

©Copyright by

William Joseph Blume

1995

SYMBOLIC ANALYSIS TECHNIQUES FOR EFFECTIVE
AUTOMATIC PARALLELIZATION

BY

WILLIAM JOSEPH BLUME

M.S., University of Illinois, 1992

B.S., University of Illinois, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

SYMBOLIC ANALYSIS TECHNIQUES FOR EFFECTIVE AUTOMATIC PARALLELIZATION

William Joseph Blume, PhD
Department of Computer Science
University of Illinois at Urbana-Champaign, 1995
Rudolf Eigenmann, Advisor

To effectively translate real programs written in standard, sequential languages into parallel computer programs, parallelizing compilers need advanced techniques such as powerful dependence tests, array privatization, generalized induction variable substitution, and reduction parallelization. All of these techniques need or can benefit from symbolic analysis.

To determine what kinds of symbolic analysis techniques can significantly improve the effectiveness of parallelizing Fortran compilers, we compared the automatically and manually parallelized versions of the Perfect Benchmarks. The techniques identified include: data dependence tests for nonlinear expressions, constraint propagation, interprocedural constant propagation, array summary information, and run time tests. We have developed algorithms for two of these identified symbolic analysis techniques: nonlinear data dependence analysis and constraint propagation.

For data dependence analysis of nonlinear expressions, we developed a data dependence test called the Range Test. The Range Test proves independence by determining whether certain symbolic inequalities hold for a logical permutation of the loop nest. We use a technique called Range Propagation to prove these symbolic inequalities.

For constraint propagation, we developed a technique called Range Propagation. Range Propagation computes the range of values that each variable can take at each point of a program. A range is a symbolic lower and upper bound on the values taken by a variable. Range

propagation also includes a facility to compare arbitrary expressions under the constraints imposed by a set of ranges. We have developed both a simple but slow algorithm and a fast and demand-driven but complex algorithm to compute these ranges.

The Range Test and Range Propagation have been fully implemented in Polaris, a parallelizing compiler being developed at the University of Illinois. We have found that these techniques significantly improve the effectiveness of automatic parallelization. We have also found that these techniques are reasonably efficient.

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Rudolf Eigenmann, for his guidance and suggestions. His support and insight proved to be invaluable in the construction of this dissertation. I would also like to thank David Padua, for his many helpful comments and for providing an supportive environment that made my work possible.

I would like to thank my former officemates, especially Lawrence Rauchwerger and Sharad Mehrotra, for the many intelligent and thought-provoking conversations I had with them. I am also grateful to the members of the Polaris group, most notably John Grout, Jay Hoeflinger, Paul Petersen, and Peng Tu, in no particular order.

TABLE OF CONTENTS

CHAPTER	PAGE
1 INTRODUCTION	1
1.1 Parallelizing compilers	1
1.2 Effectiveness of late '80s parallelizing compilers	3
1.3 Improving the effectiveness of parallelizing compilers	5
1.4 Role of symbolic analysis in parallelizing compilers	8
1.5 Foreshadowing	10
1.6 Organization of dissertation	12
2 SYMBOLIC ANALYSIS TECHNIQUES NEEDED FOR EFFECTIVE PARALLELIZATION OF THE PERFECT BENCHMARKS	13
2.1 Motivation for symbolic analysis	13
2.2 Analysis of the Perfect Benchmarks	15
2.3 Symbolic, nonlinear dependence analysis	16
2.4 Constant propagation	19
2.4.1 Interprocedural constant propagation with procedure cloning	19
2.4.2 Guarded constant propagation	22
2.5 Symbolic constraint propagation	24
2.6 Subscript array analysis	28
2.6.1 Array constant propagation	29
2.6.2 Analysis of subscript arrays	29
2.7 Generating runtime tests	31
2.8 Other techniques	33
2.8.1 Compile time interpretation of programs	33
2.8.2 Algorithm recognition	34
2.9 Summary	35
2.10 Conclusions	37
3 THE RANGE TEST	39
3.1 Open issues in data dependence testing	39
3.2 Data dependence	40
3.2.1 Data dependence	41
3.2.2 Direction vectors	41
3.3 The Range Test	42

3.3.1	Disproving dependence between symbolic expressions	43
3.3.2	Permuting loops for dependence testing	48
3.3.3	Algorithm	50
3.3.4	Computing f_j^{\min} and f_j^{\max}	52
3.3.5	Time complexity	54
3.4	Symbolic range propagation	55
3.5	Generalizing the Range Test	56
3.5.1	Multidimensional arrays	57
3.5.2	Negative strides	57
3.5.3	Loop-variant variables	58
3.5.4	Loops that aren't perfectly nested	58
3.6	Examples	58
3.7	Measurements	61
3.7.1	Effectiveness	62
3.7.2	Speed	67
3.8	Related work	69
3.9	Conclusions	71
4	SYMBOLIC RANGE PROPAGATION	73
4.1	Benefits from an expression comparator	73
4.2	Comparing expressions using symbolic ranges	76
4.2.1	Algorithm	76
4.2.2	Replacing a variable with its range	78
4.2.3	Determining a replacement order	85
4.2.4	Time complexity	89
4.2.5	Performance	91
4.3	Propagating ranges	93
4.3.1	Basic operations	93
4.3.2	Algorithm	95
4.3.3	Example	97
4.3.4	Time complexity	98
4.4	Related work	99
4.5	Conclusions	101
5	DEMAND DRIVEN SYMBOLIC RANGE PROPAGATION	103
5.1	Motivation for demand-driven analysis	103
5.2	Notation	104
5.3	Range propagation	105
5.4	Computing control ranges	107
5.4.1	Needed functionality	107
5.4.2	Algorithm	107
5.4.3	Time complexity	109
5.4.4	Design justification	109
5.4.5	Optimizations	111
5.5	Computing data ranges	112
5.5.1	Data-flow graph	112

5.5.2	Algorithm	114
5.5.3	Computing data ranges from a data-flow graph	116
5.5.4	Example	121
5.5.5	Time complexity	124
5.6	Handling union and intersection operations	126
5.6.1	Control ranges	126
5.6.2	Data ranges	127
5.7	Performance	128
5.8	Related work	132
5.9	Conclusions	135
6	CONCLUSIONS AND FUTURE WORK	136
6.1	Conclusions	136
6.2	Future work	137
	BIBLIOGRAPHY	139
	VITA	144

LIST OF TABLES

2.1	Estimated amount of slowdown on Cedar from manually parallelized codes if a specific symbolic analysis technique could not be used.	36
3.1	Trace of Range Test for loop nest FTRVMT/109 shown in Figure 3.6.	59
3.2	Number of parallel loops or eliminated loop-carried dependences detected by the Range Test and the omega test.	63
3.3	Time taken by the Range and Omega tests on real programs. Timings for the rest of dependence testing as well as all the analyses performed before dependence testing are also included.	67
4.1	Rewrite rules for simplifying expressions containing ranges. Variables a , b , and c are symbolic integer-valued expressions. The variable i is an integer.	79
4.2	Rewrite rules for simplifying min expressions. The rewrite rules for max expressions is similar. Variables a , b , and c are assumed to be arbitrary symbolic integer expressions. Variable i is an integer.	81
4.3	Average time and number of replacements made by the expression comparison algorithm in Figure 4.2 for six Perfect Benchmarks codes.	93
4.4	Basic operations used by the range propagation algorithm.	94
4.5	Time taken by the range propagation algorithm for six Perfect Benchmarks codes and two NCSA codes.	99
5.1	Time taken in seconds to compute all data ranges, all control ranges, and merge all data and control ranges on a Sparc 10.	128
5.2	Average and maximum number of control ranges computed when computing a single control range.	130
5.3	Average and maximum number of control ranges, data ranges, and poisoned data ranges computed when computing a single data range.	131

LIST OF FIGURES

1.1	Speedups of the Perfect Benchmarks transformed by the 1988 versions of Kap and Vast for the Alliant FX/80.	4
1.2	Speedups of automatically and manually parallelized versions of the Perfect Benchmarks on Cedar.	5
1.3	Speedups of codes transformed by PFA and Polaris on an 8 processor set on an SGI Power Challenge in flash mode.	11
3.1	Model of loop nest for dependence testing.	40
3.2	Examples of how array accesses can be interleaved in respect to a particular loop (loop with index i for these examples.) All examples assume that $0 \leq i, j < n$. . .	45
3.3	The Range Test algorithm.	52
3.4	Algorithm for disproving carried dependences for loop L_j in respect to loops \mathcal{L} . Loop L_j is assumed to have index i_j . The word <i>monotonically</i> has been abbreviated to “mono”.	53
3.5	Algorithm for calculating the maximum value of function f for fixed values of indices i_j of loops L_j , where $L_j \notin \mathcal{L}$	54
3.6	Simplified version of loop nest FTRVMT/109 from <i>OCEAN</i>	59
3.7	Simplified version of loop nest OLDA/100 from <i>TRFD</i>	60
4.1	A loop nest, extracted from TRFD, that contains non-affine array references. Loop peeling and induction variable substitution were performed to place the loop nest into this form.	75
4.2	The symbolic expression comparison algorithm.	78
4.3	An example of a Range Dependence Graph (RDG).	86
4.4	An example of a cyclic Range Dependence Graph.	88
4.5	Computation of ranges for an small code segment.	98
5.1	The demand-driven range propagation algorithm.	106
5.2	The demand-driven control range propagation algorithm.	108
5.3	Demand-driven algorithm for computing a sparse immediate dominating control-flow edge relationship.	111
5.4	Fields of a node structure.	112
5.5	Algorithm to create and initialize a data-flow graph node.	113
5.6	The demand-driven data range propagation algorithm.	114
5.7	Algorithm to create children for a data-flow graph node.	115
5.8	Algorithm to commit data ranges in the data-flow subgraph rooted at <i>root</i> . . .	115
5.9	Algorithm to compute data ranges from the given graph.	117

5.10	Algorithm to compute data ranges for nodes whose definitions are ϕ -functions.	118
5.11	Example used to show how function <code>get_data_range</code> works.	122
5.12	The data-flow graph created by function <code>get_data_range(i_4)</code> for the example in figure 5.11.	123

Chapter 1

INTRODUCTION

1.1 Parallelizing compilers

In response to the need for faster, more-powerful machines, parallel architectures were introduced by the high performance computing industry. Unfortunately, to fully utilize these machines, users needed to write explicitly parallel programs. This posed several difficulties for the user community. First, they had to rewrite their existing programs (a.k.a dusty decks) to use the parallel machine. Second, most of the resulting explicitly parallel programs were not portable. Third, writing efficient parallel programs often required optimizations that need intimate knowledge of the machine's architecture and the programs' access patterns, (e.g., data distribution, prefetching, or blocking). To address these difficulties, parallelizing compilers were developed to transform sequential programs to parallel ones.

Parallelizing compilers can be broken into two components; a component that identifies parallelism in a program, and a component that exploits this parallelism. The component that identifies parallelism attempts to determine what parts of a program can be run in parallel.

The component that exploits parallelism determines which of these parallel parts should be run in parallel, as well as how to generate efficient code for them. In this dissertation, we will only concentrate upon the first component: the identification of parallelism.

To identify parallelism in programs, parallelizing compilers perform data dependence analysis. Data dependence analysis determines all the pairs of array or scalar accesses in a program that may touch the same memory location. If two array or scalar accesses may touch the same memory location, they are said to be data dependent. A loop can be run fully in parallel if there are no data dependences from one iteration to another iteration, ignoring dependences between read accesses. Because of the importance of data dependence analysis, there has been much research in this field [3, 25, 37, 42, 50].

