treated as a separate basic block, which may contains other nodes, namely — assignments, loops and if-else conditions. The *OUT* set of the while loop node is same as the OUT set of the while loop body.

- *For Loop Node:*

A For loop can be considered as a combination of one WHILE statement, which checks for the loop condition and two *ASSIGNMENT* statements — one for Loop Counter Initialization and the other for Loop Counter Increment. The *IN* set of a For node is the union of the *OUT* set of the parent of the For node and that of all GENerated items that was a part of the IN set of the node but was not *KILLed*.

- *If-Else Node:*

  The If-Else node is divided into two sets as shown in Figure 5.3. The IN set of the node is similar as for all other nodes, and hence contains the set of OUT items from its parent. The If node is divided into two bodies, the THEN body and the ELSE body. The value of IN set of both the THEN and ELSE bodies are same as the IN set of the If node. The OUT set of the If node, is then computed as the union of the individual OUT sets of both THEN and ELSE parts.

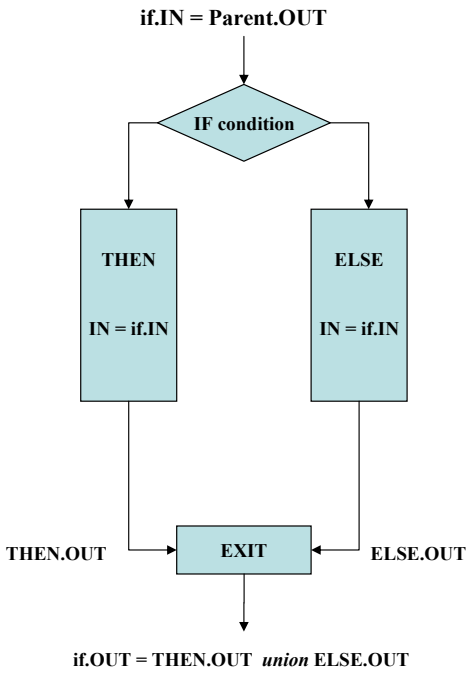


FIGURE 5.3: IN-OUT Computation for If-Else Nodes

The pseudo code for the above traversal for computation of *IN* and *OUT* is as follows:

Our main aim for all the above traversals, was to, get hold of the information of which variables are defined where and their subsequent use, so that our memory minimization

Algorithm 4 *Traversal 3: Compute In, Out and repeat until the values of In-Out do not change*

```
In some top-down Traversal of Abstract Syntax Tree T
repeat
  if v is a FOR node then
    v.IN = (v.parent.OUT ⋃ v.GEN ∪ (v.IN - v.KILL))
    b = v.body
    b.IN = (v.IN ⋃ b.out)
    v.out = (b.out)
  end if
  if v is an ASSIGNMENT node then
    v.IN = (v.parent.OUT)
    v.OUT = (v.GEN ∪ (v.IN - v.KILL))
  end if
  if v is a WHILE node then
    v.IN =(v.parent.OUT)
    b = v.body
    b.IN = (b.OUT ⋃ v.IN)
    v.OUT = (b.OUT)
  end if
  if v is a IF node then
    v.IN = (v.parent.OUT)
    t = v.THENpart
    e = v.ELSEpart
    t.IN = (v.IN)
    e.IN = (v.IN)
    v.OUT = (t.OUT ⋃ e.OUT)
  end if
until neither of OUT changes
```

algorithm has required information for it to decide where to store it — memory or disk. Our long term goal is to reduce memory to disk traffic and only when we have information about definition and use of a variable (specially arrays in our case), is it possible for us to foresee if any form of fusion will improve on the existing code. For this, we draw hypothetically reaching definition edges or more appropriately, *USE-DEF* chains for all the variables. We perform this operation in our next traversal.

As discussed earlier, we need the information about the generation and use of a variable, for us to determine potential and compatible loop fusions.This is done by defining the Def-Use Chains of the variables. As shown in Algorithm 5, the abstract syntax tree is again traversed bottom up and whenever we encounter an assignment statement we perform our checks and computations. In this traversal, we are basically interested on the Right-Hand Side of the assignment. For each child of the assignment node, we track from the Symbol Table list whether the variable had been declared earlier. If the variable has been declared earlier, we build a reaching definition edge such that, we store a set of pointers, so that the

Algorithm 5 *Traversal 4: Add the reaching definition edges*

```
In some bottom up traversal of T

if v is a BINARY OPERATOR then
   Read v.parent.IN set

   if v.child EXISTS in v.IN set then
      /* The variable has been used earlier */
      Locate.Assign = Pointer to the assignment statement where v.child1 was used

      if v.child = Locate.Assign1.child1 then
         /* Only if the value of v.child has been assigned to some expression, do we need to add the reaching
         definition edge */
         /* Pointer from located assignment node to assignment node */
         v.child.use-def = Append to list the set ← (Locate.Assign, v.child.pointer)
      end if

   end if

end if
```

first pointer refers to the pointer to the assignment node, where the variable was declared and the second pointer is the pointer to the present node. This enables us to keep a track of the definition and use of each variable.

This traversal also helps us recognizing intermediate outputs for subtrees. Any variable which is defined but not used are treated as the output of a subtree.

5.4 Array Size Inference

After the fourth traversal, we have all the desired information, it is required to apply the memory minimization algorithm and determine which loops can be potentially fused and hence, which variable needs to be stored where and eventually the disk to memory traffic.

In the fifth traversal of the tree, we compute the Indexset Sequences and Dimensions of the arrays and indexset sequences for binary or unary operations along the AST. The algorithm for the computation of the same are shown in Algorithm 6.

For the type of code we are looking at, we can have a look at the following example in Figure 5.4 for a better understanding. In this, we compute the value of f_1, such that $f_1[j] = f_1[j] + A(i, j)$.

We have just one loop j around f_1, when all the elements of f_1 are assigned to zero. Any static number has dimension equal to *Null*. The dimension for f_1 is thus $\langle j \rangle$. After we move

Algorithm 6 *Traversal 5: Computing Indices and Dimensions*

```
In some top down traversal of T
for each node v in some top down traversal of T do
  if v is an ASSISGNMENT Node then
    v.Dim = v.child1.Dim OR v.child2.Dim
  end if
  if v is a BINARY Operator then
    v.Dim = ⟨MergeNesting(v.child1, v.child2)⟩
  end if
  if v is a Leaf Node then
    if v Exists in Symbol Table then
      if v.parent is BINARY Operator and v is the Left Child of v.parent then
        v.Dim = ⟨∗⟩
      else
        PointerTemp = Locate Last Pointer Set in v.use-def
        DimTemp = Locate Second Node in PointerTemp
        v.Dim = ⟨ Loops around DimTemp ⟩
      end if
    else if v is a constant then
      v.Dim = ⟨∗⟩
    else
      v.Dim = ⟨Dimensionsofv⟩
    end if
  end if

end for
```

on to the next Basic Block in the AST, we find that there are two loops around f_1. Already earlier during assignment, we know that the dimension of f_1 is $\langle j \rangle$. In this Basic Block, we have another array. $A[i,\ j]$ being used and A has dimensions $\langle i, j \rangle$. Since f_1 's dimensions are already known, we also know that for the binary operation(addition) taking place, the indexset sequence for computing possible fusion is primarily dependent on dimensions of A. Hence, the dimensions for f_1 remains the same and the indexset sequence for the addition node is $\langle ij \rangle$, which implies that the loops i and j can be reordered if that helps in memory minimization.

