

## ABSTRACT

Title of thesis: AN INTEGRATED  
PROGRAM REPRESENTATION  
FOR LOOP OPTIMIZATIONS

Greeshma Yellareddy, Master of Science, 2011

Thesis directed by: Professor Rajeev Barua  
Department of Electrical and Computer Engineering

Inspite of all the advances, automatic parallelization has not entered the general purpose compiling environment for several reasons.

- There have been two distinct schools of thought in parallelization domain namely, affine and non-affine which have remained incompatible with each other over the years. Thus, a good practical compiler will have to be able to analyze and parallelize any type of code – affine or non-affine or a mix of both.
- To be able to achieve the best performance, compilers will have to derive the order of transformations best suitable for a given program on a given system. This problem, known as “Phase Ordering”, is a very crucial impedance for practical compilers, more so for parallelizing compilers. The ideal compiler should be able to consider various orders of transformations and reason about the performance benefits of the same.

In order to achieve such a compiler, in this paper, we propose a unified program representation which has the following characteristics:

- Modular in nature.
- Ability to represent both affine and non-affine transformations.
- Ability to use detailed static run-time estimators directly on the representation.

AN INTEGRATED  
PROGRAM REPRESENTATION  
FOR LOOP OPTIMIZATIONS

by

Greeshma Yellareddy

Thesis submitted to the Faculty of the Graduate School of the  
University of Maryland, College Park in partial fulfillment  
of the requirements for the degree of  
Master of Science  
2011

Advisory Committee:  
Professor Rajeev Barua, Chair/Advisor  
Professor Bruce Jacob  
Professor Shuvra Bhattacharyya

© Copyright by  
Greeshma Yellareddy  
2011

# Table of Contents

List of Tables	iii
List of Figures	iv
1 Introduction	1
2 Related Work	11
3 Program Representation	15
3.1 Overview . . . . .	15
3.2 PDG . . . . .	17
3.3 LNG . . . . .	21
4 Representation of Transformations	26
4.1 Loop Interchange . . . . .	26
4.2 Loop Fusion . . . . .	29
4.3 Reduction . . . . .	32
4.4 DSWP . . . . .	33
5 Results	37
5 Conclusion	42
Bibliography	43

## List of Tables

5.1	Speedup on x86 24-core machine for <i>Polybench</i> benchmark . . . . .	38
5.2	Speedup for Affine and Non-affine Transformations . . . . .	39

## List of Figures

3.1	Example program with the PDG and LNG . . . . .	18
4.1	Example for Interchange . . . . .	29
4.2	Example for Fusion . . . . .	30
4.3	Example for DSWP . . . . .	34
4.4	PDG and LNG for DSWP . . . . .	35
5.1	Example for Affine and Non-affine Transformations . . . . .	40

## Chapter 1

### Introduction

In the multi-core era, most of the world's computers are parallel, but most software remains serial. Given the huge investment in existing serial code worldwide, and that rewriting serial code into parallel code is time-consuming, error-prone, and expensive, there is a great need for automatic parallelization and cache-optimization of serial code. Some success has been seen for programs with affine array references [18][5] – an array reference is said to be *affine* if its indices are linear combinations of loop induction variables. For example,  $A[2i + 3j + 7, i - 2]$  is affine, but  $A[i^2]$  is not. Affine accesses are particularly well suited for parallelism and cache optimizations. However, only scientific and media programs are predominantly affine; and even those sometimes have small amounts of non-affine code in otherwise affine loops, ruining performance. Non-affine methods have also seen some work [6][12] primarily using graph-based methods.

Despite decades of research, the results of automatic parallelization remain somewhat disappointing. We identify two main reasons. First, existing affine and non-affine methods remain fundamentally incompatible because of different internal representations of candidate transformed code. Since most real-world code is a mix of affine and non-affine code, neither class of parallelizers has been able to conquer general-purpose code. Second, and just as important, existing compiled parallel



code finds it hard to attain even a reasonable fraction of the peak performance of a parallel computer. Parallel code that is manually tuned by a highly trained computer engineer almost always yields code that is superior, often by a large margin, compared to automatically parallelized code.

The root cause of the superior performance of manually tuned code is that a human programmer is often able to deduce program optimization transformations that a compiler is unable to find. In theory, the compiler should be able to find the transformation order chosen by the human – in most cases, both use the same toolkit of program transformations to achieve lower run-time. Such human-applied or compiler affine transformations include tiling, loop interchange, fission, fusion, reversal, skewing, interleaving, peeling, and strip-mining. Non-affine transformations [12][13] include Decoupled Software Pipelining (DSWP), Parallel-stage DSWP (PS-DSWP), Cyclic Multi-threading (CMT+) and Control Speculation. These transformations are extremely crucial to performance to increase parallelism by breaking loop-carried dependencies, or improving cache locality, or both.

Unfortunately, the problem of finding a good transformation order and transformation parameters (e.g., tile sizes) is a challenging problem. This is the well-known “phase ordering” problem in compilers: a sequence of transformations must be applied in precisely the right order, with the right parameter values if applicable, to exactly the same loop dimensions to achieve the best run-time. Although this is a somewhat important problem for serial program transformations, it is crucial for transformations used for parallelization. For example, we have encountered a loop in the *gemver* benchmark in the Polybench benchmark suite for which the

best transformation order is a loop interchange, followed by a fusion, followed by a strip-mining, followed by another loop interchange to different dimensions . No shorter transformation order gives a run-time that is even close to the best order. We have found several benchmarks where the run-time of a basic parallelizer without transformations is improved by a factor of 5-10X by using a carefully constructed, manually discovered sequence of transformations.

To effectively solve the phase-ordering problem, a compiler must be able to apply mixed sequences of affine and non-affine code together. This is because real-world programs often belie easy categorization as just “non-affine” or just “affine”. In reality, most programs are a mix of affine and non-affine code, with the ratio being more affine code in scientific domains, and less so in the general-purpose domain. For example, scientific codes are often mostly affine, but may have small amounts of non-affine code that may introduce loop-carried dependencies, such as a *printf()* or other calls with side effects, pointer accesses, or heap data structure accesses. Existing affine compilers today tend to be very limited in the scope of programs they can transform. They are infamously “brittle”, breaking on codes that were not written to perfectly match what they can handle. The result is that affine parallelizers fail on such codes.

Non-affine parallelizers are also hobbled – they can handle any code, but cannot apply affine transformations. This limits them since affine methods can exploit such codes in a much more scalable fashion. The result is that non-affine methods yield lower performance than is possible with an truly integrated affine + non-affine method.

The above two shortcomings make it clear that combined compiler methods to handle affine and non-affine transformations together are desirable since they are likely to achieve higher performance of the compiled code. However composing both types of transformations in a single compiler to be used simultaneously on the same program loop is very challenging. Existing compilers do not do this.

The fundamental difficulty in integrating affine and non-affine transformations is that they use different and incompatible program representations. Good program representations are essential to the success of compilers. For example, the control-flow graph and data flow representations such as SSA are invaluable for optimization even though they can be derived from the compiler's intermediate representation (IR). This is because they make underlying control- and data-flow information explicit, thus decoupling the discovery of such information from its use. Such decoupling allows for the modular design of a compiler, and avoids unnecessary re-computation of analysis results in every optimization pass. Similarly, the well-known Program Dependence Graph (PDG) is widely used for parallelization since it makes both types of dependences – control and data – explicit. This is useful even though everything in the PDG can be derived from the control-flow graph and dataflow representations.

In a similar vein, there is a need for a unified program representation that can make explicit the information needed for both affine and non-affine transformations. Unfortunately, existing program representations for both are unusable by the other. As a result, affine and non-affine transformations remain hard to combine. To understand why, we consider both types of representations below.