Because a loop cannot be executed fully in parallel if it has cross-iteration dependences, many techniques have been developed to eliminate such dependences. Two techniques that are implemented in nearly all commercial parallelizing compilers are induction variable substitution, and scalar privatization. Induction variable substitution, which is the reverse of strength reduction [2], replaces induction variables in loops, (e.g., $i = i + 2$) with assignments whose right-hand-sides use only loop-invariant variables and loop indices, (e.g., $i = 2*j$, where j is the index of an enclosing loop). Scalar privatization recognizes scalars that act as temporary variables in a loop iteration, (i.e., they are always written before read in an iteration), and gives each processor a local copy of these scalars.

Introductions to and summaries of the research performed on parallelizing compilers can be found in Banerjee, Eigenmann, Nicolau, and Padua [5], Padua and Wolfe [40], or the textbook by Zima and Chapman [51].

1.2 Effectiveness of late '80s parallelizing compilers

As described in the previous section, much research has been done in developing techniques to identify and exploit parallelism in programs. Many of these techniques have been implemented in research or commercial parallelizing compilers. However, around 1989–1991, there were few studies on the effectiveness of these techniques. Additionally, almost all of these studies used only small, synthetic kernels.

To determine the effectiveness of parallelizing compilers on real programs, we measured the speedups of the Perfect Benchmarks that were parallelized by the 1988 versions of the two commercial parallelizing compilers: Kap and Vast [20, 8, 10]. We also measured the effectiveness of the individual restructuring techniques used by these compilers. By *speedup*, we mean the time taken by the original sequential code divided by the time taken by the parallelized version of the code. The Perfect Benchmarks is a suite of 13 Fortran 77 programs representing applications in a number of areas in engineering and scientific computing [6]. The measurements were made on an Alliant FX/80, which is an 8 vector-processor machine. Since the pipeline of each vector unit is four deep, the theoretical achievable speedup from both vectorization and parallelism is 32. The practical maximum speedup is more around 16 to 20.

The results of this study were very poor. Figure 1.1 shows the speedups of the vector, concurrent, and vector-concurrent versions of the Perfect Benchmarks, as produced by Kap, as well as the vector-concurrent versions produced by Vast. Out of the thirteen codes, only two had a speedup of 8 or more. Seven of the thirteen codes had speedups of 1.5 or less. The other four codes had speedups between 2 and 4.5. The speedups for Kap and Vast were nearly identical. In addition to these results, we found that most of the restructuring techniques applied by these parallelizing compilers had little effect on the programs' performance.

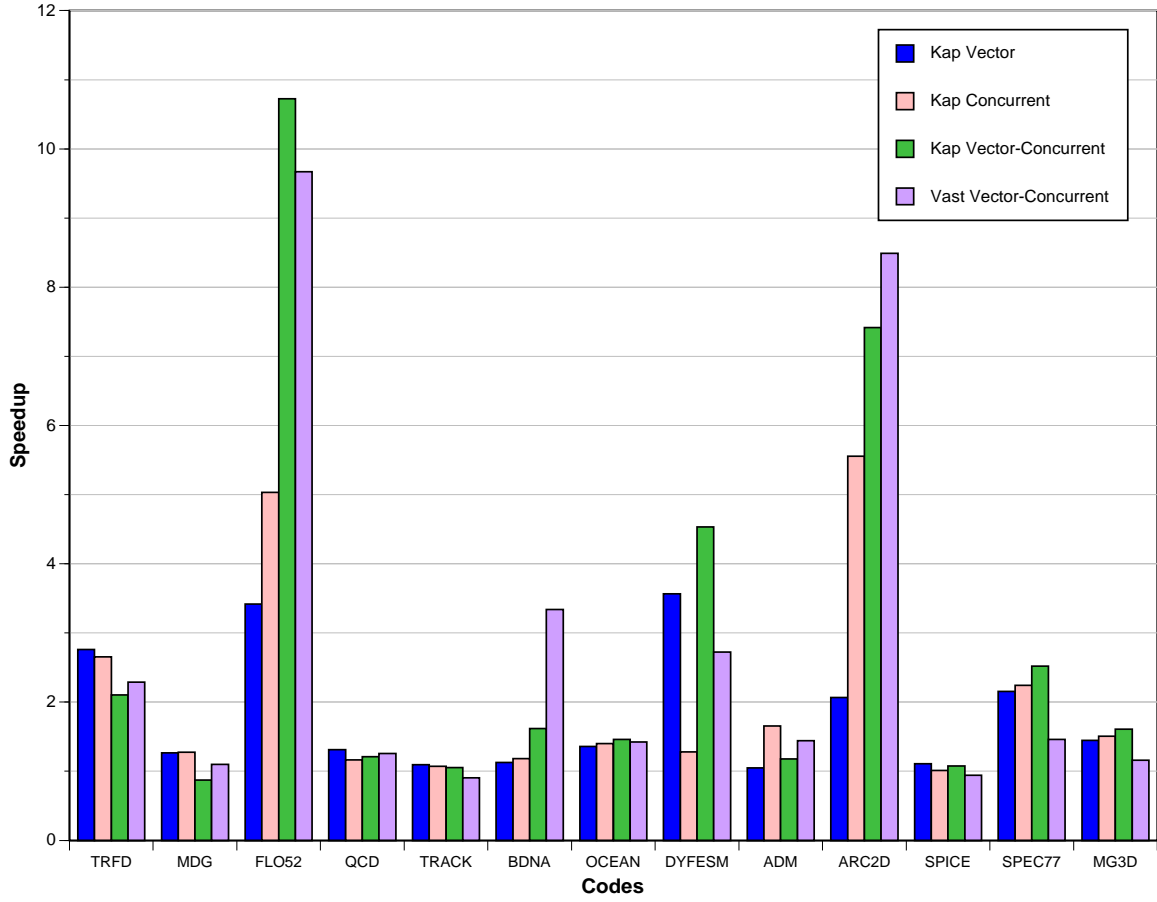


Figure 1.1: Speedups of the Perfect Benchmarks transformed by the 1988 versions of Kap and Vast for the Alliant FX/80.

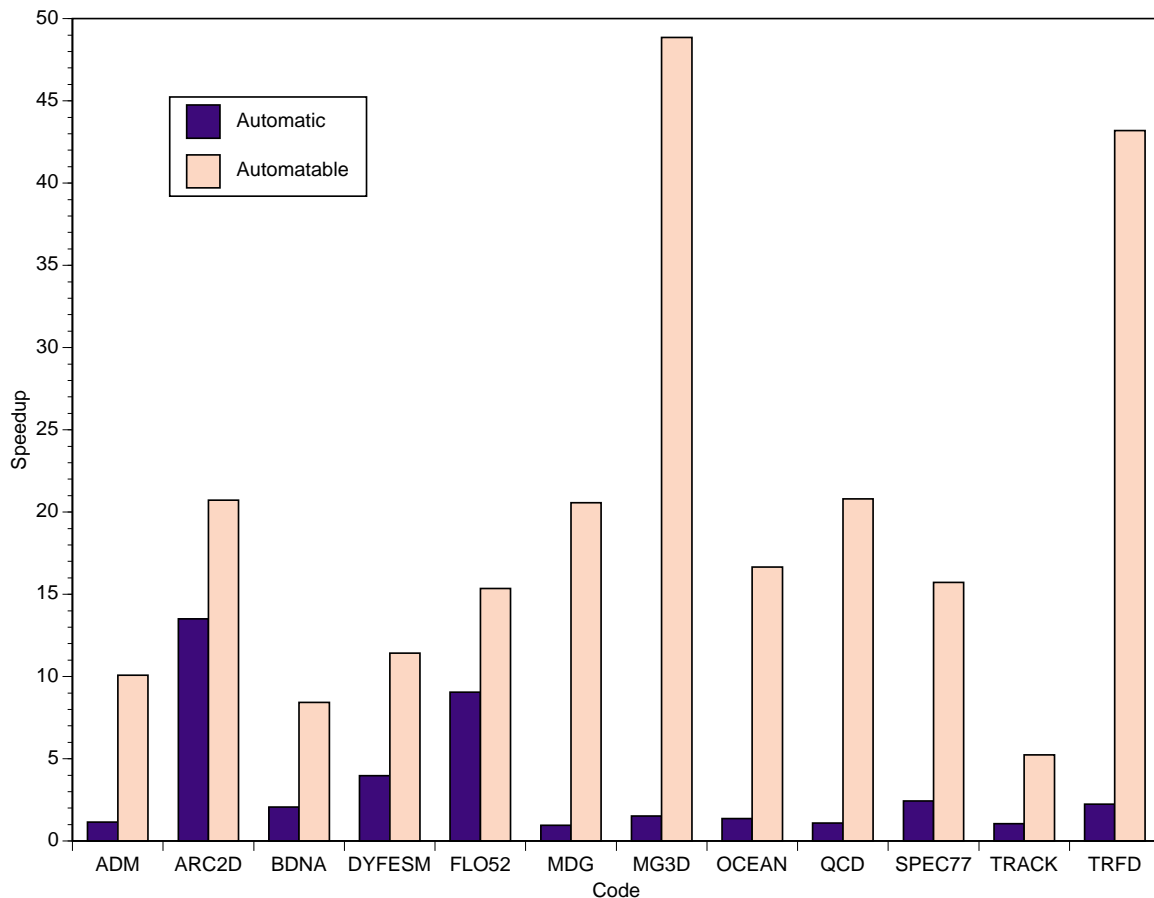


Figure 1.2: Speedups of automatically and manually parallelized versions of the Perfect Benchmarks on Cedar.

1.3 Improving the effectiveness of parallelizing compilers

In response to this poor effectiveness of the then state-of-the-art parallelizing compilers, we asked ourselves: are these results due to the fact that these programs contain little parallelism, or is it because parallelizing compilers are not sufficiently powerful enough to identify and exploit the parallelism in the programs? To answer this question, our research group at the Center of Supercomputing Research and Development decided to manually parallelize the Perfect Benchmarks, using only those techniques that are likely to be implementable in a parallelizing compiler [23, 22, 21].

The results of this manual parallelization effort were very heartening. From the twelve codes in the Perfect Benchmarks that were hand-parallelized, we were able to achieve a mean speedup of ten to twenty on the Cedar multiprocessor, which consists of four clusters of Alliant FX/8's. The speedups of these manually parallelized codes versus the speedups of Kap/Cedar is shown in Figure 1.2. The *Automatic* bars represents the speedups achieved by Kap/Cedar, which is based upon the 1988 version of commercially available parallelizer named Kap. The *Automatable* bars represent the speedups of the hand transformed codes. Remember that only those transformations that were theoretically implementable in a compiler were applied to these codes. These results indicate that it is possible to automatically parallelize real programs effectively. Hence, there exists significant room for improvement for parallelizing compilers.

Although the speedups achieved by this manual parallelization effort were impressive, the main lesson that we learned from this effort was that only a few additional techniques were needed to achieve these speedups for most of the codes [23, 22, 21]. These techniques were:

- Interprocedural analysis
- Nonlinear dependence analysis
- Array privatization
- Generalized induction variable substitution
- Reduction parallelization

Interprocedural analysis is needed since the important¹ parallel loops in many programs contain procedure calls. Additionally, data such as symbolic constants is often needed to be passed

¹We say that a loop or loop nest is *important* if that loop or loop nest takes up a significant fraction of the program's execution time.

across procedure boundaries to provide enough information to disprove data dependences. This data is also needed to determine and compare the sections of arrays that are written or read by a loop, which is needed for techniques such as array privatization.

Nonlinear dependence analysis is required since some programs contain nonlinear array subscripts or loop bounds in important parallel loop nests. Most existing data dependence tests assume that all array subscripts and loop bounds are linear, (i.e., affine). Hence, such tests would fail when they encounter a nonlinear expression.

Array privatization is needed since the iterations of many large loops in real programs use arrays as temporary workspaces. That is, no data is passed between iterations for such arrays. Array privatization simply gives each processor a local copy of these arrays, thus breaking the dependences due to the sharing of memory locations by such arrays [43].