In order to compute the dimensions for an array, we traverse back to the node by refereing to the symbol table and the Use-Def chain, where it has been defined for the first time. We record the information of the surrounding loops in this case and incidentally the same loop variables in the same sequence they are around the array is the dimension for the array.

As mentioned earlier, for any variable, the dimensions are *Null*.

For computing the indexset sequence, we refer to the loops we define for indexset sequence in Subsection 4.2.

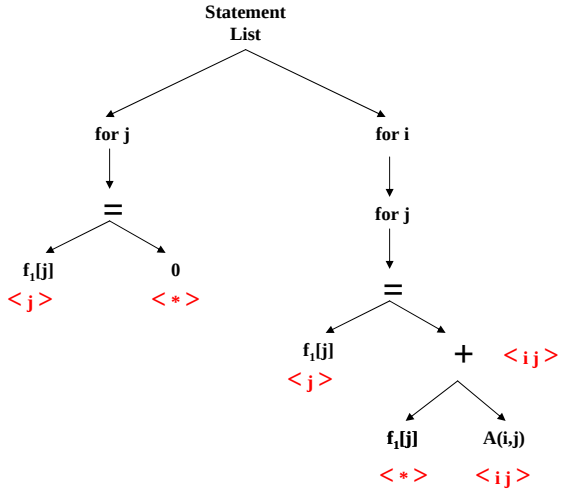


FIGURE 5.4: Example Demonstrating Computation of Indices and Dimensions

5.5 Loop Fusion

All the earlier traversals of the abstract syntax tree was to collect sufficient data flow so as to enable the loop fusion algorithm have knowledge before computing potential fusions. Once we know which variable is generated where and where it is used, we basically have the information to decide on what can be the part of the producer loop and what can consume the same in an optimized fashion. We consider only a static memory allocation model for the time being and hence we compute the memory usage by simply adding the size of all arrays. We use a dynamic programming algorithm for finding a memory-optimal loop fusion configuration for a given abstract syntax tree T. For each node v in subtree t, we compute a solution and represent it as $v.solns$. Each solution for each node v has additional information, namely: fusion $s.fusion$ between v and its parent and $s.cost$ for memory cost and usage so far. When we traverse the tree in a bottom-up fashion, if we reach a leaf, we store the value of $v.fusion$ as $v.indices \bigcup v.parent.indices$. If we reach anywhere but a leaf, we identify the kind of node and its children. We do so because each of the structures in a handwritten code require different treatment. We explain the management of each kind of language constructs as follows:

- *Assignment Node:*

  For an assignment node there are more indices on the RHS than on the LHS and hence we need to perform a reduction operation which removes unfusible indices and adjusts the cost accordingly.

- *Binary Operation:*

  For Binary Operations, the parent has more dimensions than the child and hence, it is necessary to enumerate all possible ways in which the loop structures for the subtrees can be fused with the parent, such that the solutions for the two subtrees can be combined.

- *While Condition Block:*

  We optimize the code inside the while block independently. We do not allow any piece of code to move out or into the while loop. In future work, it may be possible to move data in or out of the loop from other parts of the tree and produce more optimized code. For now we limit ourselves to local optimization in the while subtree portion only.

- *For Loop Block:*

  Since the statements inside the for loop are logically already propagated, we treat it as a NoOp node.

- *If-Then-Else Blocks:*

  We ignore such blocks in this initial algorithm and perform the *then* an *else* blocks as independent subtrees. Similar to the *while* loop, we do not interchange code fragments with the the code outside the *If* statement.

- *Semicolon/Statement Termination:*

  We treat statement termination essentially as a binary operation with nil extension.

Once we get the possible fusion solution sets for the nodes, we traverse the tree once again in a top-down fashion. If v has 2 children, we try to *merge* the compatible solutions from each child. We then *prune* the inferior solutions. We define a solution s is *inferior* to another *s'*, such that *s.nesting* is more or equally constraining and *s'.nesting* do not use less memory than *s.nesting*.

5.6 Code Generation

The solution tree data structure that is constructed by our algorithm, is designed to efficiently represent the information needed by the algorithm, but it is not very convenient for generating code. The Extended and Reduced are no more useful once the optimal solution is computed, and hence we remove them from the tree, after the optimal solution is found. Also, the fusion stored in a solution summarizes only the loop nesting constraints of the subtree.

To simplify code generation, we first plan to translate the solution tree into a fusion tree representation in which for each node in the expression tree we store the nesting and the fusion with the parent. This fusion tree representation will be generated in a top-down traversal of the fused abstract syntax tree. In this traversal, any constraints on loop nestings from higher up in the fused abstract syntax tree are propagated down to the leafs. The resulting fusion tree do not have the extraneous Reduced or Extended nodes, and the fusions and nestings stored at each node contain the full loop nesting information.

Once the full fusion information is available at every node, the existing for loops can be removed and the final code can be generated by topologically sorting the nodes in the tree according to dependency and loop fusion information and inserting the final for loop nodes.

Thus the algorithm can be illustrated in the form of a flowchart diagram as shown in Figure 5.5.

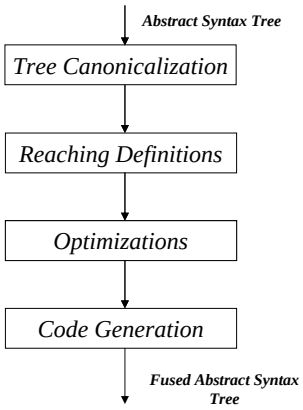


FIGURE 5.5: Flowchart Depiction of Stages of our Algorithm

6. An Example

In the previous chapter, we described our memory minimization algorithm. This section illustrates with an example how the algorithm works. To illustrate how the algorithm works, we use an example that computes a tensor expression.