Affine mathematical representations represent affine array references and loop-carried dependencies between them using custom mathematical representations for affine code, such as matrices in traditional methods [18, 7], and systems of linear inequalities in polyhedral methods [5]. These representations have the advantage that they contain detailed information needed to apply affine transformations. Further since they contain little other information, they are fast to update when the compiler is searching through a very large number of possible transformation sequences. The loop's IR is converted to the mathematical representation before the transformation search commences. Thereafter the IR is not used during the search. Only when the search has found the best transformation sequence is the IR regenerated so that code generation can commence from it.

Unfortunately after an affine transformation is applied, existing affine representations do not represent the program explicitly at all. Indeed, the result of applying an affine matrix transformation is not represented in any general-purpose representation that non-affine transformations can reason about. Instead affine-transformed codes only exist in mathematical representation, with no concrete representation until final code generation. As a result, if a non-affine transformation is to analyze the result of a set of affine transformations, it has to derive the resultant code by applying these transformations in the IR and then building the relevant program representation it depends on from scratch. This is clumsy, non-intuitive, unnatural and defeats modularity. *To build a truly integrated compiler, it would be desirable if both affine and non-affine transformations explicitly represented code in a way that is useful for either types of subsequent transformations, so a seamless transformation*

*search is possible. This is the goal of our unified representation.*

Another drawback of affine mathematical representations is that they are not complete program representations – they only represent affine references and their loop-carried dependencies, but not the rest of the loop. Hence a general-purpose tool to estimate the run-time of a code fragment at compile-time cannot be applied after an affine transform. Such *static run-time estimators (SREs)* can be a useful tool to compare candidate transformation sequences to choose the best.

**Non-affine representations** There is less standardization in the literature on representations for non-affine transformations. Methods such as DSWP [12] use the well-known Program Dependence Graph (PDG) representation. Earlier methods such as those in the Paraphrase compiler use the Closure of Data and control Dependencies Graph (CDDG) [11, 10]. Both the PDG and CDDG represent instructions in the program along with data and control dependencies between them. They are suitable for non-affine transformations which often attempt to break (or speculate on) dependencies, so applying transformations is quick at compile-time since they usually only need to modify affected dependencies. Hence transformation orders can be searched for quickly.

Unfortunately graph based representations such as the PDG have their own drawback: they do not represent affine array indices or affine distance vectors, crucial for performing affine transformations. Further, there is no easy means to represent affine transformations such as loop interchange which changes the way a loop-nest is structured or loop reversal which changes the direction a loop is accessed be-

cause such information is not explicit in the PDG. The PDG has no special way of representing structured loops such as for loops with bounds, induction variables and increments. Whereas this kind of information may not be required for pure non-affine based techniques, it is essential for affine analysis.

Hence graph-based representations are not suitable for applying mixed sequences containing both affine and non-affine transformations either. They have no quick way of deriving affine information required for affine analysis. They have no quick way of updating affine information after a non-affine transformation. Of course the IR can be rewritten to regenerate affine information after each transformation in the search, but that would be too slow to be feasible given the large number of transformations a search might attempt.

**Discussion** The end result of the drawbacks of the existing representations above is that there is no unified explicit program representation today that is suitable for applying both affine and non-affine transformations in a unified manner. There is a need for a single integrated compiler framework that enables building parallelizing and cache-optimizing compilers for any type of code – affine code, non-affine code, or any mix of the two. It should robustly apply the best possible transformation order for each program by quickly searching through a large number of transformation sequences. A unified explicit program representation would greatly facilitate this goal.

We propose a new representation for programs that maintains the results of both affine and non-affine transforms explicitly, so that either type of transform

can use it for subsequent transforms. Based on the combination of the new loop representation called the Loop Nest Graph (LNG) and the well-known Program Dependence Graph (PDG), it has several features that are desirable when searching for program transformations in any compiler, but particularly parallelizing or cache-optimizing compilers. While the PDG maintains information regarding instructions *within* a loop and non-affine dependencies between these instructions, the LNG (which is built on top of the PDG) maintains information regarding the loops in a loop-nest and affine dependencies across the loops in the loop-nest. The advantages of this representation include:

- **Modular** One of the biggest advantages of using the LNG + PDG representation is the decoupling of the building of the program representation phase and the analysis phase. Intuitively, this is similar to how a PDG decouples parallelization methods from methods to construct the PDG. It is also similar to how an SSA representation decouples dataflow optimizations from SSA construction. As a result, the analysis phase can work on building the algorithms to use this information directly without having to deal with building individual representations themselves.
- **Recording the effects of Transformations directly** Another advantage of our representation is the ability to reflect the effects of both the affine and non-affine transformations in a unified framework such that any future analysis can use the information directly, instead of going back to IR. We do this by recording the transformations directly in the LNG + PDG representation. The

updates however are not intended to maintain complete information about the transformations, and as a result, do minimal changes often updating only one of the two structures.

- **Use in detailed SREs** The LNG + PDG forms a natural representation to be used by a detailed Static Runtime Estimator(SRE) to estimate run-time taking into account detailed system characteristics such as cache, memory, network, synchronization, and pipeline characteristics. Use of an SRE helps the compiler evaluate and choose between different compositions of transformations. Using a compiler framework capable of deriving candidate orders of transformations, and using a good SRE to choose between them, we can foresee automatic compilation results approaching the performance of manually tuned code by using the characteristics of the target machine in transformation search.

Current program representations have limitations that inhibit their use by SREs. Affine mathematical representations such as the distance vectors and polyhedral representations are not complete program representations – they only represent affine references and their loop-carried dependencies, but not the rest of the loop. As a result, SREs cannot be applied on them. On the other hand, although the PDG maintains complete enough information about the instructions in a loop, the information about structured **for** loops needed for affine run-time estimation, such as loop-nest structure and bounds, is not explicit in the PDG. Instead the SRE would need to recover it, violating the modular design of SREs and PDGs. In contrast the LNG and PDG combination has explicit information



for both types of parallelizers, enabling modular SREs.

Although the focus of this paper is not the transformation search but the program representation used for it, a prototype search method using our representation has been built. Results show that it finds transformation orders among a selection of both affine and non-affine transformations without going back to the IR.

## Chapter 2

### Related Work

Compilers rely on low overhead program representations when they need to evaluate different orders of transformations. Several program representations have been proposed for parallel compilers to aid with analysis and representation of transformations. These representations are motivated by the kind of analysis and the kind of loop optimizations considered by the compilers.

An interesting variant to this approach of using program transformations is iterative compilation [17, 1]. Instead of relying on any common intermediate program representation to derive and evaluate different compositions of transformations, iterative compilation strategy involves generating code for various orders of transformations and executing it to compare the performance. Since it involves going back to the IR for every candidate order of transformations, iterative compilation can have a significant compile time overhead. Although some methods reduce the compile time by bringing down the number of choices considered by the use of limited set of heuristics [17] or machine learning algorithms [1], the compile time however is still significant enough to restrict its use in practical compilers. Our use of a low overhead means of representing just the essential information in the program, helps us overcome this high compile time problem. *With an accurate SRE, our representation will provide the same benefits as iterative compilation with orders of*

*magnitude less compile-time.*

For affine programs, distance vectors [20, 3] have been used by the SUIF compiler [18] to analyze and parallelize affine code. A distance vector represents a memory dependence between two array accesses across the iteration space of their common loop nests. For example the distance vector (2,0) means that each outer loop iteration depends on an iteration that is two before it, and the inner loop iterations are independent. The collective set of such distance vectors for every memory dependence in the loop forms the basis for analyzing the loop for various affine loop transformations such as loop interchange, loop reversal and loop skewing. One of the biggest advantages of this representation is that analysis and representation of the above affine loop transformations can be abstracted into simple matrix transformations.

Unfortunately, one limitation of the representation is that merely updating distance vectors cannot represent the effect of non-unimodular loop transformations such as loop fusion, loop fission and loop peeling. As a result, heuristics which are based solely on the distance vectors for evaluation of different orders of transformations cannot use these transformations directly. Wolf *et al.* [19] propose an algorithm where fission and fusion are considered in the initial and final phases, while the middle phase of the algorithm uses the distance vectors to compute and evaluate various orders of transformations. Thus, these transformations are never truly part of evaluation of various orders. For example, this model cannot come up with a transformation order such as interchange followed by fusion followed by interchange. In our representation, we use distance vectors in combination with the

LNG and the PDG to create a hierarchical loop dependence graph, through which we can represent all these transformations in any order.