Generalized induction variable substitution is simply induction variable substitution extended to handle cases such as triangular loop nests, multiply nested loop nests with symbolic bounds, or loop nests with coupled induction variables [28].

Finally we have seen that a large number of important loop nests have cross-iteration dependences from reduction statements. Reduction statements are statements that update the value of some scalar or array element using an associative operation, (e.g., $s = s + a(i)$ or $b(i) = b(i) * x$). For such cases, a compiler can extract such statements from a loop and replace them with a parallel version, similar to a parallel-prefix computation [41].

Because of the potentially great impact that these additional transformations would have on the effectiveness on parallelizing compilers, our research group has developed and implemented these transformations in Polaris, a state-of-the-art research parallelizing compiler being devel-

oped at the University of Illinois. A description of Polaris and algorithms for these techniques can be found in [24, 9, 7].

Many of these additional techniques must perform some sort of symbolic analysis to be effective. The goal of this dissertation is to identify and develop symbolic analysis techniques that will improve the effectiveness of these techniques, with the emphasis upon data dependence analysis. Additionally, because these symbolic analyses will be applied upon real programs that are tens-of-thousands of lines long or longer, the algorithms developed for these analyses must be both robust and efficient.

1.4 Role of symbolic analysis in parallelizing compilers

Many restructuring techniques used by parallelizing compilers need some sort of symbolic analysis to be effective. By symbolic analysis, we mean any kind of analysis that can manipulate or propagate values that contain symbolic terms (e.g. program variables). In this section, we will describe how symbolic analysis can aid data dependence testing. The symbolic analysis needed by other transformations used by parallelizing compilers is often very similar.

Typically, data dependence tests for parallelizing compilers demand that loop bounds and array subscripts are a linear (affine) function of loop index variables. That is, they are of the form

$$c_0 + \sum_{j=1}^n c_j i_j$$

where c_j are integer constants and i_j are loop index variables. Some tests are more lenient, allowing some i_j 's to also be loop invariant variables [42]. Unfortunately, array subscripts and loop bounds in real programs are not always in this format. Instead, the coefficients may not be integer constants, or the subscripts or array bounds may contain loop variant

variables or subscript array references. We will call such expressions *nonlinear*. In response to such unmanageable expressions, a variety of symbolic analyses and transformations have been developed. These analyses and transformations handle the offending expressions in one of two ways:

1. Eliminate the offending variable or subexpression from the subscript expression.
2. Extend data dependence analysis to cope with such expressions.

The most straightforward, and usually the easiest, way to handle nonlinear subscript expressions or expressions containing loop variant variables is to eliminate these nonlinearities or loop variants. Constant propagation [46] and induction variable substitution [28, 48] are the most common methods used to transform array subscripts into testable linear expressions. Symbolic simplification of expressions [14, 29, 32] is also important for canceling common terms and eliminating complex expressions.

Although transforming a subscript expression or loop bound into a testable linear form is preferred, it is not always possible. We have seen examples in several of the codes in the Perfect Benchmarks that had array subscripts which could not be transformed into a linear form or which contained loop-variant expressions or subscript array references that could not be eliminated. Additionally, some common compiler transformations introduce nonlinearities to subscript expressions. The two most common offenders are linearization of arrays,² which is often needed for inlining or interprocedural analysis, and generalized induction variable substi-

²By the linearization of an array, we mean the transformation of two or more dimensions of an array into a single dimension. For example, if the array `a`, which was originally declared to be `a(n,m)`, was linearized, its declaration will be changed to `a(n*m)`, and a reference `a(i,j)` will be changed to `a(i + n*j)`. Do not confuse the term “array linearization” with the terms “linear” or “nonlinear”. The similarities between these terms are purely accidental.

tution. In these cases, dependence analysis techniques must be modified to cope with nonlinear expressions and loop variant variables.

The symbolic analysis techniques developed in this dissertation handle nonlinear expressions and loop variant variables by extending data dependence analysis. Because of this, we had to develop a data dependence test to handle such nonlinear expressions, (called the Range Test, which will be described in Chapter 3), and develop techniques to manipulate and reason about nonlinear expressions, (called Range Propagation, which will be described in Chapters 4 and 5).

1.5 Foreshadowing

To wet the reader's appetite for the analysis techniques developed in this paper, Figure 1.3 displays the speedups achieved by Polaris versus the speedups achieved by PFA, which is a commercial parallelizing compiler that is based on Kap and is for the SGI Power Challenge. These speedups were collected on a 8 processor set on a SGI Power Challenge. The codes listed are six codes from the Perfect Benchmarks plus three application codes used by the National Center of Supercomputing Applications. As shown, the programs parallelized by Polaris were able to match or exceed the performance of the codes parallelized by PFA, with one exception.³ In fact, the speedups of two-thirds of the Polaris transformed codes are at least double the speedups of the PFA transformed codes.

Now, the increase in the performance of the Polaris transformed codes is not solely due to the analysis techniques developed in this dissertation. Instead, they are due to a combination of the analysis techniques developed in this dissertation and the additional techniques described

³Polaris did not do as well as PFA on ARC2D only because PFA interchanged some loops to increase data locality while Polaris did not. Polaris did not perform these loop interchanges because the component responsible for exploiting parallelism is under development. No speedup was shown for PFA on CLOUD3D because PFA crashed while compiling this code.

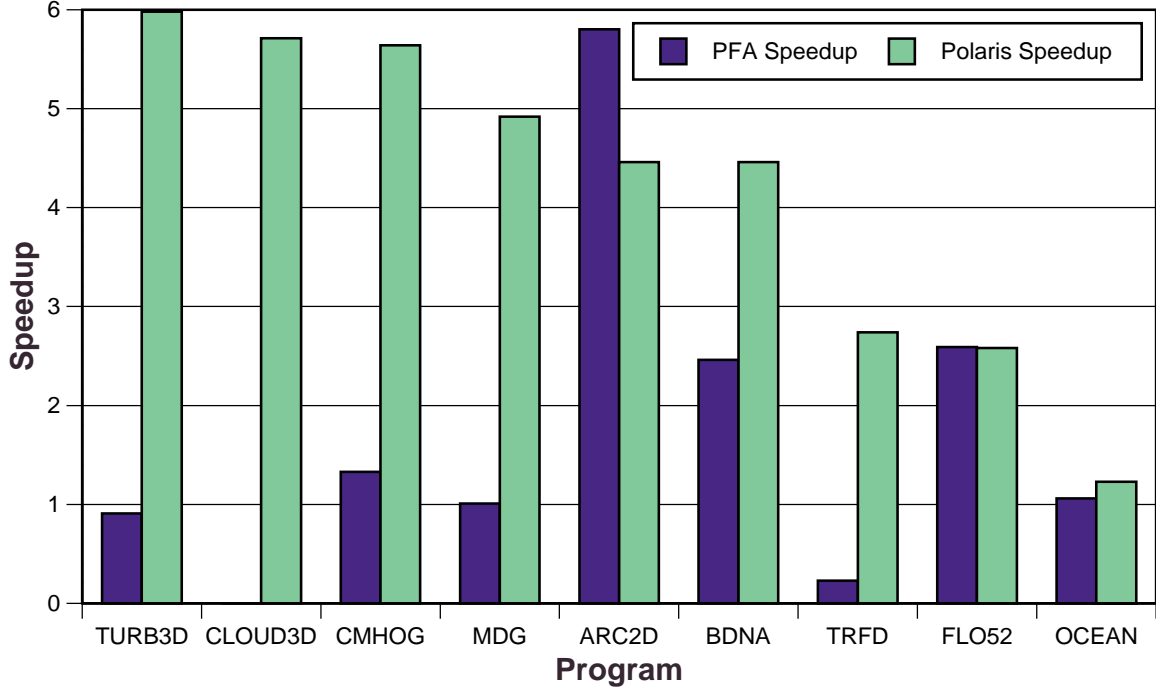


Figure 1.3: Speedups of codes transformed by PFA and Polaris on an 8 processor set on an SGI Power Challenge in flash mode.

in the previous section. However, for some of the codes shown in Figure 1.3, our techniques had a profound impact on the effectiveness of Polaris. More specifically, if the analysis techniques developed in this dissertation were not applied by Polaris, the performance of the Polaris transformed codes would have been the same as the PFA transformed codes for TRFD and OCEAN and the increase in performance by the Polaris transformed codes would have been significantly diminished for CMHOG and MDG.

Additionally, our analysis techniques were reasonably efficient for these nine codes. Polaris took about 2–15 minutes on a single processor of the SGI Power Challenge to compile the codes shown in Figure 1.3. At most a third of these compilation times were spent in the symbolic analysis techniques developed in this dissertation. These codes are about 600–14000 lines long.

Hence, we feel confident to claim that the symbolic analysis techniques developed in this dissertation are efficient and have a significant impact on the effectiveness of parallelizing compilers.

1.6 Organization of dissertation

This dissertation is organized as follows. Chapter 2 will describe what symbolic analysis techniques should improve the effectiveness of parallelizing compilers. These techniques are identified by an in-depth study of the manually parallelized versions of the Perfect Benchmarks. The rest of this dissertation will then describe the implementations of two of three most important symbolic analysis techniques identified by this study. We call these two techniques the *Range Test* and *Range Propagation*.⁴ Chapter 3 will describe the implementation of the Range Test, a symbolic data-dependence test designed to handle nonlinear array references and loop bounds. Range Propagation will then be described in Chapter 4. Range Propagation derives inequality relationships, called ranges, from a program’s text, then uses these ranges to prove or disprove inequality relationships between two arbitrary symbolic expressions. A more efficient and demand-driven version of Range Propagation will be described in Chapter 5. Finally, chapter 6 will summarize the results of this dissertation.

⁴The third of the three most important symbolic analysis techniques is interprocedural constant propagation with procedure cloning. Although we have implemented this technique in Polaris, it is a faithful implementation of the algorithm described by Callahan et. al. [13]. Because this technique and its effectiveness has been thoroughly discussed by others, we will not discuss it in this dissertation.

Chapter 2

SYMBOLIC ANALYSIS TECHNIQUES NEEDED FOR EFFECTIVE PARALLELIZATION OF THE PERFECT BENCHMARKS

2.1 Motivation for symbolic analysis

To allow the user community to write programs that run efficiently on a parallel architecture without needing to write explicitly parallel code, parallelizing compilers were developed to transform sequential programs into parallel ones. Unfortunately, an effectiveness study of parallelizing compilers performed by our research group in 1989-1992 on the Perfect Benchmarks

found that parallelizing compilers are not very effective at parallelizing real programs, (i.e., the transformed programs do not get good speedups from the original sequential codes.) [20, 8, 10].

In response to this, our research group manually parallelized the Perfect Benchmarks, using only those transformations that are theoretically implementable in a compiler, to determine how effective a parallelizing compiler can be [23, 22, 21]. This manual parallelization effort shown that it was possible for parallelizing compilers to transform programs into a parallel form that gets good speedups from the original sequential code. Additionally, it found that only a few restructuring techniques beyond those already implemented in parallelizing compilers were needed to achieve these good speedups for many of the codes.

Many of the additional restructuring techniques identified by this manual parallelization effort need to perform some sort of symbolic analysis to be effective. The manual parallelization effort have also seen several important cases where data dependence tests need to be done symbolically to prove independence. In this chapter we will identify which symbolic analysis techniques will improve the effectiveness of these techniques, with the emphasis upon data dependence analysis.

The study described in this chapter was important preliminary work that led to the development of the two symbolic analysis techniques, called the Range Test and Range Propagation, which we have developed in this dissertation. More specifically, this study have shown us that analyses like the Range Test and Range Propagation are needed to effectively parallelize a significant fraction of the Perfect Benchmarks. Additionally, the study provided us with cases that these techniques must handle. Descriptions of these two techniques would be given in later chapters.