Let us consider the following tensor as an example [31]:

$$W[k] = \sum_{ijkl} A[i,j] \times B[j,k,l] \times C[k,l] \tag{6.1}$$

This multidimensional integral or tensor contraction expression can be broken down into a sequence of smaller formulas as follows:

$$\begin{aligned} f_1[j] &= \sum_i A[i,j] \\ f_2[j,k,l] &= B[j,k,l] \times C[k,l] \\ f_3[j,k] &= \sum_l f_2[j,k,l] \\ f_4[j,k] &= f_1[j] \times f_3[j,k] \\ f_5[k] &= \sum_j f_4[j,k] \\ W[k] &= f_5[k] \end{aligned} \tag{6.2}$$

A typical code for computation of the above expression is shown in Figure 6.1.

We demonstrate the operation of our algorithm in two sections: an Analysis of the Reaching Definitions and the Loop Fusion algorithm, which are the major parts of our algorithm.

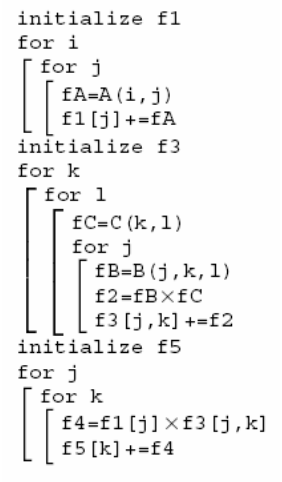


FIGURE 6.1: Example of Partially Fused Code for Computation of Tensor Expression as shown in Eqn. 6.1

6.1 Reaching Definitions

As discussed in the Algorithm Section, before we apply the memory minimization algorithm, we traverse the AST four times and collect all the information we need to apply the loop fusion algorithm. The equivalent AST for the unfused code in Figure 6.1 is shown in Figure 6.2.

In order to explain the algorithm better with the help of the example in Figure 6.1, we number all the assignment nodes as shown in Figure 6.3 and refer to those nodes later in this section.

We will illustrate with the example in Figure 6.1 and demonstrate what information each traversal collects and how those data are later used by the memory minimization algorithm.

- *Traversal 1: Building and Populating Symbol Table* When the compiler builds the symbol table, it has three columns. The first column is where the symbol is located in the memory, the second is the type and the third is the name of the symbol. We have an additional information to store the set of pointers to the assignment nodes where the variable is used or defined. For the code we use as an example, we depict a symbol table in 6.1 which for explanatory purpose just has the variable name and set of pointers. Only these two information are needed for understanding the importance of this traversal and how these data are used later on.

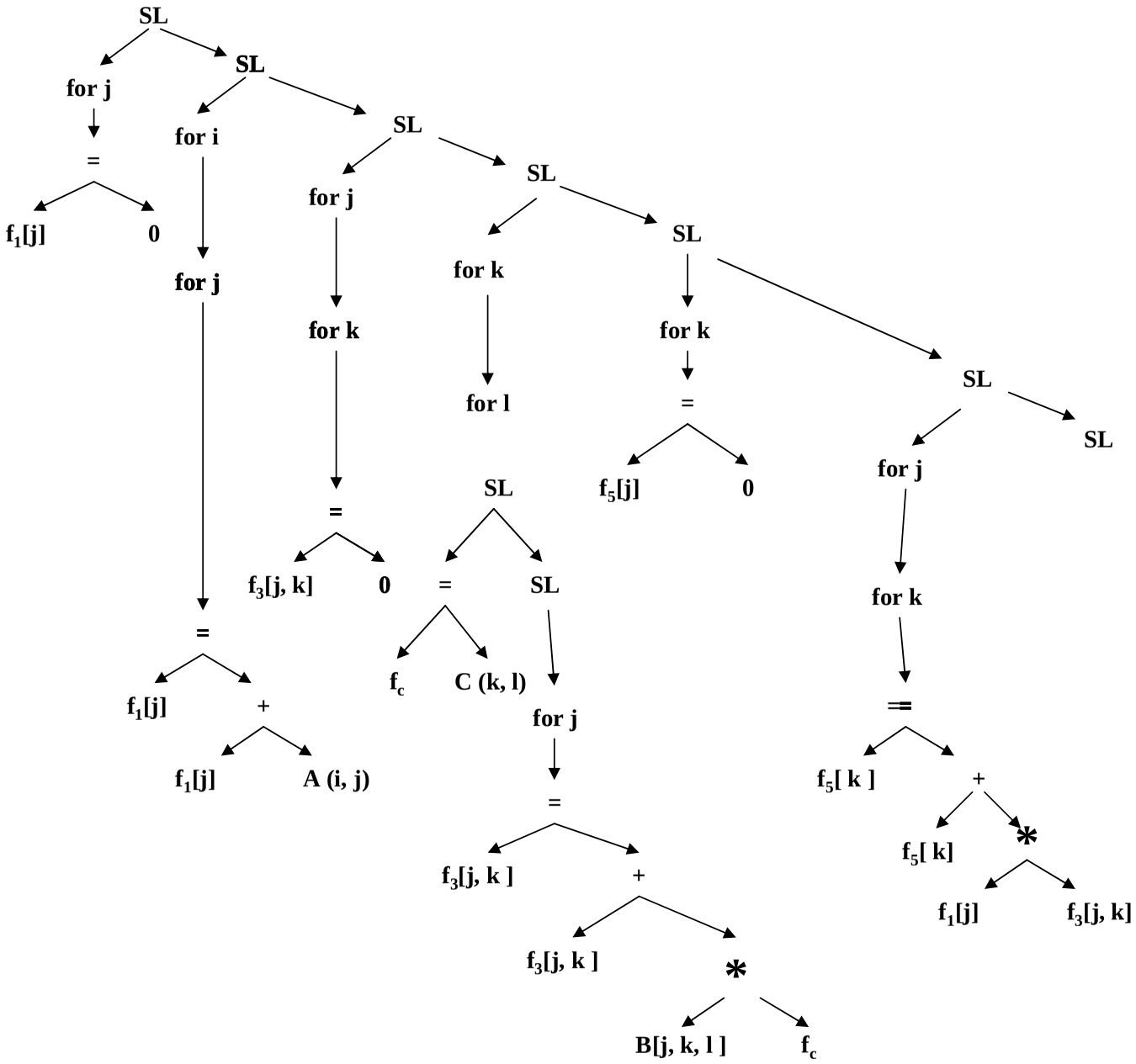


FIGURE 6.2: Abstract Syntax Tree for the Partially Fused Code shown in Figure 6.1