The polyhedral representation [4] is the other significant program representation for affine code, which has been used by PLUTO compiler [5]. Polyhedral methods [8] represent each statement in an affine loop separately as a point in an iteration domain. The iteration space thus defined is a polyhedron in a  $d$ -dimensional space, where  $d$  is the nesting depth of the loop in question. The polyhedral representation represents complex compositions of affine transformations as a scheduling function. Although the Polyhedral model is very powerful, it has a serious drawback – since it translates affine code into a series of linear inequalities, this mathematical representation has no way of representing non-affine code. As a result it is inapplicable to mostly-affine code which has small amounts of non-affine code – this is common in real-world codes. Further, we will be able to represent every transformation that can be represented in Polyhedral framework by means of the LNG + PDG combination. Indeed many of the program transformations we found in polyhedral papers that they said are not handled by the traditional model can be found by using a mixed series of unimodular and non-unimodular transformations we considered using our representation.

Dependence-graph representations such as the PDG [9], the CDDG [11, 10] and the parallel program graph [16] are often used by non-affine transformations. The Paraphrase-2 [15] compiler uses the CDDG to partition and schedule a given program. The CDDG is a hierarchical structure that enables detecting parallel tasks at various levels; however it is an immensely complicated structure to make changes

to and thus is not suitable for representing transformations directly in the program representation, which is an essential feature if we want to apply several transformations. The PDG is another dependence-graph representation that is particularly conducive for incremental optimizations. The PDG encapsulates useful information that helps detect opportunities for optimizations, vectorization and parallelization. The main drawback of all graph-based representations is that they are restricted to non-affine transformations; for example, the PDG has been used for transformations such as DSWP, PS-DSWP and Control Speculation [12, 13]. We extend the PDG for affine analysis by combining it with the LNG.

A related representation is the PPG [16] which builds on the PDG for representing already parallel programs. The PPG extends the serial representation of PDG by adding parallel control flow edges and synchronization edges for parallel programs. Unlike the PPG, our aim is to be able to detect parallelism opportunities rather than represent parallel programs. Hence our representation is catered to representing the serial program. Through this representation of LNG + PDG, we seek to extend the PDG representation for encompassing different kinds of transformations.

## Chapter 3

### Program Representation

#### 3.1 Overview

The most important and challenging factor motivating the development of our program representation is the ability to analyze and represent both affine and non-affine transformations. Affine and non-affine methods have distinctly different characteristics. Whereas affine analyses use mathematical representations such as distance vectors to analyze and represent transformations, non-affine analyses rely on the use of dependence graphs such as the CDDG and the PDG. Lack of any obvious commonality between the two methods, as observed by the existing technologies, poses a major challenge.

Understanding the commonalities and the differences in the features that the affine and non-affine transformations look for is the key to a unified representation. A good representation relies on the ability to make explicit the information needed by both these domains. Though the kind of analyses and transformations considered by the two domains are seemingly different, they rely on a common feature of being able to understand the inherent dependencies in the program and more particularly within a loop. The difference lies in the way the dependencies are studied and interpreted by the different transformations. Non-affine transformations analyze a dependence graph to partition it into several independent or less-dependent tasks.

Affine transformations, on the other hand aim to run loop iterations in parallel with other iterations by exploiting regular dependence patterns encapsulated in distance vectors. To do so, they use loop restructuring, iteration space reordering and loop-nest reordering. As a result, non-affine transformations require the ability to understand all the dependencies *within* a loop; affine transformations require the ability to understand and represent the characteristics relevant to the loop-nest such as the loop-nest structure and the loop-carried dependencies *across* all the different levels of the loop-nest. Thus the difference in the representations needed.

There is a need for a representation that can act as a bridge between these two distinct set of needs helping realize a truly integrated program representation. The integrated program representation should be capable of not only maintaining the information relevant to both the affine and the non-affine methods but also represent the effect of transformations without going back to IR. Current representations are catered for either affine or non-affine methods exclusively because of which they maintain information relevant only for these methods. Combining the two representations – the PDG and distance vectors – though seems intuitive, is also insufficient for our purpose. Neither the PDG nor the distance vectors can reflect the changes of the other set of transformations directly. The two representations are inherently incompatible with each other.

We propose a new loop representation, the Loop Nest Graph (LNG) which in combination with the PDG and distance vectors has all the above features. The LNG acts as the bridge between the affine-catering mathematical representation (distance vectors) and the dependence graph representation (PDG). By representing every

loop as an individual node, the LNG represents a graph of dependencies between the loops in a loop-nest, thus naturally representing the nesting structure in an easily accessible way. Because of its reliance on loops as individual nodes, the LNG also forms the natural representation to maintain affine information such as loop characteristics for structured loops and distance vectors as annotations on loop-carried affine dependencies. This representation therefore serves the dual purpose of representing information required by the affine analysis, as well as representing the effects of affine transformations in a way that non-affine analyses can understand and update.

This representation forms a two-tiered structure. The PDG is at the lower tier representing dependence information between individual instructions in a loop which is useful for non-affine analysis. The LNG in combination with distance vectors is at the top tier maintaining loop and dependence information in a loop-nest which the affine analysis can use effectively. Thus combining the LNG with the PDG and distance vectors gives us the flexibility to take advantage of existing research with little or no modification.

## 3.2 PDG

The PDG [9] is a well-known and powerful representation which is suited for several loop optimizations and parallelization techniques. The PDG has been used for several loop transformations such as Loop Peeling and Loop Unrolling. It is also the basis for parallelization techniques such as DSWP [12]. An important



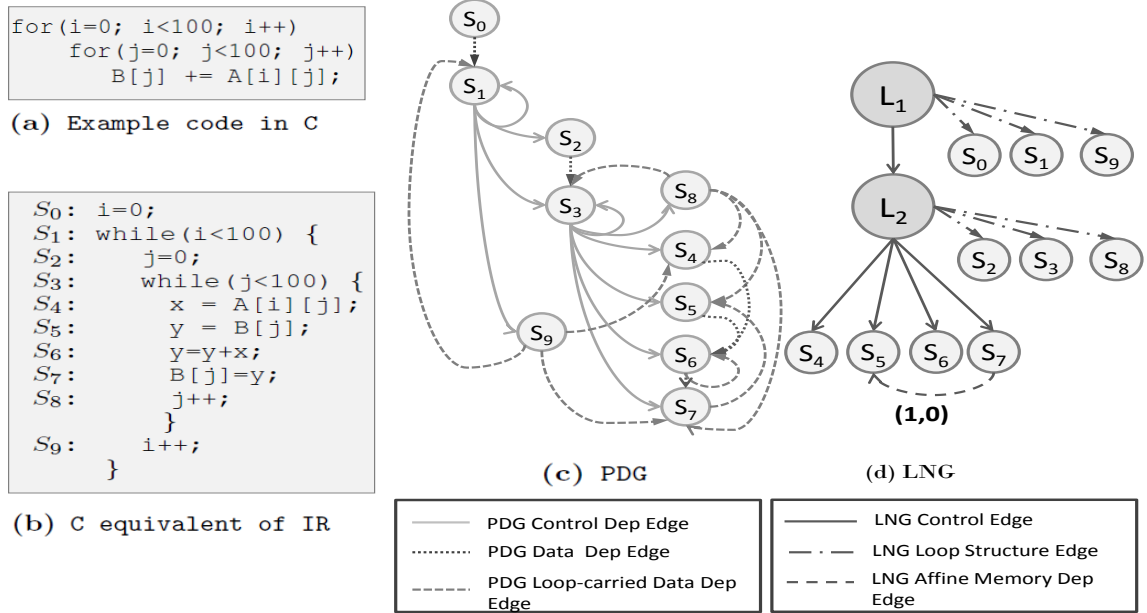


Figure 3.1: Example program with the PDG and LNG

advantage of the PDG is that it can be used to incrementally apply several of these optimizations without having to go to the IR.