2.2 Analysis of the Perfect Benchmarks

To identify symbolic analysis techniques that are useful for aiding parallelizing techniques, we have compared real applications that were restructured by current parallelizing compilers with their manually parallelized counterparts. Only transformations that can be considered to be “automatable”, or theoretically implementable in a compiler, were used on these manually parallelized codes. These transformations typically consisted of one or more of the following: marking a loop as parallel, privatizing one or more arrays, eliminating induction variables, and transforming reduction statements into a parallel form. We then examined these manually parallelized loop nests to determine what kinds of symbolic analysis are required to guarantee that the applied transformations are legal.

We have chosen the Perfect Benchmarks [6] for our analyses. The Perfect Benchmarks is a suite of 13 Fortran 77 programs that total about 60,000 lines of source code. They represent applications in a number of areas of engineering and scientific computing. In many cases, they represent codes that are currently used by computational research and development groups. Our analyses will only use 12 of the 13 programs. The 13th code, named SPICE, was not examined for we did not have access to a manually parallelized version of the code.

In our comparisons, we will assume that the parallelizing compiler can do certain transformations well, although these transformations may not yet exist in current commercial compilers. These assumptions are needed to prevent one from incorrectly concluding that all symbolic analysis techniques are ineffective for a code just because some other part of the parallelizing compiler is not sufficiently powerful enough to effectively parallelize it. We will assume that the compiler can privatize arrays [43], parallelize loops containing reduction statements, and perform interprocedural dependence analysis to parallelize loops with function calls [13, 12]. We

will also assume that the compiler is capable of performing symbolic analysis techniques that already have been well covered by others. This includes the constant propagation of symbolic expressions [46], the elimination of induction variables [28, 48], and symbolic simplification of expressions [14, 26]. However, we will mention cases where the transformations or analysis techniques above need minor modifications or more accurate information.

In our analysis of the Perfect Benchmarks, we have found a variety of symbolic analysis techniques needed by the transformations described above to achieve the speedups of the manually parallelized versions. Except for *FLO52*, every code required some sort of symbolic analysis technique to improve its performance. However, the distribution of these techniques was quite uneven. Some codes, such as *BDNA* and *MDG*, required only a few techniques to allow parallelizing compilers to match the speedups attained from the manually parallelized versions. Other codes, most notably *QCD* and *TRACK*, need a long succession of complex analysis techniques just to parallelize a single important loop nest.

Rather than examining each code and describing what additional techniques will be needed to effectively parallelize it, we will present the symbolic analysis techniques we have identified, and give examples showing why these techniques are important.

2.3 Symbolic, nonlinear dependence analysis

As mentioned in Chapter 1, current dependence tests can only handle expressions of the form $c_0 + \sum_{j=1}^n c_j i_j$, where c_j are integer constants and i_j are loop index variables. However, expressions with c_j 's that are arbitrary symbolic expressions do exist in important loop nests of the Perfect Benchmarks. Current tests are unable to prove independence for these cases since they are unable to handle the symbolic coefficients and, as a result, consider the expression

nonlinear. However, for many of the nonlinear expressions that we have seen, the compiler can find some permutation of the loop nest, by examining the steps and bounds of the nested loops, where the total range of values that can be accessed by the innermost j of the permuted loops fits within the stride of $(j + 1)$ th innermost loop. For such cases, all the loops may be run in parallel. (We will describe an algorithm that can perform such a test in Chapter 3). For example, there cannot be any output dependences in the following loop:

```

do i = 0, n
  do j = 1, m
    x(m*i + j) = ...
  end do
end do

```

We have seen several cases in our examination of the Perfect Benchmarks where such dependence analysis of nonlinear expressions is needed. In some cases, the nonlinear expressions were introduced by the compiler. In *MDG* and *TRFD*, nonlinear subscript expressions appeared in array subscripts after the elimination of an induction variable in a multiply nested loop. In *ADM*, nonlinear subscript expressions were the result of the linearization of a two dimensional array when a subroutine was inlined in an important loop. For example, the following important loop nest in *MDG* has an induction variable in the following doubly nested loop:

```

do i = 1, nt
  jj = i
  do j = 1, nor1
    var(jj) = var(jj) + ...
    jj = jj + nt
  end do
end do

```

After induction variable recognition, the loop is transformed into:

```

do i = 1, nt
  do j = 1, nor1
    var(nt*j + i - nt)
    = var(nt*j + i - nt) + ...
  enddo
enddo

```

Current data dependence tests would not be able to detect independence of this loop because of the `nt*j` term, but by examining the ranges of elements accessed by the outer loop and the stride of the inner loop, a compiler should be able to prove both loops as independent without too much difficulty. Nonlinear expressions can also occur naturally in programs. For example, nonlinear expressions occurred in many of the important loop nests in *OCEAN*. A simplified version of one of the most important of these loop nests, which takes 20% of the code's sequential execution time, is shown below. Only a symbolic data dependence test that can handle nonlinear expressions can identify all the nested loops as independent.

```

do jl = 1, i2k
  exj = ...
  do jj = jl, 64, 2*i2k
    do mm = 1, 129
      js = 129*jj + mm - 129
      js2 = js + 129*i2k
      h = data(js) - data(js2)
      data(js) = data(js) + data(js2)
      data(js2) = h * exj
    end do
  end do
end do

```

The need for a such a dependence test was first discussed by Eigenmann et. al. [23]. Maslov [36] presented the delinearization algorithm, which can handle any subscript expression $c_0 + \sum_{j=1}^n c_j i_j$ with symbolic loop-invariant expressions for the c_j 's. Essentially, the delinearization algorithm partitions the array expression into several independent subexpressions, and tests these partitions separately for dependences. Haghighat [27] describes how to prove that a subscript expression is strictly increasing or decreasing. By proving that a subscript expression is strictly increasing or decreasing, one can eliminate all self-dependences on the array access with the subscript expression. It can handle a more general class of symbolic expressions than Maslov's. For example, it can prove that there are no output dependences for the array write

$a((i * i - i)/2 + j) = \dots$, where $1 \leq j \leq i$. Unfortunately, it cannot eliminate dependences between two array accesses with unequal array subscripts.

2.4 Constant propagation

One of the fundamental techniques needed for effective symbolic analysis is constant propagation. Constant propagation is a data-flow analysis pass which attempts to determine the constant value of each variable reference, where this value is valid along all execution paths. The constant propagation of integer constants aids analysis of symbolic expressions by eliminating some of the symbolic terms. Constant propagation of symbolic expressions helps analysis by removing loop-variant variables from the subscript expressions. Also, the constant propagation of symbolic values of two or more variables may allow the compiler to determine additional relationships between these variables. The constant propagation of symbolic expressions is sometimes called *forward substitution* by the parallelizing compiler community.

2.4.1 Interprocedural constant propagation with procedure cloning

From our experience with the Perfect Benchmarks, we have found that a constant propagation pass, whether it is for integers or for symbolic expressions, must work interprocedurally [13] to allow many of the codes to be parallelized. Many more constants can be found if the propagator is allowed to pass constants across procedure boundaries. To do this effectively, procedure cloning[12] is often required. One good example for interprocedural constant propagation is the code *OCEAN*, which requires extensive amount of interprocedural constant propagation, along with some procedure cloning, to parallelize seven of its most important loop nests. These loop nests together account for 60% of *OCEAN*'s sequential execution time and at least 84% of its

parallel execution time if they are not parallelized. For example, one important loop in *OCEAN* is below. (Loop normalization, induction variable substitution, forward substitution (constant propagation of symbolics expressions), and dead code elimination was needed to transform the loop nest into the form below.)

```

do j = 0, mtrn-1
  work(1) = c1 * ac(n + q*j + 1)
  do i = 2, m/2, 1
    temp1 = ac(p*(2*i - 1) + q*j + 1)
    .      - ac(p*(2*i - 3) + q*j + 1)
    temp1 = c2 * temp1
    temp2 = ac(p*(2*i - 2) + q*j + 1)
    work(i) = temp1 + temp2
    work(m-i+2) = temp1 - temp2
  enddo
  work(m/2 + 1) = c3
  .      * ac(p*(2*(m/2) - 1) + q*j + 1)
  do i = 1, m, 1
    ac(p*(i-1) + q*j + 1) = work(i)
  enddo
enddo

```

Without constant propagation, this loop is unparallelizable because of the index expressions of array `ac` are of the form similar to $p*i + q*j$. Traditional dependence analyses cannot handle these subscripts because of the non-integer coefficients, and symbolic, nonlinear expression data dependence test cannot handle them because the variables `p`, `q`, `mtrn`, and `m` are not comparable. The array privatization pass would also have difficulties in proving that array `work` is privatizable because the variable `m` must be evenly divisible by 2 for the entire array to be defined by the loop. However, interprocedural constant propagation can assign integer constant values to `p`, `q`, `mtrn`, and `m`. With these constant values, traditional data dependence tests can prove that there are no cross iteration dependences for array `ac`, and the array privatizer can identify the array `work` as privatizable. Thus, interprocedural constant propagation allows all loops in the example above to be identified as parallelizable.

One example of where interprocedural constant propagation of symbolic expressions is needed is the outermost loops in the two most important loop nests in TRFD, which together take more than 99% of the code's sequential execution time. Although the inner loops can be parallelized, the outermost loops must be parallelized to get good speedups from this program. One of these loop nests, which account for 69% of the code's sequential execution time, is shown below. (Induction variable substitution, forward substitution, and dead code elimination was needed to transform the loop nest into the form below.)

```

...
call olda(x, num, num)
...
subroutine olda(x, num, morb)
do mrs = 1, (num*(num + 1))/2
  ...
  do mi = 1, morb
    do mj = 1, mi
      xrsij((mi*(mi-1) + mj)/2 + (mrs-1)*((num*(num + 1))/2)) = ...
    end do
  end do
end do
...

```

To parallelize this loop nest, one needs to use a nonlinear data dependence test. To prove that the outermost loop is parallel, the test must prove that the range of all elements spanned by the inner loops falls within the stride made by the outermost loop. The innermost loops access $(morb*(morb + 1))/2$ adjacent array elements while the outermost loop jumps in strides of $(num*(num + 1))/2$. Now, with interprocedural constant propagation of symbolic constants, the nonlinear dependence test would be able to see that $num = morb$ and thus $(morb*(morb + 1))/2 = (num*(num + 1))/2$ and that the outermost loop is parallel. However, without interprocedural constant propagation, it would not be able to determine any relationship between *morb* and *num* and thus wouldn't be able to prove that the range spanned by the inner loops is

less than or equal to the stride of the outermost loop. Hence, without interprocedural constant propagation, the outermost loop of this loop nest cannot be identified as parallel.

2.4.2 Guarded constant propagation

In some cases, the control flow of a program must be taken into account to discover some constants. More specifically, some variables can take on one of several constant values, dependent upon the values of one or more boolean variables or expressions. We call such constants as *guarded constants*. We have found one code (*ARC2D*) where finding guarded constants are essential for parallelization of an important subroutine (**filerx**) which takes a small but significant amount of the code's parallel execution time (10% if outermost loop is not parallelized).