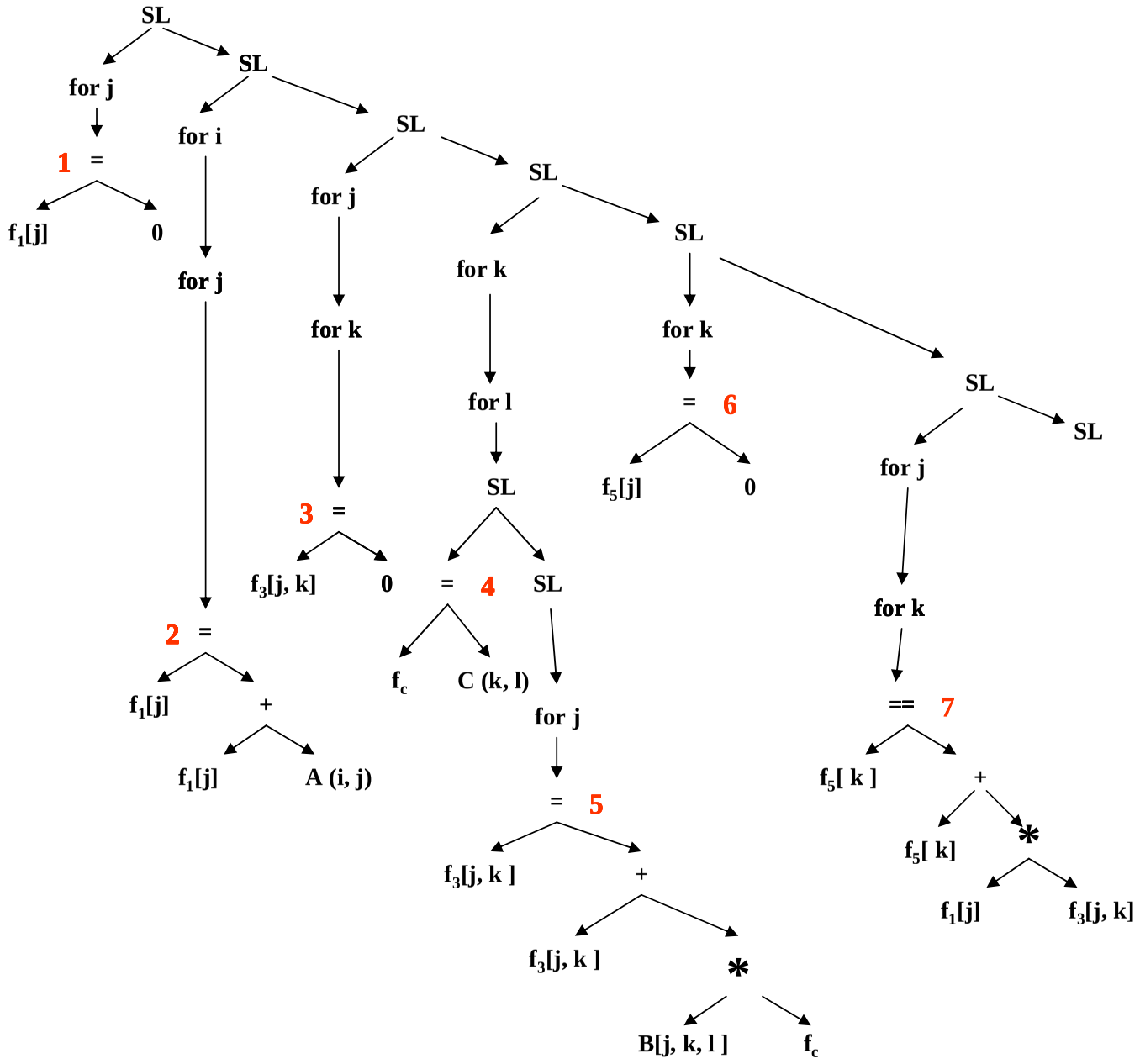


FIGURE 6.3: Abstract Syntax Tree with Numbered Nodes for the Example Code shown in Figure 6.1

For the set of pointers, we use the number on the nodes, as shown in Figure 6.3.

TABLE 6.1: Symbol Table

Variable Name	Pointer Set
f1	1, 2, 6
A	2
f3	3, 5, 7
fc	4, 5
B	5
f5	6, 7

- *Traversal 2: Computing GEN and KILL sets for each assignment nodes* This top-down traversal over the AST helps us determine which variables are generated and killed. Whenever a variable is generated for the second time and thereon, it kills all the existing values thereof. In Table 6.2, we show the variables that are killed in the corresponding nodes. Since it is implied that a new definition of a variable kills the earlier definitions, we do not show it redundantly in the table.

TABLE 6.2: List of Variables Generated After Traversal 2

Node	Variables Generated	Variables Killed
1	f1	f1(generated in Node 2)
2	f1	f1(generated in Node 1)
3	f3	f3(generated in Node 5)
4	fc	—
5	f3	f3(generated in Node 3)
6	f5	f5(generated in Node 7)
7	f5	f5(generated in Node 6)

- *Traversal 3: Computing IN and OUT sets for each node*

  The *IN* and *OUT* sets for each node are computed using the Reaching Definition Algorithm as in Algorithm 3 and are shown in Table 6.3. Owing to the fact that this is an iterative algorithm and the process continues till the *IN* and *OUT* set converges, we just present the final *IN* and *OUT* sets for each assignment node, as only they are the ones we are interested in for later computations.

  Whenever we have the same variable in both *IN* and *OUT* sets for an assignment node, it can be concluded that the variable was redefined in that assignment node.

TABLE 6.3: *IN* and *OUT* Sets for Assignment Nodes Only in Traversal 3

Node	IN	OUT
1	NULL	f1
2	f1	f1
3	f1	f1, f3
4	f1, f3	f1, f3, fc
5	f1, f3, fc	f1, f3, fc
6	f1, f3, fc	f1, f3, fc, f5
7	f1, f3, fc, f5	f1, f3, fc, f5

- *Traversal 4: Drawing the reaching definition edges and the USE-DEF chains*

  Once we have the *IN* and *OUT* sets we draw the reaching definition edges by actually forming the USE-DEF chains. This is a bottom-up traversal of the AST and only after this traversal do we complete our data-flow analysis.

TABLE 6.4: *USE-DEF* Chains in Traversal 4

Variable Name	Pointer Set
f1	(7, 2) , (2, 1)
f3	(7, 5), (5, 3)
fc	(5, 4)
f5	(7, 6)

Table 6.4 shows the pointer sets for each variable as *USE-DEF* chains. We also illustrate the same tabular representation in the AST in Figure 6.4. The blue edges represent the *USE-DEF* chains; such that the pointer from the node represents the consumer and the node to which the edge is directed to represents the producer.

We encountered that our algorithm and implementation removes certain undesirable reaching definition edges in some special cases. This is exhibited in example shown in Figure 6.5.

When we compute the reaching definitions for this piece of code, the resultant abstract syntax tree with the edges are shown in Figure 6.6.