The PDG has several characteristics that makes it conducive to the optimizations we are interested in. One of the most important reasons the PDG is particularly relevant is its reliance on dependence information. In basic terms, the presence of a dependence between two instructions implies that there exists constraints regarding the execution of the two instructions, and the absence of a dependence implies that the two instructions can execute in parallel. As a result, dependencies play a crucial role in exposing parallelism. Since the program dependence graph captures dependencies in a program instead of emphasizing the flow of control or data, it becomes a natural choice for a compiler interested in parallelization.

Another feature of the PDG that is useful is its ability to represent the dependencies at any level we are interested in – program level, function level and loop level. Every level captures dependencies relevant only to that level’s view. For example, a function level view of the PDG can capture dependencies between two loops, but may not capture dependencies such as loop-carried dependencies within each of the loops. In our representation, we use the PDG to represent dependencies for every loop in the function (including each level within a loop nest).

The loop PDG is the base layer of our representation, maintaining detailed dependence information of the loop which may be required by certain analyses. Every loop in the function is associated with a PDG representation that maintains dependence information between every pair of instructions in the loop.

The loop PDG represents the loop as a graph where every instruction in the loop is a unique node and edges represent the dependencies between the nodes.

The PDG is a directed graph  $G_{PDG} = (V_{PDG}, E_{PDG})$  |

$V_{PDG} = \{S_0, S_1 \dots, S_n\}$  where  $S_i$  is an instruction in the loop

$E_{PDG} = \{(u, v) \mid \forall u, v \in V_{PDG} \text{ and } v \text{ is dependent on } u\}$

3.1(c) shows the PDG for the loop in example 3.1(a). Each instruction in the IR shown in the example 3.1(b) is a node in the PDG. In this example,  $\{S_0, S_1 \dots, S_9\}$  represents the set of nodes in the PDG.  $(S_1, S_2)$  is an example edge that represents the dependence of the instruction  $S_2$  on  $S_1$ .

Different kinds of dependencies are represented in the PDG. Each edge  $(u, v)$  in the PDG is associated with one of these dependence types.

- **PDG Control Dependence Edge:** ( $E_{PDG_{CD}}$ ) A node  $v$  is control dependent on node  $u$  if the node  $u$  determines whether node  $v$  executes or not. Use of control dependencies instead of control flow helps avoid the fixed sequencing forced by the latter. The control dependence property is derived from the control flow graph of the loop [9]. For the control flow graph,  $G_{CFG}$  of a loop

$(u, v) \in E_{PDG_{CD}}$  is a PDG control dependence edge *iff*

- $\exists$  a path  $P$  from  $u$  to  $v$  in  $G_{CFG}$  with any  $w \in P$  *post-dominated* by  $v$  and
- $u$  is not *post-dominated* by  $v$ .

In the Example 3.1(c),  $\{(S_1, S_2), (S_3, S_5)\}$  are some examples of control dependence edges in the PDG.

- **PDG Data Dependence Edge:** ( $E_{PDG_{DD}}$ ) A data dependence edge denotes the possibility of both the instructions (nodes) accessing the same location. This automatically imposes execution constraints, thus restricting parallelism. Hence data-dependencies form important constraints in parallelism detection. Since data dependencies, especially the memory dependencies are difficult to disambiguate, they often are the most crucial parallelism-inhibiting dependencies. Several transformations specifically target breaking or moving these dependencies to enhance parallelism.

$(u, v) \in E_{PDG_{DD}}$  is a PDG data dependence edge *iff*

- $\exists$  a path from  $u$  to  $v$  in  $G_{CFG}$  and
- $u$  and  $v$  can access the same location and
- either  $u$  or  $v$  or both are a write or store operation.

Data dependence edges are further classified into *register dependence* and *memory dependence* edges.  $(u, v)$  is a register dependence edge if  $u$  writes to a register  $v$  reads from<sup>1</sup>. In the example 3.1(c),  $(S_4, S_6)$  is an example register dependence edge.  $(u, v)$  is a memory dependence edge if  $u$  writes to a memory location that could be read by  $v$ . In the example 3.1(c),  $(S_7, S_5)$  is a memory anti-dependence edge.

Data dependence edges are determined using data-flow analysis techniques and the control flow graph. Eliminating memory dependencies further requires several kinds of memory dependence analyses, alias analyses and memory disambiguation techniques to prove that a pair of memory operations do not alias.

**PDG Loop-carried Data Dependence Edge** ( $E_{PDGLCDD}$ ) is a special kind of data dependence edge in the loop PDG which has the additional property of being loop-carried in nature.  $(S_9, S_4)$  is an example of a loop-carried register-dependence edge and  $(S_7, S_5)$  of a loop-carried memory-dependence edge for the memory location  $B[j]$ .

### 3.3 LNG

The LNG is a proposed top-level graph which defines the loop structure of a function. It forms a layer over the loop PDGs, providing information from a loop-nest perspective. The LNG has the following three features :

- Represents every loop as an individual node, thus making loops as top-level

---

<sup>1</sup>Here registers mean virtual registers before register allocation, or physical registers after

structures making them directly accessible for any loop related information. Representing the loop as a node also helps define the loop-nest structure, which is a useful representation for several affine transformations.

- Makes structured loops explicit. Structured loops such as **for** loops are the prime candidates for affine analyses. Several characteristics of the structured loops such as induction variables and loop bounds can be derived precisely and are hence represented directly.
- Encapsulates affine dependencies. For every PDG loop-carried memory dependence edge that is affine, we copy that edge to the LNG and annotate it with its distance vector. This is convenient for affine transformations since unimodular transformations can be defined solely based on the distance vectors of the loop. In addition, non-unimodular transformations can be represented by updating other parts of the LNG+ PDG in addition to the distance vectors.

The LNG thus forms a generic program representation representing dependence information from a loop-nest perspective, which is particularly relevant to affine analysis. The biggest advantage of the LNG is its easy compatibility with other dependence graph structures such as the PDG. Any analysis that needs loop-nest related information and specific loop attributes can directly use the LNG without having to derive it from the IR or the PDG.

The LNG represents the loop nest as a graph where every loop and every instruction is a unique node, with edges that define the loop-nesting relationship and affine memory dependence relationship between two nodes.

The LNG is a directed graph  $G_{LNG} = (V_{LNG}, E_{LNG})$  |

$V_{LNG}$  is a set of nodes, one for each instruction and loop in the program; and

$E_{LNG}$  is a set of edges  $(u, v)$  which have either (a) a loop-nest relationship as defined below or (b) an affine memory dependence.

3.1(d) shows the LNG for the loop in 3.1(a).  $\{L_1, L_2, S_0, \dots, S_9\}$  are the set of LNG nodes.  $\{(L_1, L_2), (L_2, S_5)\}$  are some examples of LNG edges.

There are two kinds of nodes in the LNG:

- **LNG Loop Node:** ( $V_{LNG_{Loop}}$ ) Every loop in the function is represented as an LNG loop node. Each loop node further maintains important loop characteristics such as loop bounds and induction variable characteristics for structured loops.

In the example 3.1(d),  $V_{LNG_{Loop}} = \{L_1, L_2\}$ .

- **LNG Instruction Node:** ( $V_{LNG_{Inst}}$ ) Every instruction is represented as an LNG instruction node. In the example 3.1(d),  $V_{LNG_{Inst}} = \{S_0, S_1, \dots, S_9\}$ . There is a one-to-one correspondence to the instructions represented in the PDGs<sup>2</sup>.

Similar to the PDG, the LNG maintains five types of edges, listed below.