The definition of these constants are:

```

L1    do j = 1, jmax
        jplus(j) = j+1
        jminu(j) = j-1
    enddo
S1    if (.not. peridc) then
        jplus(jmax) = jmax
        jminu(1) = 1
        jlow = 2
        jup = jmax-1
    else
        jplus(jmax) = 1
        jminu(1) = jmax
        jlow = 1
        jup = jmax
    endif

```

And the simplified body of routine **filerx** is:

```

L2    do n = 1, 4
L3        do j = jlow, jup
            work(j) = ...
        enddo
S2        if (.not. peridc) then
            work(1) = ...
            work(jmax) = work(jmax-1)
        endif
    enddo

```

```

        endif
L4      do j = jlow, jup
        ... = work(jplus(j)) - 2*work(j)
        .   + work(jminu(j))
        enddo
        ...
    enddo

```

To parallelize the loop L2, the compiler must be able to identify that array **work** is privatizable. To do this, it must be able to prove that every access of an array element returns a value generated by a definition of that same element in the same iteration. In another words, the definitions of **work** cover all subsequent uses. By using array range propagation, described later, the compiler can determine that the values of subscript arrays **jplus** and **jminu** range between the values 1 and **jmax**. Therefore, it can show that the loop L4 can use any of the elements in the range **work(1:jmax)**. The compiler can also easily determine that the range **work(jlow:jup)** is defined from loop L3. However, the compiler cannot prove that the array **work** is privatizable because it is unable to determine whether there are any overlaps between the range defined and the range used because it is unable to compare the variables **jlow** and **jup** with 1 or **jmax**. Additionally, the compiler must ignore the two definitions of **work** in the body of the **if** statement S2. However, if control flow is taken into account, it can be seen that **jlow** and **jup** take one of two constant values depending upon the value of the constant boolean **peridc**. More specifically, **jlow** = **peridc** ? 1 : 2 and **jup** = **peridc** ? **jmax** : **jmax**-1, (borrowing the **?:** expression from the C language). Using this information, the compiler can determine that the range **work(1:jmax)** is defined at the start of L4. Thus, the definitions of **work** covers every use in the same iteration and array **work** can be privatized.

Tu and Padua [43] offer an efficient algorithm to propagate guarded constants. We have also seen several examples in *ARC2D*, *MDG*, and *QCD* where control flow must be taken into account in array def/use analysis for array privatization if some important arrays is to be

parallelized. That is, the compiler must be able to identify as privatizable a certain class of arrays that are conditionally defined or that have its definitions and uses in separate conditional statements. For example, the conditional definition of `work` at `S2` in the example above must be taken into account to determine that the entire array is defined at the start of `L4`. Tu and Padua discuss these difficulties in more detail.

2.5 Symbolic constraint propagation

In our opinion, one of the most useful and general of these techniques that we identified was symbolic constraint propagation. Symbolic constraint propagation is the determination of equalities and inequalities between program variables (e.g. `a < b`) at specific points in the program unit. This information can then be used to determine the relationship between two arbitrary expressions. We have found that this ability to determine whether one symbolic expression is less than another is very useful for a variety of compiler passes.

Constraint propagation can be used for a variety of purposes, including:

- Data dependence analysis
- Array privatization
- Dead code elimination

Constraint propagation is needed by symbolic data dependence analysis for several purposes. One common need for constraint propagation are queries whether certain variables are non-zero. For example, in the loop:

```

do i = 1, 100
S1    a(n * i + c) = ...
end do
```

there are no cross-iteration output dependences from **S1** to **S1** if and only if **n** \neq 0. Constraint propagation can also be very useful at identifying that there is no dependence for array subscripts containing loop variant variables. The code *TRACK* has several important loops that need such an analysis. A greatly simplified version of one of these loops is:

```

      ntrold = lsttrk
      do k1 = 1, nm1
        do kt = 1, ntrold
S1          if (k1 .ne. ihits(kt)) then ...
              end do
              if (...) then
                lsttrk = lsttrk + 1
S2          ihits(lsttrk) = ...
              endif
            end do

```

Current data dependence tests are unable to compare the use of **ihits(kt)** at statement **S1** with the definition of **ihits(lsttrk)** at statement **S2**. Therefore, these tests would have to assume that a dependence exists. However, constraint propagation can determine that **lsttrk** $>$ **ntrold** at **S2**. With this information, a compiler can determine that there is no dependence between **S1** and **S2**. Constraint propagation can also be part of the dependence test itself. For example, Banerjee's Inequalities Test can be extended to work with symbolic expressions with the additional information calculated by constraint propagation [26].

Constraint propagation can also be used for array privatization. An array privatizer often needs to determine whether a range of array elements that are defined (**a(1:m)**) covers another constraint of elements used (**a(1:n)**). This requires the comparison of the bounds of these ranges. (e.g. is **m** \geq **n**?). Constraint propagation can improve the accuracy of these tests. One example for constraint propagation occurs in the code *BDNA*. This loop, if not parallelized, would account for 60% of the code's parallel execution time.

```

do i = 1, n
  do k = 1, i-1
S1      xdt(k) = ...
  end do
  l = 0
  do j = 1, i-1
    if (...) then
      l = l + 1
S2      ind(l) = j
    end if
  end do
  do j = 1, l
S3      ... = xdt(ind(j))
  end do
end do

```

To identify array `xdt` as privatizable, the compiler must show that the definition of `xdt` at `S1` covers all uses of `xdt` at `S3`. Unfortunately, the presence of the subscript array `ind` would prevent the privatizer from determining any relationship between `S1` and `S3`. However, analysis of the definition of `ind` at `S2` can easily determine that $\text{ind}(1:1) \leq i-1$. By propagating this relationship to `S3`, the compiler will be able to determine that the `xdt` is privatizable.

Constraint propagation can also be used for dead code elimination of conditional statements. Such an extension to dead code elimination has shown to be very useful to handle last value assignments generated by induction variable substitution. When an induction variable is eliminated in a loop, it is sometimes desirable to place an assignment of the last value of that induction variable after the loop. However, because the loop may be a zero-trip loop, (i.e., it has no iterations), this last assignment must be protected by a conditional statement. This conditional statement prevents the forward substitution of this last value, and thus hinders further analysis. However, by using constraint propagation to prove that the loop is not a zero trip loop, conditional may be eliminated. A good example of this occurs in one of the two important loop nests in *TRFD*. This loop has an induction variable (`mijk1`) in a loop nest that is nested four deep. The innermost two loops of the nest used wrap-around variables, which

prevent the elimination of the induction variable from all four nests. However, after peeling off the first iteration, eliminating the wrap-around variables with forward substitution, and induction variable recognition, we get the code below.

```

do mi = 1, morb
  do mj = 1, mi
    ...
    do ml = mj, mi
      xijkl(mijkl + ml - mj + 1) = ...
    end do
    if (mj - 1 .le. mi)
S1      mijkl = mijkl + mi - mj + 1
    end if
    ...
    do mk = mi + 1, morb
      ...
      do ml = 1, mk
        xijkl((mk*mk - mi*mi
.          - mi - mk)/2
.          + ml + mijkl) = ...
      end do
    end do
    if (mi .le. morb) then
S2      mijkl = mijkl + morb
.          + (morb*morb - mi*mi
.          - mi - morb)/2
    end if
  end do
end do

```

Right now, the conditional statements surrounding the last values for `mijkl` at S1 and S2 prevent induction variable substitution from eliminating `mijkl` entirely from the loop. By using constraint propagation, the compiler can determine that both tests are always true, and can eliminate the conditionals. Without the conditionals, we can forward substitute the value of `mijkl` at S1 into the subsequent uses, then use induction variable recognition to eliminate `mijkl` from the entire loop nest, allowing the outermost loop to be parallelized.

There has been some work in the determination of variable constraints. Much work has been spent in determining the possible range, or interval, of values that variables can take,

for the purpose of array bounds checking or program verification [31, 11]. These algorithms, however, only propagate integer ranges. Cousot and Halbwachs [17] offer a powerful algorithm for determining symbolic linear constraints between variables. Their algorithm is based upon the calculation, intersection, and merging of convex polyhedrons in the n -space of variable values. However, their algorithm cannot handle nonlinear expressions such as $a < b * c$. Although it is not too common, we have seen cases where nonlinear bounds must be propagated or nonlinear expressions must be compared.

We have developed a technique, called Range Propagation, that can compute the symbolic constraints of variables in a program and use these constraints to compare arbitrary expressions. Range Propagation can handle constraints containing nonlinear expressions. Chapter 4 will describe Range Propagation in detail.

2.6 Subscript array analysis

Of all the kinds of symbolic expressions that are difficult to handle by a dependence test, expressions containing array references are usually the worst. This is because the compiler cannot determine any information on the reference pattern of the expression without having some knowledge of the contents of the array. For example, the compiler cannot determine what elements of array `a` are accessed in the expression `a(x(i))` without knowing the value of the subscript array element `x(i)`. Essentially, this problem is similar to the pointer aliasing problem of other languages, such as C.

2.6.1 Array constant propagation

In our analysis of the Perfect Benchmarks, we have found that a small but significant fraction of subscript arrays are constant. That is, they are initialized at the beginning of the program to values that are simple, easily representable expressions, and are not modified afterwards. Thus, we can replace these constant array accesses with their values. To our knowledge, there has not been any previous published work in this area. One example is a loop nest in *TRFD* which, if not parallelized, accounts for 25% of the parallel execution time.

```
L1: do i = 1, num
      ia(i) = (i * (i - 1)) / 2
    end do
    ...
L2: do i = 1, num
      do j = 1, i
S1:        x(ia(i)+j) = ...
      end do
    end do
```

Left as is, the loop nest L2 cannot be parallelized because of a potential output dependence caused by the subscripted subscript *ia*. However, if we propagate the constant value of array *ia*, we would be able to transform the subscript expression *ia(i)+j* into $(i*(i-1))/2 + j$. Using a symbolic nonlinear expression data dependence test and algebraic simplification, the compiler will be able to prove that there are no output dependences and will be able to parallelize the loop.

2.6.2 Analysis of subscript arrays

Although array constant propagation is a very desirable method to handle subscript arrays, it is applicable in only a few cases. However, the compiler may still be able to prove independence or identify an array is privatizable if it had some additional information about the subscript arrays. Some useful properties we would like to determine for subscript arrays are:

- Is the array singly valued?

$$(x(i) = x(j) \text{ if and only if } i = j)$$

- Is the array monotonically increasing/decreasing?

$$(x(i) \leq x(j) \text{ if and only if } i \leq j) \text{ or } (x(i) \geq x(j) \text{ if and only if } i \leq j)$$

- What is the forward difference $(x(i+1) - x(i))$ between elements of the array?

- What is the minimum or maximum value of the array elements.

Knowledge whether the array is singly valued or monotonically increasing or decreasing is very useful for eliminating false dependences [38]. Knowing the forward difference between elements also aids dependence analysis, which will be shown in the example below. The minimum or maximum value of the array is very useful for array privatization, as shown previously in the example for the usefulness of constraint propagation in *BDNA*.

One code that would benefit greatly with subscript array information is *DYFESM*. Many of the important loops in *DYFESM* are of the form:

```
do iblock = 1, nblock
  do i = 1, iblen(iblock)
    ... x(pptr(iblock) + i - 1) ...
  end do
end do
```

Now, the array `pptr` is initialized at the beginning of the program in the following loop:

```
iptr = 1
do i = 1, nblock
  pptr(i) = iptr
  iptr = iptr + iblen(i)
end do
```

A sufficiently powerful compiler should be able to determine that `pptr(i+1) - pptr(i) = iblen(i)`. Now, if the compiler can determine that the minimum value of the elements of `iblen` is greater than zero, the compiler can prove that array `pptr` is strictly increasing and

that all loops like the one above is parallelizable. An additional complication is that the value of the array `iblen` is dependent on input variables, thus requiring a runtime test to allow the transformation.

2.7 Generating runtime tests

Sometimes, to prove a loop is parallelizable at compile time is impossible or too expensive. In these cases, only runtime tests with two version loops can feasibly parallelize them. That is, suppose that a given loop cannot be parallelized unless a certain condition is true. To handle this, the compiler inserts a conditional statement that tests this condition. If it is true, a parallelized version of the loop nest is executed. Otherwise, the program executes the sequential version.

There are several examples in the Perfect Benchmarks where runtime test can improve the compiler's effectiveness on the parallelization of the codes. The code *DYFESM* needs runtime tests to get a significant speedup from it. As described in example in the previous section, the minimum value of the array `iblen` needs to be proven to be greater than zero to determine that many loops do not have cross iteration dependences. Since the value of `iblen` is dependent on input variables, only a runtime test can prove this condition. The cost of this runtime test would be insignificant, since `iblen` is only modified once in the program, and therefore needs to be tested only once.