In the example as shown, y is assigned a value outside any loop and then the value of y is assigned to $x[i][ij]$ inside the first nested loop and at the same time the value of y is changed to the current value of the innermost loop counter, j in our case. Similarly for the next block of nested loops, we assign the value of $x[i][j]$ to the summation of present value

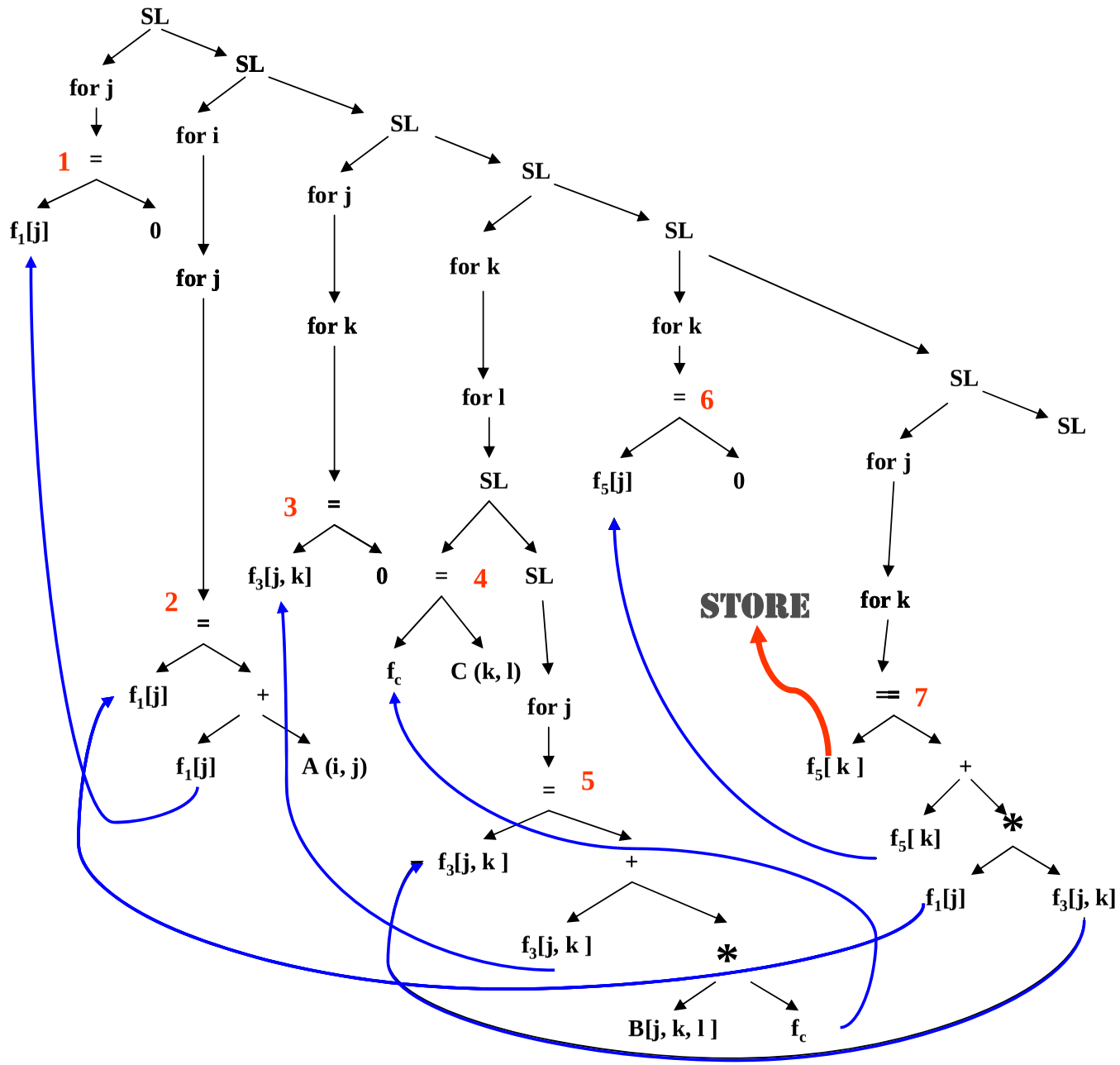


FIGURE 6.4: Abstract Syntax Tree with the *USE-DEF* Chains for the Example Code shown in Figure 6.1