- **LNG Loop Control Edge:** ( $E_{LNG_{LCE}}$ ) This type of edge denotes the loop-nest relationship between nodes. There exists an LNG loop control edge from node  $u$  to node  $v$  if  $u$  is the immediate parent loop of  $v$ .  $\{(L_1, L_2), (L_2, S_5)\}$  are some examples of the LNG control edges. An LNG control edge from one loop

---

<sup>2</sup>During implementation, the LNG does not replicate all the information about the instruction in the PDG; instead it contains a structure with a pointer to the PDG instruction. Having the structure allows the LNG to add information such as extra edges to the instructions.

node  $u$  to another  $v$  defines the loop nesting information that the  $v$  is a child loop of  $u$ . An edge from an LNG loop node to an LNG instruction node defines the association of the instruction with its immediate parent loop. Determining the LNG loop control edges involves well-known dominator analysis used to recognize loops in a program.

There exists an LNG control edge  $(u, v) \in E_{LNG_{LCE}}$  iff

- a)  $u \in V_{LNG_{Loop}}$  and  $v \in \cup\{V_{LNG_{Loop}}, V_{LNG_{Inst}}\}$  and
- b)  $u$  is the *immediate dominator* of  $v$ .

- **LNG Affine Memory Dependence Edge:** ( $E_{LNG_{AMDE}}$ ) Each loop-carried memory dependence between two affine memory access instructions is represented in the LNG with an LNG affine memory dependence edge. These edges are further annotated with **distance vectors**. Non-affine loop-carried dependencies are not included in this set. In the example 3.1(d),  $(S_7, S_5)$  is an LNG affine loop memory edge. The edge is annotated with the distance vector  $(1, 0)$  associated with the memory accesses.

In addition to the above two edge types, the LNG maintains three special edges associated with every structured **for** loop, which we together refer to as the LNG Loop Structure Edges ( $E_{LNG_{LSE}}$ ). We follow the usual definition of **for** loops: a **for** loop is a loop whose exit condition is a relational operator comparing an induction variable with a loop-invariant quantity. The three defining characteristics of a **for** loop are the incoming definition, update, and exit condition of the loop's induction variable (IV). These are represented in the LNG by the following three types of

edges:

- **LNG Loop IV Definition Edge:** ( $E_{LNG_{IV-Def}}$ ) A special kind of LNG loop structure edge from a **for** loop to its induction variable definition before the loop. The set of such induction variable definitions is called  $V_{LNG_{IV-Def}}$ . In the example 3.1(d),  $E_{LNT_{IV-Def}} = \{(L_2, S_2), (L_1, S_0)\}$  and  $V_{LNT_{IV-Def}} = \{S_0, S_2\}$ .
- **LNG Loop IV Update Edge:** ( $E_{LNG_{IV-Up}}$ ) A special kind of LNG loop structure edge from a **for** loop to the instruction in that loop that updates its induction variable. By the definition of induction variable, this updating instruction is unique. The set of such induction variable updates is called  $V_{LNG_{IV-Up}}$ . In the example 3.1(d),  $E_{LNT_{IV-Up}} = \{(L_2, S_8), (L_1, S_9)\}$  and  $V_{LNT_{IV-Up}} = \{S_8, S_9\}$ .
- **LNG Loop IV Exit Condition Edge:** ( $E_{LNG_{IV-ECE}}$ ) A special kind of LNG loop structure edge from a **for** loop to the instruction in the loop that calculates its exit condition. The set of such induction variable exit condition instructions is called  $V_{LNG_{IV-ECE}}$ . In the example 3.1(d),  $E_{LNT_{IV-ECE}} = \{(L_2, S_3), (L_1, S_1)\}$  and  $V_{LNT_{IV-ECE}} = \{S_1, S_3\}$ .

For convenience sake, we define

$$V_{LNT_{LSE}} = V_{LNT_{IV-Def}} \cup V_{LNT_{IV-Up}} \cup V_{LNT_{IV-ECE}} \text{ and}$$

$$E_{LNT_{LSE}} = E_{LNT_{IV-Def}} \cup E_{LNT_{IV-Up}} \cup E_{LNT_{IV-ECE}}.$$



## Chapter 4

### Representation of Transformations

To illustrate how the LNG and the PDG are updated for transformations, we consider four transformations as examples below.

#### 4.1 Loop Interchange

Loop interchange is an important affine-based loop transformation. Loop interchange is desirable for a number of reasons such as improving cache performance or increasing granularity of parallelism. Loop interchange can also act as an enabler transform – enabling other transforms such as fusion to become applicable. Traditional analysis for the legality of loop interchange relies on analyzing distance vectors of the loop-nest.

The effects of loop interchange are captured in the two representations in the following manner:

- Loop interchange is a reordering transformation. It only affects the order in which the loop is executed, without changing the code inside the loop. As a result, the most important effect of interchange is on the induction variable behavior of the two loops. This is reflected in the LNG by interchanging the two loop nodes and the associated LNG loop structure edges.

For a given LNG  $G_{LNG}$ , a new  $G'_{LNG}$  is derived for loop interchange of *Loop1* and

*Loop2* by interchanging the position of  $V_{LNG_{Loop1}}$  with  $V_{LNG_{Loop2}}$  along with their associated loop structure edges in the LNG.

In the example, this would involve interchanging L1 with L2, along with their associated loop structure edges.

To reflect the corresponding changes in the PDG, we use the special loop structure nodes that are maintained in the LNG ( $V_{LNG_{LSE}}$ ). Thus this would involve interchanging the control edges associated with the loop structure nodes of the two loops, which would mean:

Swap the control-dependence edges associated with  $V_{LNG1_{IV-Def}}$  and  $V_{LNG2_{IV-Def}}$

Similarly, swap the control-dependence edges associated with  $V_{LNG1_{IV-Up}}$  and  $V_{LNG2_{IV-Up}}$

Similarly, swap the control-dependence edges associated with  $V_{LNG1_{IV-ECE}}$  and  $V_{LNG2_{IV-ECE}}$

- Because of the change in the order of execution, loop interchange affects the data dependencies in both the structures. It changes the order in which memory is accessed, thus changing the dependence patterns (distance vectors) associated with the affine memory dependencies ( $E_{LNG_{AMDE}}$ ). This is reflected in the LNG by performing an interchange on the columns of the distance vectors associated with the loop-nest as described by unimodular theory [2]. In the example 3.1(d), the two columns of the distance vector (1, 0) would be interchanged to yield (0, 1).

$\forall$  edge  $e = (u, v) \in E_{LNG_{AMDE}}$

interchange\_distance\_vectors( $e$ );

Since each LNG affine memory dependence is a copy of the unannotated dependence in the PDG, when the former changes the latter must be changed as well.

When a distance vector component is changed to zero by interchange, the dependence can be removed from the PDG. Thus the dependence edge  $(S_7, S_5)$  is removed after the interchange.

If  $d1, d2$  were the depths of loops interchanged,

for( $d=d1, d2$ )

$\forall$  edge  $e = (u, v) \in E_{LNG_{AMDE}}$

if( $distance\_vector(e)[d]$  changes to 0)

remove\_edge( $pdg\_edge(u, v)$ )

else if ( $distance\_vector(e)[d]$  changes to non-zero)

add\_loop\_carried\_edge( $pdg\_edge(u, v)$ )

where  $pdg\_edge(u, v)$  is the edge from node  $u$  to node  $v$  in the PDG associated with the loop at depth  $d$  in the loop-nest.

Loop interchange does not affect the register dependencies in the PDG because the intra-loop dependencies are not affected by interchanging the loops.

The example 4.1 shows the effect of interchanging the two loops in the example 3.1(b). The highlighted nodes in both the LNG and the PDG show the swapped nodes and the highlighted edge shows the change in the loop-carried memory dependence due to the change in distance vector.