Another example of the usefulness of runtime tests can be found *ADM* and *MG3D*. Both of these codes need to simplify an expression of the form $(a/b)*b$ to a to prove some crucial arrays as privatizable. Unfortunately, the expression a/b is an integer division, so $(a/b)*b$ equals a only if b evenly divides a . A simplified version of the offending code segment in *ADM* is shown

below. The important loop nests that call the routine containing this code segment account for 43% percent of the sequential execution time. In this code fragment, we wish to prove that all `n` elements of the array `ch` are defined before used in the loop so that it can be privatized in these unseen calling loop nests. If not privatized, these loops must be left serial.

```

na = 0
l1 = 1
do k1 = 1, ifac(2)
  ip = ifac(k1+2)
  l2 = ip*l1
  ido = n/l2
  if (na .eq. 0) then
    call radbg (ido, ip, l1, c, ch, ...)
  else
    call radbg (ido, ip, l1, ch, c, ...)
  endif
  na = 1-na
  l1 = l2
enddo
...
subroutine radbg (ido, ip, l1, cc, ch, ...)
dimension cc(ido,ip,l1)
dimension ch(ido,l1,ip)
define (ch(:, :, :))
use (cc(:, :, :))
return
end

```

Because of the redimensioning of array `cc` and `ch` by `radbg`, we need to prove that the elements of the three dimensional arrays `cc` and `ch` that were accessed in `radbg` access all `n` elements in the caller. This translates to proving that $n = ip * ido * l1$. With a bit of aggressive analysis and expression simplification, this can be reduced to proving that $n = (n / \prod_{i=3}^{k1} ifac(i)) * \prod_{i=3}^{k1} ifac(i)$. Now, to simplify down the right hand side down to n , we need to prove that $\prod_{i=3}^{k1} ifac(i)$ evenly divides n . This constraint holds for *ADM*, since the array `ifac` is initialized to contain all the factors of `n`. To prove this at compile time, however, is complex. The subroutine that initializes `ifac` is very difficult to analyze, mainly because

of unstructured control flow. However, if the compiler assumes that n is evenly divisible by $\prod_{i=3}^{k1} ifac(i)$ and inserts a runtime test to verify this assumption, it will be able to simplify the expression down to n and thus determine `ch` as privatizable. The cost of this test would be insignificant, since the array `ifac` is small and the time by the loop is large. The code *MG3D* suffers from a problem almost identical to this example.

2.8 Other techniques

There were a few other important symbolic analysis techniques that were necessary for the effective parallelization of the Perfect Benchmarks. Because of these techniques are computationally expensive and/or have limited applicability, we do not plan on spending much time on finding solutions to these problems. However, we do believe they are worth mentioning.

2.8.1 Compile time interpretation of programs

One important, but potentially expensive technique is the compile time interpretation of programs. Essentially, the idea is to execute the program without input data. Or in another words, to perform abstract interpretation [16] where the abstractions in the algorithm are kept to a minimum. One example that needs such an analysis is the determination that an array is filled with the factors of some scalar, as described in the example for *ADM* in Section 2.7. Another example occurs in *QCD*, where much of the code cannot be parallelized unless the control flow of a specific routine can be determined. A simplified version of this routine is shown below:

```

subroutine syslop (ptr, nn, ...)
integer ptr(*), nn
nn = 1
do while (ptr(nn) .ne. 14)
  select case (ptr(nn))
    case (1:4)
```

```

        ind = ptr(nn)
        ...
    case (5:8)
        ind = ptr(nn) - 4
        ...
    case (15:18)
        ind = ptr(nn) - 14
        ...
    ...
end select
nn = nn + 1
end do
...
return
end

```

Each case in the case statement above makes modifications to several important arrays that are not shown. The scalar `ind` is used for indexing for some of these arrays. Hence, to determine exactly how these arrays are being modified, exact knowledge of what cases are taken is needed. However, this requires exact knowledge of the contents of the array `ptr`. (In a way, the array `ptr` can be seen as an instruction stream that this routine interprets to determine what operations it should perform on other inputs.) In some cases, the passed in value of `ptr` is constant. For these cases, the compiler can “run” this routine to determine the sequence of operations to perform on the other arrays. In other cases, `ptr` is not constant. However, compile time interpretation may still work since the calculated values of the elements of `ptr` are simple enough to summarize at compile time. For example, `ptr` may have `r1` elements of value `5+dir`, then `r2` elements of value `1+dir`, then the value 14.

2.8.2 Algorithm recognition

Another expensive technique required by some codes is algorithm recognition. Basically, the compiler needs to be able to recognize that a given code fragment implements a certain algorithm so that it may replace it with a parallel version. Algorithm recognition and replacement can be

feasible if the code fragments are small and relatively simple. For example, some commercial parallelizing compilers can replace matrix multiplies or recurrences with library calls. However, when the algorithms to be replaced are longer and more complicated, performing algorithm recognition becomes impractical.

We have seen two examples in the Perfect Benchmarks where the recognition of moderately complex algorithms were required to fully exploit the parallelism in these codes. In *QCD*, a loop nest that took 50% of the serial execution time could not be parallelized because of calls to a random number generator. If this subroutine can be recognized as a random number generator, some actions can be taken to handle these dependencies. For example, the random number generator can be replaced by a parallel version or surrounded by locks. Although this would cause the results of the parallel loop to differ from the results of the serial loop, the results should still be valid. To legally perform this optimization, however, some sort of user interaction would still be required to guarantee its validity. For another code, *SPEC77*, a major loop nest is prevented from being parallelized by a called subroutine that performs a linear search in a sorted array. This subroutine was optimized by the programmer to start its search at the element it found on its last call. Unfortunately, this optimization creates a dependence between calls to this subroutine. So, to parallelize the loop that calls this subroutine, the compiler must recognize that this subroutine uses a linear search in a sorted array, and replace it with a side-effect-free search algorithm.

2.9 Summary

A summary of the techniques required to parallelize the Perfect Benchmarks and their importance is shown in Table 2.1. A number in a cell for a specific code and technique is the estimated

Code	Nonlinear Depend. Anal.	Inter. Const. Prop.	Guard. Const. Prop.	Con- straint Prop.	Const. Array Prop.	Array Anal.	Run Time Tests	Compile Time Interp.	Alg. Recog.	Auto- Matable Speedup
ADM	1.1	3.3		1.8			4.3^2	4.3^2		7.8
ARC2D		1.1	1.1	1.1		1.1				1.5
BDNA				2.1		2.1				4.1
DYFESM	3.6	3.1				3.6	3.6			3.9
FLO52										1.7
MDG	1.2	18.5		1.2						20.3
MG3D		36.9		36.9		36.9	36.9^2	36.9^2		36.9
QCD	11.3			19.0	19.0	19.0		19.0	11.4	19.1
OCEAN	8.3	11.5		8.3					1.5	12.2
SPEC77									2.4	6.4
TRACK				2.9			3.3			5.3
TRFD	18.2		18.2	18.2	1.4					18.2

Table 2.1: Estimated amount of slowdown on Cedar from manually parallelized codes if a specific symbolic analysis technique could not be used.

slowdown incurred from the performance of the manually parallelized code if the technique could not be used. That is, the slowdown equals t_e/t_a , where t_e is the estimated time taken with the technique disabled and t_a is the time taken by the manually parallelized version. The value t_e was calculated by assuming that all important loop nests¹ that use the given technique could not apply the transformations that allowed the faster execution times for the nest. That is, such nests will have a execution time equal to the time taken by the automatically parallelized version of the loop nest, as generated by Kap/Cedar, which is based upon the 1988 version of the commercial parallelizing compiler named Kap. Other important loop nests, which didn't use the technique, have an execution time equal to the manually parallelized version's. An empty cell indicates that no important loops use the given technique. The last column in the table, *Automatable speedups*, displays the speedup from the automatically parallelized codes to the manually parallelized codes using only techniques that could be implemented in a compiler, which is the ratio of the two bars in Figure 1.2.

¹We consider a loop nest as *important* if its parallelization may significantly affect the speedup of the entire program.

One caveat to Table 2.1 is that the techniques are not orthogonal. First, some techniques are dependent upon information provided by others. For example, symbolic nonlinear expression data dependence tests almost always needs the information provided by constraint propagation so that it can effectively compare expressions. Secondly, some important loop nests can be parallelized by either using one symbolic analysis technique or another. The only examples that suffers from this problem in Table 2.1 are the runtime test and compile time interpretation techniques for *ADM* and *MG3D*. For these slowdowns, either of the two techniques can be used to parallelize the important loop(s). These numbers correspond to the problem of determining whether b evenly divides a in the integer expression $(a/b) * b$, as described in Section 2.7.

2.10 Conclusions

We have shown that current commercial parallelizing compilers do poorly on real codes. We have also shown that a compiler can theoretically achieve good speedups for these codes. From this, we deduced that there is significant room for improvement of parallelizing compilers. We then examined the codes to determine what symbolic analysis techniques are required to get these good speedups. The techniques that we identified ranged from minor extensions to current techniques to complex and expensive transformations. The most interesting of these techniques were: symbolic, nonlinear expression data dependence tests, constraint propagation, guarded constant propagation, constant array propagation, subscript array analysis, and the generation of run time tests.

We have implemented some of these techniques within the Polaris parallelizing compiler [39], which is being developed at the University of Illinois. More specifically, we have implemented

²For *ADM* and *MG3D* the given slowdown is incurred only if neither runtime tests nor compile time tests can be used.

an interprocedural constant propagator with procedure cloning, a symbolic, nonlinear dependence test called the Range Test, and a symbolic variable constraint propagator called Range Propagation. The rest of this dissertation will discuss the implementations of the Range Test and Range Propagation.

We believe that the symbolic analysis techniques that we have identified, along with other powerful techniques such as interprocedural analysis, array privatization, improved handling of induction variables, and reduction parallelization, can significantly improve the effectiveness of parallelizing compilers on real codes. Our implementations of interprocedural constant propagation, the Range Test and Range Propagation supports this. For example, Polaris, with these techniques, was able to identify all the parallel loops for all the examples in Section 2.3. The resulting performance of Polaris is shown in Figure 1.3.

We also believe that some of these symbolic analysis techniques will be beneficial for optimizing compilers for superscalar or VLIW processors. For example, these compilers would benefit from a more accurate dependence analysis when they perform code scheduling, since having fewer dependences gives them greater freedom to move code. Another example is to use constraint propagation to eliminate redundant conditional jumps, which should improve performance since conditional jumps may cause the processor's pipeline to stall.

Chapter 3

THE RANGE TEST

3.1 Open issues in data dependence testing

There has been much research in the area of data dependence analysis [3, 25, 37, 42, 50]. Modern day data dependence tests have become very accurate and efficient. However, most of these tests require the loop bounds and array subscripts to be represented as a linear (affine) function of loop index variables. That is, the expressions must be in the form $c_0 + \sum_{j=1}^n c_j i_j$ where c_j are integer constants and i_j are loop index variables. Expressions not of this form are called *nonlinear*.

Because nonlinear expressions prevent the application of dependence tests, parallelizing compilers perform several analyses and optimizations to eliminate nonlinear expressions. For example, constant propagation and induction variable substitution are used to remove loop-variant variables. Other techniques have also been developed to handle additive, loop-invariant, symbolic terms or to eliminate unwanted operations such as divisions [25, 37, 42].

```

L1 : DO i1 = P1, Q1, R1
...   ...
Ln :      DO in = Pn, Qn, Rn
S1 :          A(f(i1, ..., in)) = ...
S2 :          ... = A(g(i1, ..., in))
                      ENDDO
...
ENDDO

```

Figure 3.1: Model of loop nest for dependence testing.