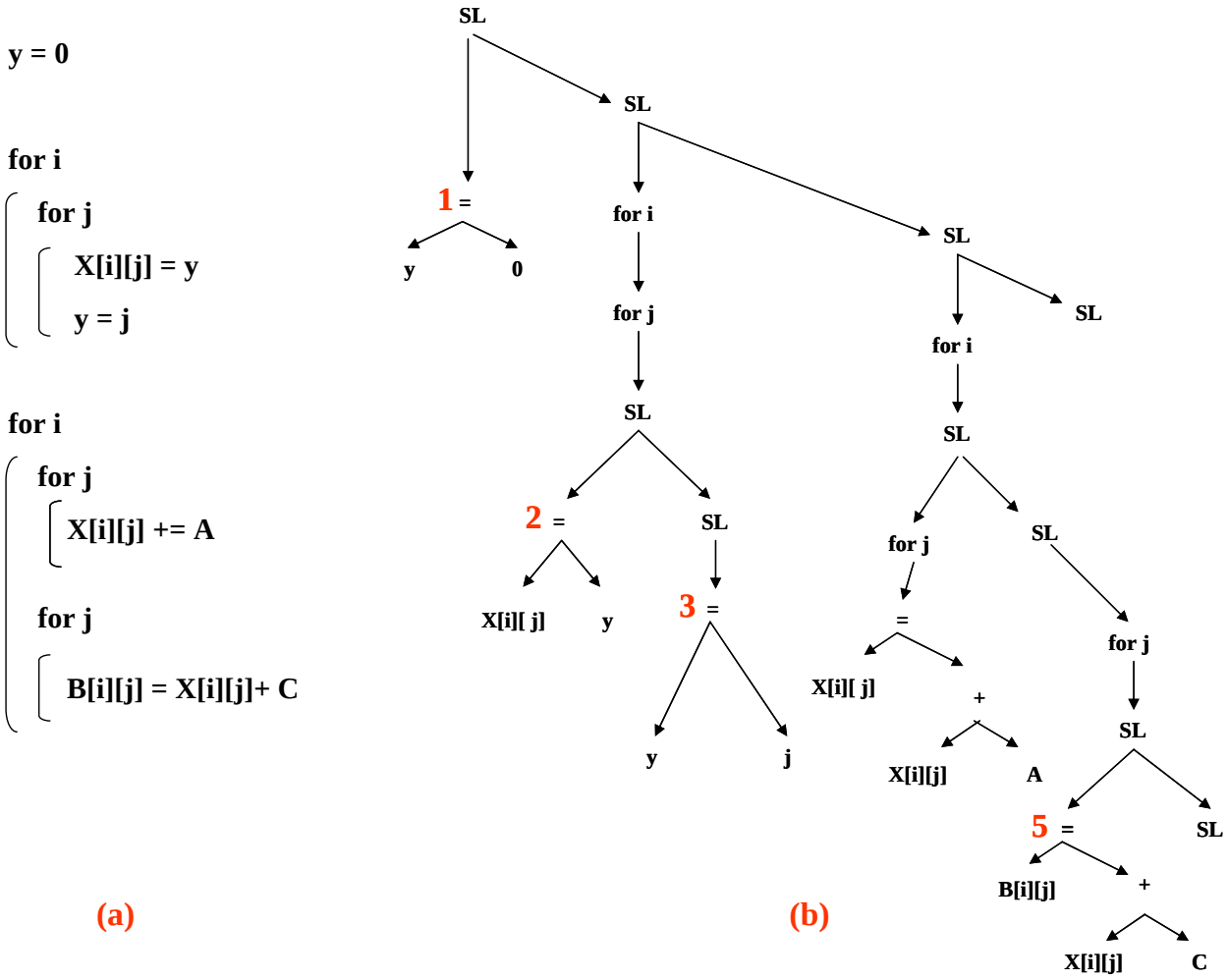


FIGURE 6.5: Example of Unfused Code and Equivalent AST

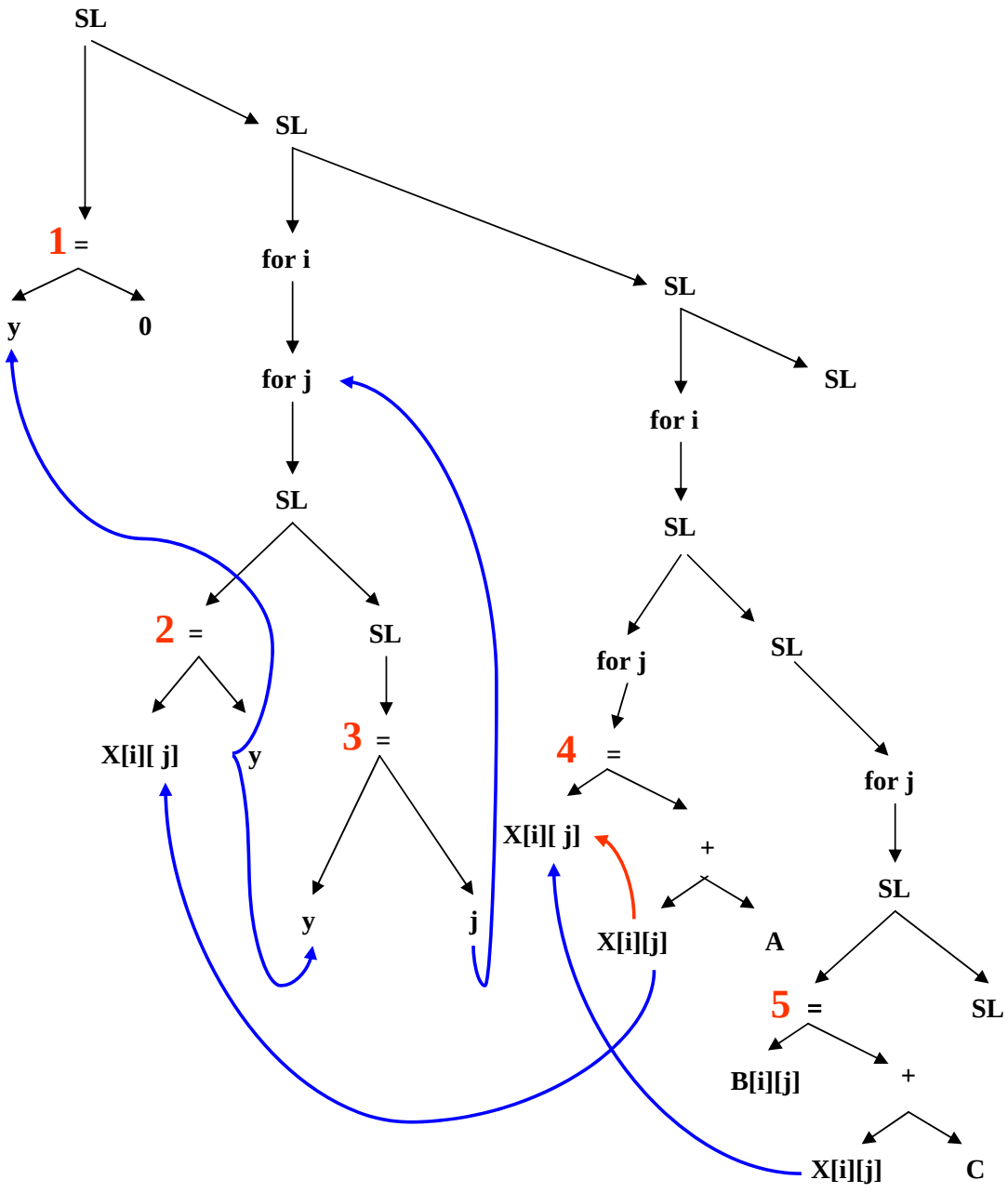


FIGURE 6.6: Reaching Definitions for Unfused Code used in Figure 6.5

of $x[i][j]$ and A. The reaching definition edge as shown in red, in Figure 6.6, is undesirable. This is because x is an array and we point to a node for the *def* which do not have the correct value. On the other hand, since y is not an array, we can have two *use-def* chains for y. We restrict possibilities of erroneous reference to a data that needs to be computed but has not been defined yet.

6.2 Loop Fusion

Once we have enough information by computation of the reaching definition edges, we are ready to fuse potential nodes, as explained in Chapter 5. We first compute the indices and dimensions using the algorithm described in Algorithm 6 and show the results for the same in Table 6.5.

TABLE 6.5: Computation of *Indices* and *Dimensions*

Node	Variable	Operation
1	f1[j]	PointerTemp = $\langle 2,1 \rangle$ DimTemp = 1 f1[j].Dim = $\langle j \rangle$
1	0	$\langle * \rangle$
1	Assignment	$\langle j \rangle$ *OR* $\langle * \rangle$
2	f1[j]	$\langle * \rangle$
2	A(i, j)	A.Dim = $\langle ij \rangle$
2	+	MergeNesting($\langle * \rangle$, $\langle ij \rangle$)
2	f1[j]	PointerTemp = $\langle 2,1 \rangle$ DimTemp = 1 f1[j].Dim = $\langle j \rangle$

Once the sizes of the arrays are computed, the abstract syntax tree has required information for fusing potential loops. As explained in Section 5, we have the reduction and extension in the Assignment Nodes and Binary Operation Nodes respectively. For an assignment node there are more indices on the RHS than on the LHS and hence we need to perform a reduction operation which removes unfusible indices and adjusts the cost accordingly. For Binary Operations, the parent has more dimensions than the child and hence, it is necessary to enumerate all possible ways in which the loop structures for the subtrees can be fused with the parent, such that the solutions for the two subtrees can be combined. The reduction and extension operation can be demonstarted using Figure 6.7.