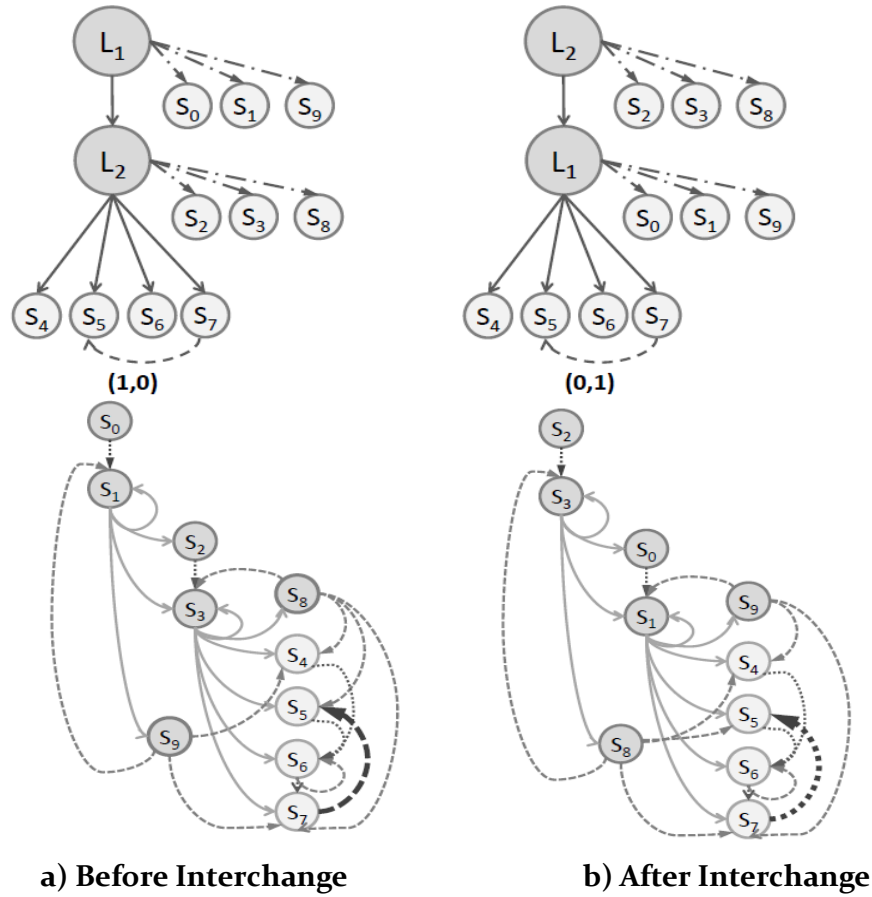


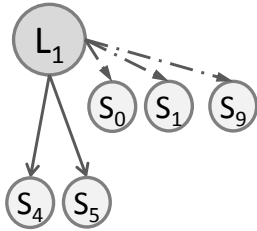
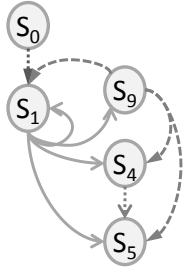
Figure 4.1: Example for Interchange

## 4.2 Loop Fusion

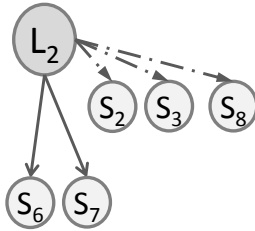
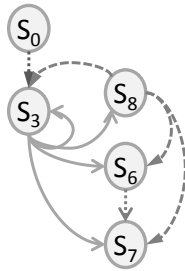
Loop Fusion is another well-understood and often-used loop transformation. Loop fusion affects the runtime in two ways. First, fusing two loops brings down the number of loops thus reducing the synchronization nodes during parallelization. Fusion also has a number of cache benefits, where it can promote cache and data reuse, thus improving the runtime. Loop fusion is advantageous and desirable for both affine and non-affine methods.

Loop Fusion affects both the LNG and the PDG in the loop control structure.

```
for(i=0; i < N; ++i)
  C[i] = A[i];
```



```
for(i=0; i < N; ++i)
  D[i] = B[i];
```



```
for(i=0; i < N; ++i) {
  C[i] = A[i];
  D[i] = B[i];
}
```

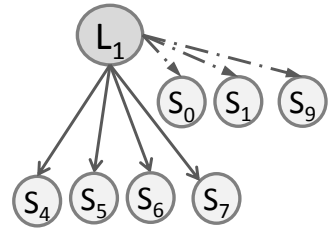
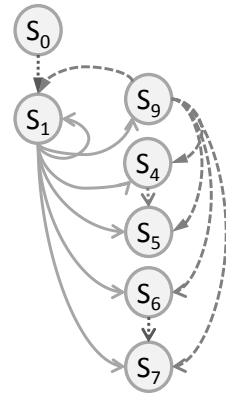


Figure 4.2: Example for Fusion

Like for interchange, changes in the LNG guide the changes to the PDG. Fusing two loops can be visualized as *fusing* the two corresponding LNG loop nodes into one LNG loop node such that *all the dependencies associated with the two loop nodes will now be associated with the fused node*.

Let  $L_1$  and  $L_2$  represent the two LNG loop nodes and  $L_{fused}$  the fused loop node. For fusion to be legal, the loop structures must be the same for  $L_1$  and  $L_2$ , so we can inherit the loop structure for  $L_{fused}$  from either (say  $L_1$ ). Then we inherit the LNG + PDG nodes and dependences for  $L_{fused}$  by unioning those for  $L_1$  and  $L_2$ . Thus, if  $L_{fused} = L_1$ , we inherit the edges from  $L_2$  by:

for( $e = \text{outedges}(L_2)$ )

if( $e \notin E_{LNG2LSE}$ )

change the source of edge  $e$  to  $L_1$

Representing fusion of the two loops in the PDG would involve fusing the loop structures of the corresponding loop PDGs into one fused PDG. Control dependencies in the PDG that define the loop structure are associated with exit condition node ( $V_{LNGIV-ECE}$ ). Instead of replicating the PDG for a fused loop, we use one of the candidate PDGs as our base (say  $L_1$ ) and add the control dependencies for the loop  $L_2$  to the PDG. Thus, fusion of the PDGs involves associating the control dependencies that define the loop structure in the loop  $L_2$  with the  $V_{LNG1IV-ECE}$  of  $L_1$ .

```

for( $e = \text{outedges}(V_{LNG2_{IV-ECE}})$ )
  if( $e \in E_{PDG_{CD}}$ )
    change the source of edge  $e$  to  $V_{LNG1_{IV-ECE}}$ 

```

Whereas the data dependencies within the loops are not affected, new data dependencies can result from the fusion of loops due to pre-fusion inter-loop dependencies converting to intra-loop dependencies after fusion. Since the analysis for fusion involves analyzing such inter-loop dependencies, we use this analysis phase to save the set of possible resultant dependence edges. Once the fusion is determined to be legal, we use the saved information to update the dependencies. Thus changes to data dependencies due to fusion follow directly from the analysis.

The example 4.2 shows an example of the fusion of two loops and the resultant PDG + LNG structure.

### 4.3 Reduction

Reduction is a well-known loop transformation useful in the parallelization of both affine and non-affine programs. Reduction is possible on any statement of the form  $v = v \oplus expr$  in a loop, where  $\oplus$  is a commutative and associative operation such as sum, product, min and max; and  $v$  is loop-invariant in that dimension of the loop. This creates a loop-carried dependence in the loop, inhibiting parallelism. This reduction dependence can be broken by creating private copies of the variable  $v$  for each parallel thread and finally accumulating all the copies into the original after the execution of the entire loop.

Reduction appears in the PDG as a loop-carried data dependence edge. The effect of performing a reduction operation is equivalent to removing the loop-carried dependence  $E_{PDG_{LCDD}}$  associated with the operation  $v = v \oplus expr$ . Reduction does not affect the LNG because we do not represent any register dependencies in the LNG.

Reduction is a constant-time operation involving the removal of the edge corresponding to the reduction operation.