Unfortunately, not all nonlinear expressions can be removed. Because of this, we developed the Range Test, a dependence test that can handle symbolic, nonlinear array subscripts and loop bounds. In the Range Test, we mark a loop as parallel if we can prove that the range of elements accessed by an iteration of that loop do not overlap with the range of elements accessed by other iterations. We prove this by determining whether certain symbolic inequality relationships hold. Powerful variable constraint propagation and symbolic simplification techniques were developed to determine such inequality relationships. To maximize the number of loops found parallel using the Range Test, we examine the loops in the loop nest in a permuted order.

3.2 Data dependence

In this section we will give a brief definition of data dependences. For a more thorough description of data dependence and dependence analysis, see Banerjee et al [5, 3, 50].

To ease the presentation of the Range Test, we will assume that we have a perfectly nested FORTRAN-77 loop nest as shown in Figure 3.1. We will also assume that the tested array A has only one dimension. The array access functions (f and g), the loop's lower and upper bounds (P_i and Q_i), and the loop's stride (R_i) may be arbitrary symbolic expressions made up of loop-invariant variables and loop indices (i.e. i_x) of enclosing loops. We will also assume

that all loop strides (R_i) are positive. It is not difficult to extend our test to handle imperfectly nested loops, negative strides, multidimensional arrays, and loop-variant variables. In fact, our implementation includes these extensions. Section 3.5 will describe how to make these extensions.

3.2.1 Data dependence

We define an index subspace \mathcal{R}_j , where $1 \leq j \leq n$, to be the set of all loop index vectors $(\alpha_1, \dots, \alpha_j)$ that fall within loop bounds of the outermost j loops. More formally,

$$\mathcal{R}_j = \{(\alpha_1, \dots, \alpha_j) \ : \ P_1 \leq \alpha_1 \leq Q_1, \dots, P_j \leq \alpha_j \leq Q_j, \\ (\alpha_1 - P_1) \bmod R_1 = 0, \dots, (\alpha_j - P_j) \bmod R_j = 0\}$$

The conditions $(\alpha_i - P_i) \bmod R_i = 0$ are required to make sure that each α_i can only take on values that are some multiple of the loop's stride from the loop's initial value. The index space \mathcal{R} is defined to be equal to the index subspace \mathcal{R}_n .

A *data dependence* exists between array accesses $A(f(\vec{\alpha}))$ and $A(g(\vec{\beta}))$ if and only if at least one of the two accesses is a write, $f(\vec{\alpha}) = g(\vec{\beta})$, and $\vec{\alpha}, \vec{\beta} \in \mathcal{R}$.

3.2.2 Direction vectors

Suppose that a dependence exists between $A(f(\vec{\alpha}))$ and $A(g(\vec{\beta}))$. Then, the *direction vector* $\vec{d} = (d_1, \dots, d_n)$ for this dependence is defined as:

$$d_i = \begin{cases} < & \text{if } \alpha_i < \beta_i \\ = & \text{if } \alpha_i = \beta_i \\ > & \text{if } \alpha_i > \beta_i \end{cases}$$

(This definition of direction vector is true only for loops with positive strides. For a loop L_i with a negative stride, the $<$ and $>$ cases of definition of d_i are swapped.) Since there may be more than one pair of integer vectors $\vec{\alpha}$ and $\vec{\beta}$ that satisfy the dependence equation, there may be more than one direction vector between the statements S_1 and S_2 .

A dependence is *carried* by loop L_i (or loop at level i) if and only if there exists a dependence vector \vec{d} where $d_1 = '=' , \dots , d_{i-1} = '='$ and $d_i = '<'$. If a loop does not carry any dependences, then that loop may be run in parallel without synchronization.

3.3 The Range Test

The Range Test grew out of a simple observation in our hand analysis of real programs: most parallel loop iterations access adjacent array ranges. These ranges can be very regular (e.g., an inner loop accesses a fixed-length array section and the outer loop strides over this section), they can be increasing or decreasing (e.g., if the two loops are triangular); or, they can be irregular (e.g., if they represent array sections that are carved out of a large array; start and length of the sections are typically stored in index arrays)¹. With one additional observation we can describe the majority of all access patterns: the loops visiting these ranges may be interchanged, such that the access patterns appear “interleaved”. Now, if we managed to prove that such adjacent array ranges do not overlap – possibly “looking through interchanged loops” – we could tell that the loops are parallel. The following section describes such a test formally.

¹The Range Test does not yet handle such subscripted subscript patterns (e.g., $A(X(i))$)

3.3.1 Disproving dependence between symbolic expressions

Essentially, the Range Test disproves carried dependences between $A(f(\vec{i}))$ and $A(g(\vec{i}'))$ for a loop at level j , by proving that the range of elements taken by f and g do not overlap for adjacent iterations of the loop at level j . It determines whether these ranges overlap by comparing the minimum and maximum values of these ranges. The formal definition of these minimum and maximum values are defined below. Section 3.3.4 describes how to compute these minimum and maximum values

Definition 1 *Let $f_j^{\min}(i_1, \dots, i_j)$ and $f_j^{\max}(i_1, \dots, i_j)$ be functions that obey the following constraints:*

$$\begin{aligned} f_j^{\min}(i_1, \dots, i_j) &\leq \min \left\{ f(\vec{i}') : \vec{i}' \in \mathcal{R}, i'_1 = i_1, \dots, i'_j = i_j \right\} \\ f_j^{\max}(i_1, \dots, i_j) &\geq \max \left\{ f(\vec{i}') : \vec{i}' \in \mathcal{R}, i'_1 = i_1, \dots, i'_j = i_j \right\} \end{aligned}$$

Intuitively, $f_j^{\min}(i_1, \dots, i_j)$ and $f_j^{\max}(i_1, \dots, i_j)$ are functions that return the minimum and maximum values that f may take for a particular iteration of the outermost j loops. In our implementation of the Range Test, these functions are represented as symbolic expressions made up of loop indices i_1, \dots, i_j and loop-invariant variables.

The ability to determine the minimum or maximum of f or g in respect to some set of loops leads us to our first dependence test. If the maximum of f is less than the minimum of g in respect to some subset of loops, then these loops cannot carry any dependences. The theorem below states this formally.

Theorem 1 *If $f_j^{\max}(i_1, \dots, i_j) < g_j^{\min}(i_1, \dots, i_j)$ for all $(i_1, \dots, i_j) \in \mathcal{R}_j$, then there can be no dependences between $A(f(\vec{i}))$ and $A(g(\vec{i}'))$ with a direction vector \vec{d} of the form $d_1 = '=' , \dots , d_j = '='$.*

PROOF. Suppose that such a dependence exists, (i.e., $f(\vec{i}) = g(\vec{i}')$ with direction vector \vec{d}). By Definition 1, we have $f(\vec{i}) \leq f_j^{\max}(i_1, \dots, i_j)$ and $g_j^{\min}(i'_1, \dots, i'_j) \leq g(\vec{i}')$. Because of the direction vector \vec{d} , $g_j^{\min}(i'_1, \dots, i'_j) = g_j^{\min}(i_1, \dots, i_j)$. Since $f_j^{\max}(i_1, \dots, i_j) < g_j^{\min}(i_1, \dots, i_j)$, it must hold that $f(\vec{i}) < g(\vec{i}')$. Contradiction. \square

Theorem 1 currently can only disprove dependences, with direction vectors of the form $(=, \dots, =, *, \dots, *)$. With a small modification, it can be enhanced to disprove direction vectors of the form $(=, \dots, =, <, *, \dots, *)$. That is, a direction vector \vec{d} of the form $d_1 = '=', \dots, d_j = '='$ and $d_{j+1} = '<'$. This can be done by tightening the bounds on index i_{j+1} to be $P_{j+1} \leq i_{j+1} \leq Q_{j+1} - R_{j+1}$ when computing $f_j^{\max}(i_1, \dots, i_j)$ and tightening the bounds on i'_{j+1} to be $P_{j+1} + R_{j+1} \leq i'_{j+1} \leq Q_{j+1}$ when computing $g_j^{\min}(i_1, \dots, i_j)$. Remember that we assume that the loop stride R_{j+1} is positive. Such an optimization is useful for disproving loop-carried dependences for ranges of array accesses that do not overlap except for the very first or very last iteration of the loop. In our experience, such ranges do occur in real programs.

Theorem 1 proves that there are no carried dependences between $A(f(\vec{i}))$ and $A(g(\vec{i}'))$ for loops with indices i_{j+1}, \dots, i_n , if the range of possible values taken by f for these loops does not overlap with the range of possible values taken by g . However, it cannot prove that there are no carried dependences for a certain loop if the possible values taken by f and g are interleaved for that loop. Figure 3.2 shows some examples of how array accesses can be interleaved for a particular loop nest. We have found such examples do occur often in practice. All these examples assume that we have a loop nest of the form

```

L1 : DO i = 0, n - 1
L2 :   DO j = 0, n - 1
S1 :     A(f(i, j)) = ...
S2 :     ... = A(g(i, j))
      ENDDO
    ENDDO

```

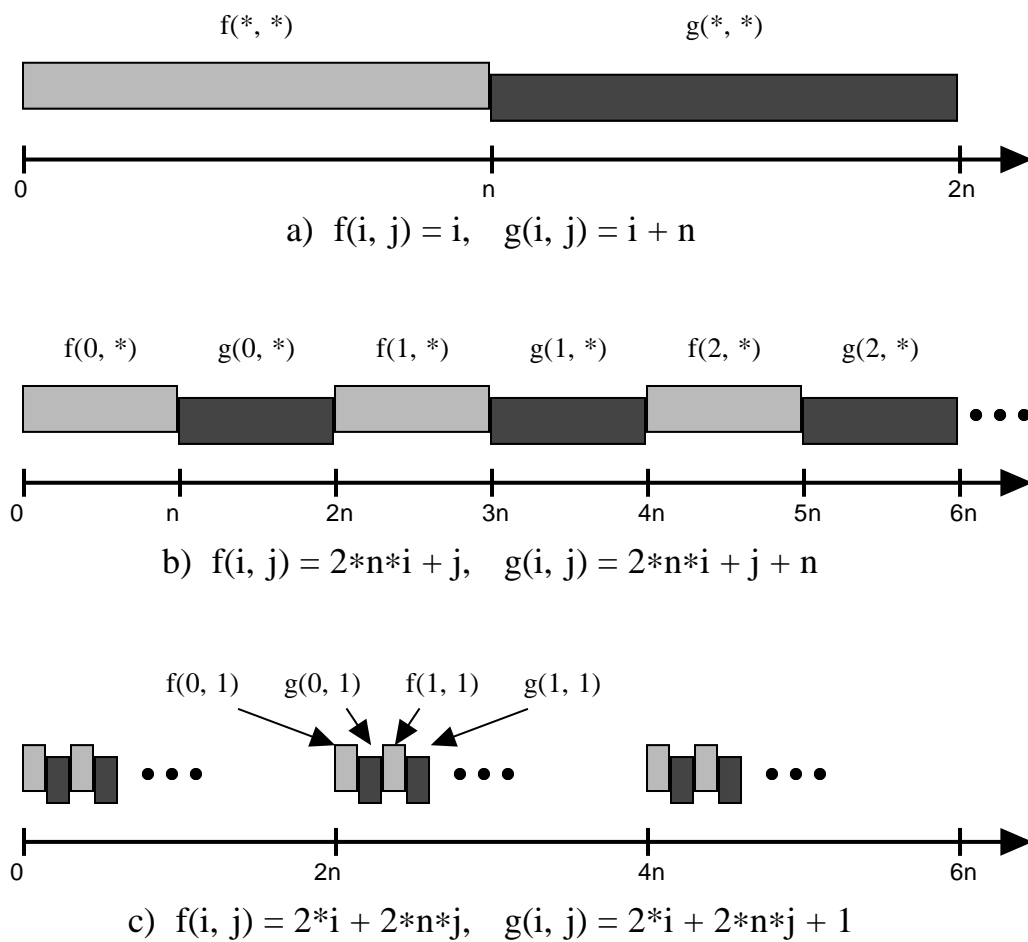


Figure 3.2: Examples of how array accesses can be interleaved in respect to a particular loop (loop with index i for these examples.) All examples assume that $0 \leq i, j < n$.

and that the Range Test is currently attempting to prove that S_1 and S_2 do not carry dependences for loop L_1 . For figure 3.2a, the range of accesses made by $A(f(i, j))$ and $A(g(i, j))$ never overlap, so Theorem 1 can prove that loop L_1 does not carry a dependence for this access pair. However, the set of accesses made by $A(f(i, j))$ and $A(g(i, j))$ are interleaved in figures 3.2b and 3.2c, causing the test from Theorem 1 to fail, even though there isn't a carried dependence. We will present a second dependence test that can disprove carried dependences for a special case of these interleavings, where the possible values taken by f and g for a single iteration are not interleaved with the possible values taken by other iterations of f and g . Figure 3.2b shows an example of this case. However, before we describe this test, we must define the property of *monotonicity* for a particular loop index. (We will deal with Figure 3.2c in Section 3.3.2.)