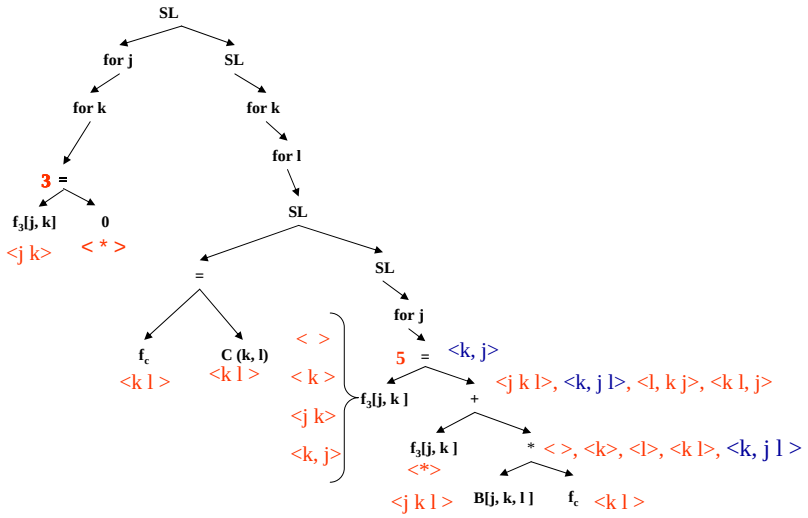


FIGURE 6.7: Example of Reduction & Extension Operations

We then use the memory minimization algorithm as used for the TCE module and we build the fused abstract syntax tree. Our algorithm thus as a first step develop an optimization framework which will later on be used to design the cost model and later on other potential extensions as discussed in Chapter 7.

7. Conclusion

The performance of a general purpose language is highly dependent on the code optimizer embedded in the compiler. For computationally intensive codes which require huge memory space, it is very convenient if the compiler generates the code for disk to memory traffic. It becomes easier for the programmer, if the compiler manages the intermediate temporary arrays in a fashion that minimizes memory and gives the programmer the abstraction of an infinite storage space. This thesis is a generalized use of the memory minimization algorithm used in the TCE. While the memory minimization module was applied on expression trees, derived from the mathematically much simpler tensor expressions, we implemented it on abstract syntax trees, which are produced by parsing handwritten code in a general purpose language. The expression trees are at a much higher level of abstraction compared to abstract syntax trees and the unpredictable nature of handwritten code makes our problem all the more challenging. We used standard data flow analysis techniques to track the use of variables and arrays in the tree. Once we have sufficient information about the lifetime of a variable, where it is being defined and used, and whether it needs to be stored or not, we apply the fusion algorithm on the tree. This gives us possible solution sets which are then pruned to find the optimial solution. We would like to implement tiling and integrate it with the fusion module to deliver better performance. We would also like to get rid of the assumptions we used in our work, namely allow common sub-expressions in the input code, use the polyhedral model to allow array index expressions to be more complex than simple loop variables. The most essential extension of our work in order to make it more generalized, would be to give the programmer the privilege of making loop bound expressions anything rather than constraining themselves to constants. After we are done with the above implementations, we would like to design an algebraic cost model to help us get the memory utilization numbers in a more generalized form and work for any arbitrary code structure. We would also like to implement tiling independently and combine both fusion and tiling outputs to build the integrated algebraic cost model discussed earlier.

The major contribution of this thesis and the ongoing research for this module is to programmers who code for scientific and engineering applications. These computationally intensive codes often require temporary intermediate arrays for breaking down huge mathematical expressions into smaller ones. Our compiler is designed to take care of such temporary storage, so that the programmer has the abstraction of infinite memory.

Bibliography

[1] A case for source-level transformations in matlab, 1999.

[2] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loops. In *Proceedings of International ACM Conference on Supercomputing*, Santa Fe, NM, 2000.

[3] Afred V. Aho, Jeffrey D.Ullman, Ravi Sethi, and Monica S. Lam. *Compilers-Priciples, Techniques and Tools*. Addison Wesley, second edition, 2006.

[4] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert van de Geijn, and Yuan-Jye J.Wu. Plapack: Parallel linear algebra package. In *Proceedings of the SIAM Parallel Processing Conference,*, 1997.

[5] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Petsc users manual. Technical Report ANL-95/11 Revision 2.1.2, Argonne National Laboratory, 2002.

[6] G. Baumgartner, D.E. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan. A high-level approach to synthesis of high-performance codes for quantum chemistry. In *Proceedings of the Supercomputing*, November 2002.

[7] G. Baumgartner, D.E. Bernholdt, D. Cociorva, R. Harrison, C. Lam, and P. Sadayappan M. Nooijen, J. Ramanujam. Automatic synthesis of high-performance codes for quantum chemistry applications. In *Proceedings of the Workshop on Performance Optimization for High-Level Languages and Libraries*, New York, June 2002.

[8] G. Baumgartner, D.E. Bernholdt, D. Cociorva, R. Harrison, C. Lam, M. Nooijen, J. Ramanujam, and P. Sadayappan. A performance optimization framework for compilation of tensor contraction expressions into parallel programs. In *Proceedings of the International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Fort Lauderdale, Florida, April 2002.

[9] A. Bibireata, S. Krishnan, D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, D.E. Bernholdt, and V. Choppella. Memory-constrained data locality optimization for tensor contractions. In *Proceedings of the 16th Workshop on Languages and Compilers for Parallel Computing*, College Station, Texas, October 2003.

[10] D. L. Brown, William D. Henshaw, and Daniel J. Quinlan. . overture: An object-oriented framework for solving partial differential equations on overlapping grids. In *Proceedings of the SIAM conference on Object Oriented Methods for Scientfic Computing*, 1999.

[11] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to upc and language specification. Technical Report CCS-TR-99-157, IDA, Center for Computing Sciences, 1999.

[12] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. Scalapack: A portable linear algebra library for distributed memory computers — design issues and performance. Technical Report CS-95-283, University of Tennessee, Knoxville, March 1995.

[13] R. Choy and A. Edelman. Parallel matlab: Doing it right. *Proceedings of IEEE*, 2:331–341, Feb 2005.

[14] D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, and J. Ramanujam. Memory-constrained communication minimization for a class of array computations. In *Proceedings of the 15th International Workshop on Languages and Compilers for Parallel Computing*, College Park, Maryland, July 2002.

[15] D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison. Space-time trade-off optimization for a class of electronic structure calculations. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 177–186, June 2002.