#### 4.4 DSWP

Decoupled software pipelining [12] is a cyclic multi-threading technique which has been used to extract parallelism from general-purpose non-affine code. DSWP uses the PDG to analyze the dependencies and splits the loop into several pipeline stages. It has been shown that the DSWP can also be scaled to higher number of threads when coupled with other parallelization techniques such as DOALL in Parallel Stage-DSWP [13]. This makes the DSWP a very useful transformation to work with affine transformations.

In contrast to the two affine transformations mentioned previously, the PDG guides the changes in the LNG for DSWP. Changes to the PDG due to DSWP techniques follow directly from the analysis [12]. Since DSWP partitions the body of the loop into threads that execute on different pipeline stages, its only affect on the LNG is with respect to the association of instructions with the threads. We represent this information by splitting the LNG loop node in question into several



```

for (i=0; i<M; i++) {
  y=0;
  computation();
  for(j=0; j<N; j++) {
    y = y + arr[j];
  }
}

```

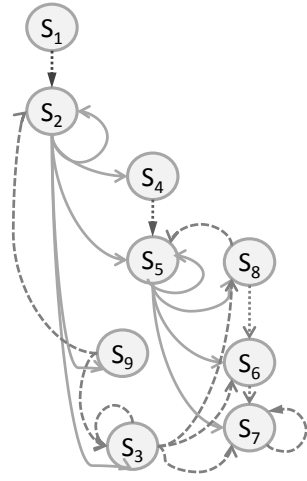
**a) Original Code in C**

```

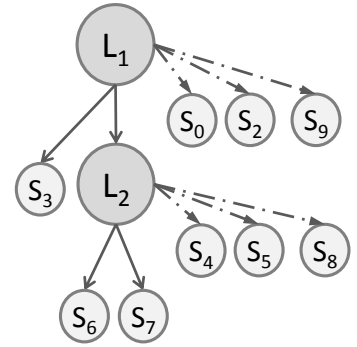
S0: i=0;
S1: y=0;
S2: while(i<M) {
S3:   computation();
S4:   j=0;
S5:   while(j<N) {
S6:     x=arr[j];
S7:     y = y+x;
S8:     j++;
  }
S9:   i++;
}

```

**b) C equivalent of IR**



**c) PDG**



**d) LNG**

Figure 4.3: Example for DSWP

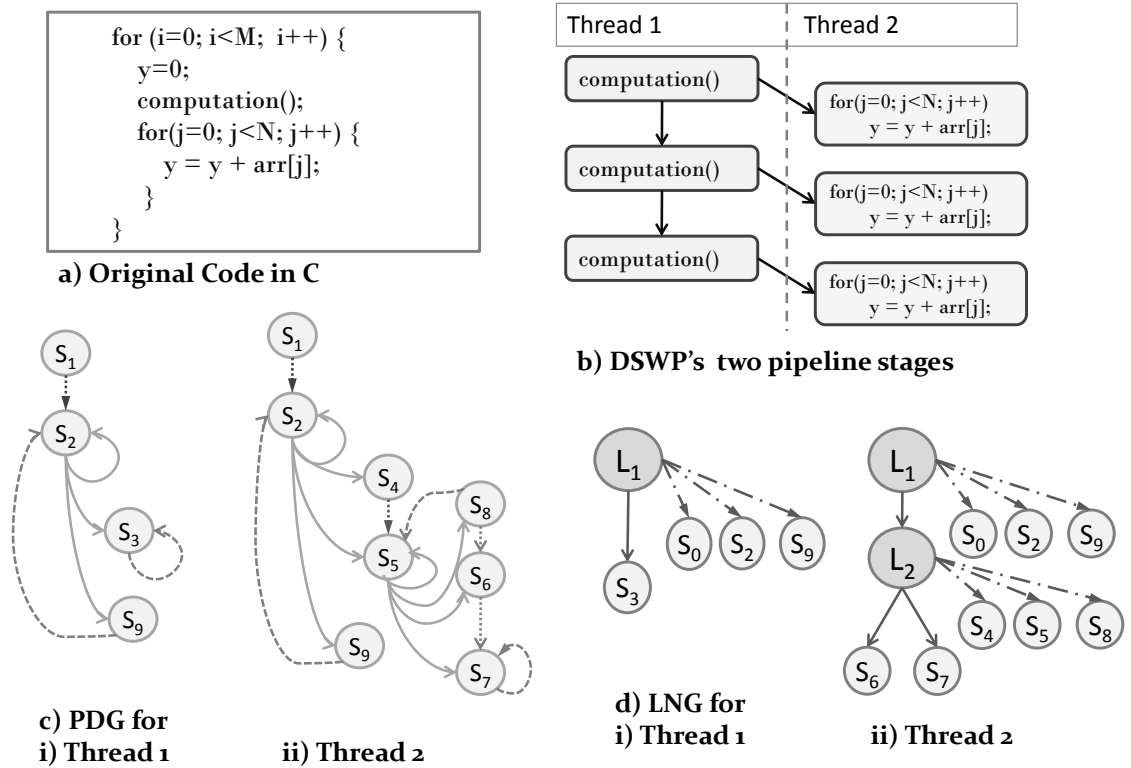


Figure 4.4: PDG and LNG for DSWP

loop nodes, and associating its contained instructions correctly with the new split loop nodes as dictated by the analysis. Updating the LNG involves producing a new LNG for each of the threads and distributing the nodes and edges between the different threaded versions of the LNG. Each of the threads is associated with a new PDG and a new LNG and represents split portions of the original PDG and LNG. The threads correspond to different pipeline stages of execution.

The figure 4.3 shows an example program with the associated PDG and LNG. The figure 4.4(b) shows the result of applying DSWP on the code in 4.3(a). 4.4(c) shows the resultant PDGs for the two threads associated with the two pipeline

stages. 4.4(d) shows the resultant LNGs for the two threads associated with the two pipeline stages.

## Chapter 5

### Results

In order to show the practicality of our representation, we use a very basic parallelizing compiler framework which consists of the following set of transformations – loop interchange, loop fusion, reduction and DSWP. The experimental compiler incrementally applies different transformations and derives a set of candidate orders by using an exhaustive search with pruning. These orders are then evaluated using a simple SRE implementation which uses a limited estimation of cache and program characteristics. At each transformation, only the LNG + PDG are updated, not the IR. Only when one of the orders is chosen, do we apply the set of transformations to the IR and produce parallel code.

We use benchmarks from the *Polybench* benchmark suite [14] – these benchmarks represent heavily used kernels in scientific and multi-media workloads and are suited for affine analysis. The table in figure 5.1 shows results of parallelism and cache-optimization on these benchmarks. As can be seen, several of the benchmarks give substantial improvement even for the single thread case due improvement in cache locality from using loop interchange and fusion. Of course the main point of our results is not the magnitude of the improvement since we are not proposing any particular search method. Rather the main point is that our results demonstrate that our LNG + PDG representation is sound, and allows mixed affine and non-

Benchmark		1	2	4	8	16	24
2mm	Baseline	1.00	2.07	4.14	6.83	12.31	12.59
	Optimized	3.90	7.78	15.17	23.89	34.06	46.19
3mm	Baseline	1.00	1.99	3.94	6.46	11.77	16.25
	Optimized	3.24	6.44	12.48	23.93	36.81	49.52
adi	Baseline	1.00	1.77	2.14	1.49	1.40	1.18
	Optimized	1.08	1.64	3.11	3.29	6.57	7.70
atax	Baseline	1.00	1.74	2.33	1.61	1.61	1.54
	Optimized	0.99	1.26	1.80	1.60	1.91	1.96
doitgen	Baseline	1.00	2.03	4.07	6.27	9.52	9.01
	Optimized	1.00	2.03	4.07	6.27	9.52	9.01
gemm	Baseline	1.00	1.90	3.98	6.86	11.06	12.10
	Optimized	4.29	8.55	16.70	30.83	24.11	29.22
gesummv	Baseline	1.00	1.73	3.01	2.40	2.49	1.94
	Optimized	1.00	1.71	3.00	2.40	2.49	1.95
jacobi-2d-imper	Baseline	1.00	3.44	7.62	5.80	14.19	11.63
	Optimized	1.00	3.37	7.64	5.65	13.04	13.45
<b>Average</b>	Baseline	1.00	1.48	2.38	2.61	4.04	4.33
	Optimized	1.57	2.63	4.61	6.79	8.85	10.69

baseline=without transformations;

optimized=with transformations

Table 5.1: Speedup on x86 24-core machine for *Polybench* benchmark

<b>Transformations (#Thr)</b>	<b>#Thr</b>	<b>Speedup</b>
<b>Base</b>	<b>1</b>	<b>1</b>
<b>DSWP</b>	<b>2</b>	<b>1.1</b>
<b>Interchange</b>	<b>1</b>	<b>2.8</b>
<b>DSWP + Interchange</b>	<b>2</b>	<b>3.4</b>
<b>DSWP + Interchange + Reduction + DOALL</b>	<b>3</b>	<b>5.8</b>

Table 5.2: Speedup for Affine and Non-affine Transformations

affine sequences to be applied correctly without going to IR. The results demonstrate that such a compiler has been correctly built using our representation, and can find runtime-improving transformations.