[16] D. Cociorva, J.Wilkins, C.-C. Lam, G. Baumgartner, P. Sadayappan, and J. Ramanujam. Loop optimization for a class of memory-constrained computations. In *Proceedings of the 15th ACM International Conference on Supercomputing*, pages 500–509, Sorrento, Italy, June 2001.

[17] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D.E. Bernholdt, and R. Harrison. Towards automatic synthesis of high-performance codes for electronic structure calculations: Data locality optimization. In *Proceedings of the Intl. Conf. on High Performance Computing, Lecture Notes in Computer Science*, pages 237–248. Springer-Verlag, 2001.

[18] S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jolla, CA, 1995.

[19] Compiling for NUMA parallel machines. *W. Li.* PhD thesis, Cornell University, Ithaca, NY, 1993.

[20] A. Fraboulet, G. Huard, and A. Mignotte. Loop alignment for memory access optimization. In *Proceedings of . 12th Int. Symp. System Synthesis*, pages 71–77, 1999.

[21] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of Workshop Languages and Compilers for Parallel Computing*, New Haven, CT, 1992.

[22] L. Guibas and D. Wyatt. Compilation and delayed evaluation in apl. In *Proceedings of 5th Annal. ACM Symposium on Principles of Programming Languages*, pages 1–8, 1978.

[23] Samuel Guyer and Calvin Lin. *Broadway: A Software Architecture for Scientific Computing.* Kluwer Academic Press, 2000.

[24] J. Johnson, R. Johnson, D. Padua, and J. Xiong. Searching for the best fft formulas with the spl compiler. In Springer-Verlag, editor, *Lecture Notes in Computer Science, Languages and Compilers for Parallel Computing*, pages 112–126, Heidelberg, Germany, 2001.

[25] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In Springer-Verlag, editor, *Lecture Notes in Computer Science, Languages and Compilers for Parallel Computing*, pages 301–320, Heidelberg, Germany, 1993.

[26] S. Khrishnan. Datalocality optimization for synthesis of out-of-core programs. Master's thesis, The Ohio State University, Columbus, OH, September 2003.

[27] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, D. Cociorva, P. Sadayappan C. Lam, J. Ramanujam, D.E. Bernholdt, and V. Choppella. Data locality optimization for synthesis of efficient out-of-core algorithms. In *Proceedings of the the Intl. Conf. on High Performance Computing*, Hyderabad, India, December 2003.

[28] C. Lam. *Performance Optimization of a Class of Loops Implementing Multi-Dimensional Integrals*. PhD thesis, The Ohio State University, Columbus, OH, August 1999.

[29] C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan. Optimization of memory usage and communication requirements for a class of loops implementing multi-dimensional integrals. In Lecture Notes in Computer Science Springer-Verlag, editor, *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pages 350–364, La Jolla, California, August 1999.

[30] C. Lam, P. Sadayappan, and R. Wenger. On optimizing a class of multi-dimensional loops with reductions for parallel execution. *Parallel Processing Letters*, 7(2):157–168, 1997.

[31] Chi-Chung Lam, Daniel Cociorva, Gerald Baumgartner, and P. Sadayappan. Optimization of memory usage requirement for a class of loops implementing multi-dimensional integrals. In *Languages and Compilers for Parallel Computing*, pages 350–364, 1999.

[32] A. Lim, S. Liao, and M. Lam. Blocking and array contraction across arbitrarily nested loops using ane partitioning. In *Proceedings of ACM SIGPLAN Symp. Principles and Practices of Parallel Programming*, 103–112 2001.

[33] N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. In *Proceedings of Int. Conf. Parallel Processing*, page II:19II:28, 1995.

[34] K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst*, 18(4), July 1996.

[35] V. Menon and K. Pingali. High-level semantic optimization of numerical codes. In *Proceedings of 13th ACMInt. Conf. Supercomputing*, pages 434–443, '1999.

[36] D. Mirkovic and L. Johnsson. Automatic performance tuning in the uhfft library. In *Proceedings of the . International Conference on Computational Science, Lecture Notes in Computer Science*, volume 2073, pages 71–80. Springer-Verlag, 2001.

[37] J. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, M. Puschel, and M. Veloso. Spiral: Portable library of optimized signal processing. http://www.ece.cmu.edu/spiral.

[38] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *Journal of Supercomputing*, 10:197–220, 1996.

[39] R.W. Numrich and J.K. Reid. Co-array fortran for parallel programming. *Fortran Forum*, 17(2), 1998.

[40] G. Pike and P. Hilfinger. Better tiling and array contraction for compiling scientific programs. In *Proceedings of ACM/IEEE Conf. Supercomputing : High Performance Networking and Computing*, pages 1–12, 2002.

[41] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J.Xiong, F. Franchetti, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. 93(2):232–275, February 2005.

[42] Matei Ripeanu and Adriana Iamnitchi. Cactus application: Performance predictions in grid environments. In *Proceedings of the EuroPar, Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[43] . De Rose and D. Padua. A matlab to fortran 90 translator and its effectiveness. In *Proceedings of International ACM Conference on Supercomputing*, pages 309–316, 1996.

[44] S. Singhai and K. McKinley. Loop fusion for data locality and parallelism. In *Proceedings of the Mid-Atlantic Student Workshop on Programming Languages and Systems*, , New Paltz, NY, 1996.

[45] L. Snyder. *A Programmers Guide to ZPL*. The MIT Press, 1999.

[46] M. Strout, L. Carter andJ. Ferrante, and B. Simon. Schedule-independent storage mapping for loops. In *Proceedings of the 8th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 24–33, 1998.

[47] M. Wolf and M. Lam. A data locality optimization algorithm. In *Proceedings of SIGPLAN Conf. Programming Language Design and Implementation*, pages 30–44, 1991.

[48] J. Xiong, D. Padua, and J. Johnson. Spl: A language and compiler for dsp algorithms. In *Proceedings of ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 293–308, 2001.

[49] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, (10):11–13, September-November 1998.

[50] K. Yotov, G. Ren X. Li, M. Cibulskis, G. DeJong, D. Padua M. Garzaran, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *Proceedings of ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 63–76, 2003.

Vita

Pamela Bhattacharya is the daughter of Dr. Pranab Sankar Bhattacharya and Mrs.Nupur Bhattacharya. She was born in 1985 at Chinsurah, a small town in the district of Hooghly in the state of West Bengal in India. Pamela graduated with Bachelor of Technology (Honors) in Information Technology degree from West Bengal University of Technology in 2006. She worked as a Programmer Analyst for Cognizant Technology Solutions, Kolkata, before she began her graduate studies as a doctoral student in the Spring of 2007 in the Department of Computer Science, Louisiana State University (LSU). She completed her thesis under the able guidance of Prof. Gerald Baumgartner and will graduate with the degree of Master in Science from LSU in August, 2008. She will join the doctoral program in computer science at University of California, Riverside, from Fall 2008.