To show the possibility of combining affine and non-affine transformations, consider the example program in figure 5.1(a) which benefits from both DSWP and loop interchange. The results in table 5.2 show that the speedup of the program is better when using a combination of affine and non-affine transformations – interchange and DSWP (5.8X) as compared to using either interchange only (2.8X) or non-affine only (1.1X). The DSWP partitions the loop such that the inner loop is on one thread and the “*computation()*” on the other thus reducing the runtime as shown in 5.1(c). Loop Interchange, which is an affine loop transformation, on the inner loop modifies the cache access pattern of one of the pipeline stages to improve

```

for (x=0; x<Tot; ++x) {
  y=0;
  computation();
  for(i=0; i<M; i++) {
    for (j=0; j<N; j++) {
      y = y + arr[j][i];
    }
  }
}

```

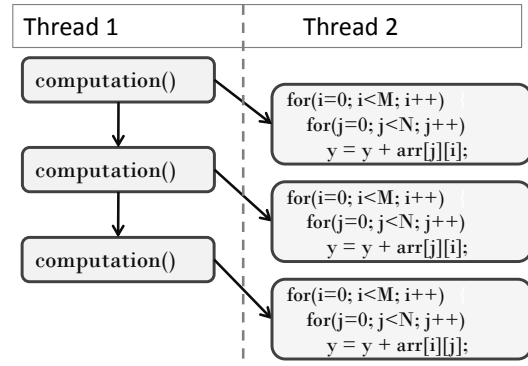
a) Example in C

```

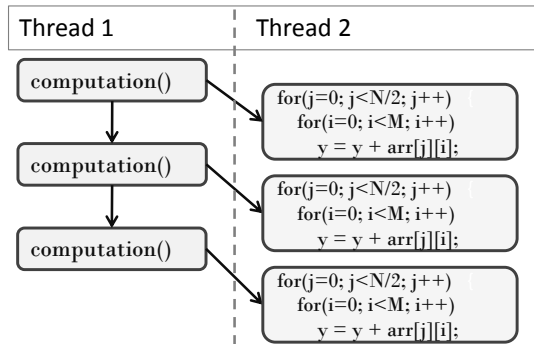
for (x=0; x<Tot; ++x) {
  y=0;
  computation();
  for(j=0; j<N; j++) {
    for (i=0; i<M; i++) {
      y = y + arr[j][i];
    }
  }
}

```

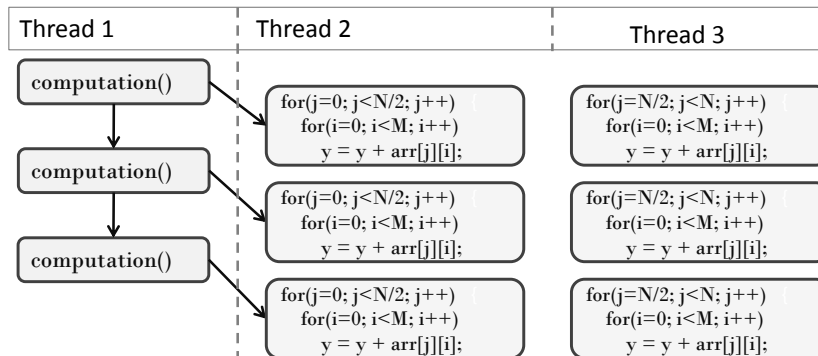
b) With Interchange only



c) With DSWP only



d) DSWP + Interchange



e) DSWP on 2 threads + Interchange with DOALL on 2 threads

Figure 5.1: Example for Affine and Non-affine Transformations

cache reuse, thus reduce runtime as shown in 5.1(b). Thus in combination, the transformations – DSWP on 2 threads and interchange with DOALL on 2 threads – yield a better result.



## Chapter 5

### Conclusion

We have combined two of the existing representations – Program Dependence Graph (PDG) and distance vectors – with a new loop-nest representation called the Loop Nest Graph (LNG) to provide an integrated program representation which is capable of working with both affine and non-affine methods. Thus combining two of the well-understood and regularly-used representations allows us to use the existing research associated with each of these representations directly without having to reinvest in new research. We have shown how to represent some of the transformations in this integrated framework.

Future work involves building a compiler framework using this representation. We have shown its use in a basic compiler which evaluates various orders of transformations. We would like to extend it by adding more affine and non-affine transformations and parallelization methods. Future work also includes a good static runtime estimator which is capable of using the various system and program characteristics to estimate the performance. With all the three pieces, we hope to build a general-purpose parallelizing compiler capable of handling both affine and non-affine programs. We believe, with the integrated capability of both affine and non-affine transformations, the parallelizing compiler will be able to handle any general-purpose code.

## Bibliography

- [1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.
- [2] U Banerjee. Unimodular transformations of double loops. In *In proceedings of the third Workshop on Languages and Compilers for Parallel Computing*, August 1990.
- [3] Utpal K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.
- [4] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, 2004.
- [5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, 2008.
- [6] Hideki Saito et al. The design of the promis compiler. In *CC '99: Proceedings of the 8th International Conference on Compiler Construction*, pages 214–228, 1999.
- [7] Mary W. Hall et al. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996.
- [8] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, pages 79–103, London, UK, 1996. Springer-Verlag.
- [9] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.
- [10] M Girkar and C Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. Parallel Distrib. Syst.*, 3:166–178, March 1992.
- [11] Milind Girkar and Constantine D. Polychronopoulos. The hierarchical task graph as a universal intermediate representation. *Int. J. Parallel Program.*, 22(5):519–551, 1994.

- [12] J Huang, A Raman, T Jablin, Y Zhang, T-H Hung, and D August. Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010.
- [13] H Kim, A Raman, F Liu, J Lee, and D August. Scalable speculative parallelization on commodity clusters. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '43*, 2010.
- [14] Polybench: The Polyhedral Benchmark Suite. *INRIA Paris - Rocquencourt*, 2010.
- [15] C D Polychronopoulos, M B Gikar, M R Haghghat, C L Lee, B P Leung, and D Schouten. The structure of parafrase-2: an advanced parallelizing compiler for c and fortran. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, 1990.
- [16] V Sarkar and B Simons. Parallel program graphs and their classification. In *Proceedings of the 6th Intl Workshop on Languages and Compilers for Parallel Computing*, 1994.
- [17] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, CGO '03*, pages 204–215, Washington, DC, USA, 2003. IEEE Computer Society.
- [18] R P Wilson, R S French, C S Wilson, S P Amarasinghe, J M Anderson, W. K. Tjiang, S Liao, C Tseng, M W Hall, M S Lam, and J L Hennessy. Suif: an infrastructure for research on parallelizing and optimizing compilers. 1994.
- [19] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. *Int. J. Parallel Program.*, 26:479–503, August 1998.
- [20] Michael Edward Wolf. *Improving locality and parallelism in nested loops*. PhD thesis, Stanford, CA, USA, 1992.