Definition 2 A function $f(\vec{i})$ is monotonically non-decreasing for index i_j iff $f(i_1, \dots, \alpha_j, \dots, i_n) \leq f(i_1, \dots, \beta_j, \dots, i_n)$ whenever $P_j \leq \alpha_j \leq \beta_j \leq Q_j$.

Similarly, a function $f(\vec{i})$ is monotonically non-increasing for index i_j iff $f(i_1, \dots, \alpha_j, \dots, i_n) \geq f(i_1, \dots, \beta_j, \dots, i_n)$ whenever $P_j \leq \alpha_j \leq \beta_j \leq Q_j$.

We can prove whether an expression is monotonically non-decreasing for a loop level j by proving that the difference $f(i_1, \dots, i_j + 1, \dots, i_n) - f(i_1, \dots, i_j, \dots, i_n)$ is always greater than or equal to zero, using the techniques described in Section 3.4. Similarly, we can prove whether an expression is monotonically non-increasing for a loop level j by proving that the difference is always less than or equal to zero.

Using this definition, we will now show how one can disprove dependences carried at level j when the possible values taken by f and g are not interleaved for a single iteration of the loop at level j .

Theorem 2 *If $g_j^{\min}(i_1, \dots, i_j)$ is monotonically non-decreasing for i_j and if $f_j^{\max}(i_1, \dots, i_j) < g_j^{\min}(i_1, \dots, i_j + R_j)$ for all $(i_1, \dots, i_j) \in \mathcal{R}_j$ and for $P_j \leq i_j \leq Q_j - R_j$, then there can be no dependences from $\mathbf{A}(f(\vec{v}))$ to $\mathbf{A}(g(\vec{v}'))$ with a direction vector \vec{d} of the form $d_1 = '='$, \dots , $d_{j-1} = '='$, $d_j = '<'$.*

PROOF. Suppose that such a dependence exists, (i.e., $f(\vec{v}) = g(\vec{v}')$ with direction vector \vec{d}). By Definition 1, we have $f(\vec{v}) \leq f_j^{\max}(i_1, \dots, i_j)$ and $g_j^{\min}(i'_1, \dots, i'_j) \leq g(\vec{v}')$. Because of the direction vector \vec{d} and because $g_j^{\min}(i_1, \dots, i_j)$ is monotonically non-decreasing for index i_j , $g_j^{\min}(i'_1, \dots, i'_j) \geq g_j^{\min}(i_1, \dots, i_j + R_j)$. (Remember that we assumed that R_j is always positive.) Since $f_j^{\max}(i_1, \dots, i_j) < g_j^{\min}(i_1, \dots, i_j + R_j)$, it must hold that $f(\vec{v}) < g(\vec{v}')$. Contradiction. \square

Theorem 3 *If $g_j^{\min}(i_1, \dots, i_j)$ is monotonically non-increasing for i_j and if $f_j^{\max}(i_1, \dots, i_j) < g_j^{\min}(i_1, \dots, i_j - R_j)$, for all $(i_1, \dots, i_j) \in \mathcal{R}_j$ and for $P_j + R_j \leq i_j \leq Q_j$, then there can be no dependences from $\mathbf{A}(g(\vec{v}'))$ to $\mathbf{A}(f(\vec{v}))$ with a direction vector \vec{d} of the form $d_1 = '='$, \dots , $d_{j-1} = '='$, $d_j = '<'$.*

PROOF. Similar to proof of Theorem 2.

By definition of loop-carried dependences, the test from Theorem 2 (or Theorem 3) must be applied twice to prove that a pair of access functions f and g do not carry dependences for the loop with index i_j : once to disprove a dependence with direction vector \vec{d} from f to g , and once to disprove a dependence with direction vector \vec{d} from g to f . Also note that the tests from these two theorems can disprove a dependence direction \vec{d} from f to g and from g to f ,

and thus disprove that the loop with index i_j does not carry a dependence, only if both f and g are monotonically non-decreasing or monotonically non-increasing for i_j .

In the previous definitions and theorems, we have assumed that the subset of loops that we are attempting to disprove dependence for are the innermost loops with indices i_j to i_n . With some minor changes in notation, these definitions and theorems still hold for arbitrary subsets of loops. The following subsection will exploit this property.

3.3.2 Permuting loops for dependence testing

As described earlier, the test from Theorem 1 can be used to prove independence when the values of access functions f and g are not interleaved, and the tests from Theorems 2 and 3 can be used to prove independence when the values of f and g are interleaved, but the values taken by f and g for a loop iteration are not interleaved with values taken by other iterations. For more complex interleavings, the tests from all three Theorems would fail. Figure 3.2c gives an example of one of these more complex interleavings. Fortunately, we have observed that most of these interleavings can be eliminated by permuting the order in which we test the loops. For example, if loop L_1 , with index i , and loop L_2 , with index j were “swapped” so that L_2 is now treated as the outermost loop, we would be able to use Theorem 1 to prove that there are no carried dependences in the “inner” L_1 loop, and use Theorem 2 to prove that there are no carried dependences in the “outer” L_2 loop.

So, the Range Test attempts to maximize the number of loops that it can identify as not carrying dependences by applying its tests upon a permuted ordering of the loops in the loop nest. The Range Test does not physically permute the loops; it is done logically and temporarily by the test. Also, the Range Test only tries those permutations such that all loops identified as

not carrying any dependences for the permuted loop nest will also not carry any dependences in the original nest.

The Range Test uses a heuristic to find a valid logical permutation of a loop nest, which seems to be quite acceptable in practice. This heuristic determines a valid permutation of a loop nest by recursively finding a valid permutation of the inner loops of the loop nest, then finding a location where it may safely insert the outermost loop in this permutation. The final location of the outermost loop is found by repeatedly moving inwards by one until it reaches a location where it can be proven that it carries no dependences, or the loop just inside this location either carries a dependence or would carry a dependence if outermost loop was inserted inside of it.

To prove that all loops not carrying dependences in the permuted loop nest generated by the heuristic above also do not carry dependences in the original loop nest, we will need the following lemma. For similar lemmas and theorems on loop permutations, see the paper by Banerjee [4].

Lemma 1 *A loop that does not carry a dependence can be legally moved deeper into the loop nest and all loops that didn't carry a dependence beforehand would still not do so.*

PROOF. Suppose that Lemma 1 is false. Let the loop at level i (L_i) be the loop that was moved deeper into the loop nest to level $k > i$. Now, since some loop carries a dependence where it did not do so beforehand, and since a loop carries a dependence only if there exists a dependence direction vector for which it carries a dependence, there must exist at least one direction vector \vec{d} for the loop nest that carries a dependence for some loop that it did not do so beforehand. Now, since moving loop L_i deeper just permutes the directions in a direction vector, the original value of direction vector \vec{d} must have also have carried a dependence for

some loop at level $j \neq i$ (L_j). Thus, moving loop L_i deeper in the loop nest must have modified direction vector \vec{d} in such a fashion that, although it once carried a dependence at loop L_j , it now carries a dependence for some other loop. By definition of carried dependences, $d_1 = '='$, \dots , $d_{j-1} = '='$, and $d_j = '<'$, for the original direction vector. Now, since L_i did not carry a dependence in the original loop nest, $i > j$ or $d_i = '='$. Now, if $d_i = '='$, then no matter where L_i is moved, loop L_j would still carry a dependence. On the other hand, if $d_i \neq '='$, then the new level k of L_i is still larger than j , since $k > i > j$. Thus \vec{d} would still carry a dependence for L_j . Contradiction. \square

We will prove that the heuristic generates a legal permutation by induction. For the base case, where the loop nest is a single loop, the permutation is trivially legal. For the inductive step, assume that the heuristic generates legal permutations for loop nests of i loops. For a nest of $i + 1$ loops, the heuristic first recursively finds a permutation of the innermost i loops, then finds a location for the $(i + 1)$ th loop in this permutation. By the inductive hypothesis, the recursive first step results in a legal permutation. For the second step, which moves the $(i + 1)$ th loop inwards, all loops between the original and final positions of the $(i + 1)$ th loop do not carry dependences, by definition of the heuristic. Thus, we can undo this second step by moving all these loops back inside the $(i + 1)$ th loop; and, by Lemma 1, all loops not carrying dependences still do not so. Therefore, the heuristic generates a legal permutation for a nest of $i + 1$ loops.

3.3.3 Algorithm

The algorithm for the Range Test, which implements the permutation heuristic described previously, is displayed in Figure 3.3. It generates the permuted loop nest, represented by the

ordered list P , by visiting each loop L_i in the original loop nest, from innermost to outermost, and finding its proper location in the set of inner permuted loops P . Simultaneously, the algorithm determines whether each L_i carries any dependences. Loops proven not to carry dependences are added to the set D . The outer **for** loop visits each loop L_i while the inner **while** loop determines whether the current L_i carries any dependence and where to insert it in the list of permuted inner loops P . Statement S_1 tests if L_i does not carry a dependence at a particular location in P . If it doesn't carry a dependence, it is added to the set of parallel loops D and inserted into P at this location. Statement S_2 tests if it is legal to move the current location of loop L_i in P inward by one. If not, it is inserted into P at this location. (It is not legal to move the current location of L_i in P inwards by one if the current location is the innermost location in P , if the next inner loop in P carries a dependence, or if the next inner loop in P would become carry a dependence should L_i be inserted inside of it.) Statement S_3 performs the actual insertion of L_i into P .

Functions RTEST1 and RTEST2 are displayed in Figure 3.4. Function RTEST1, which applies Theorem 1, returns true if and only if it can prove that the loops in \mathcal{L} do not carry any dependences for access functions f and g . Function RTEST2, which calls RTEST2x that implements Theorems 2 and 3, returns true if and only if it can prove that loop L_j carries no dependences for access functions f and g and the inner permuted loops \mathcal{L} . The functions MIN and MAX represent the f_j^{\min} and the g_j^{\max} functions described earlier. (The expression f_j^{\max} , used in Section 3.3, can be computed by calling $\text{MAX}(f, \{L_{j+1}, \dots, L_n\})$. The expression f_j^{\min} is similar.) The function $\text{MAX}(f, \mathcal{L})$ returns the maximum value that function f can take for the indices of the loops in \mathcal{L} . This maximum value is a symbolic expression made up of

INPUT: Normalized, perfectly nested loops (L_1, \dots, L_n) and array access functions f and g .
OUTPUT: Set of loops D that do not have carried dependences between f and g .

```

 $P \leftarrow ()$  (*  $P$  is an ordered list representing the permuted loop nest *)
 $D \leftarrow \emptyset$ 
for  $i \leftarrow n$  downto 1 do
   $placed \leftarrow \text{false}$ 
   $j \leftarrow 0$ 
  while not  $placed$  do
     $j \leftarrow j + 1$ 
 $S_1$  : if  $\text{RTEST1}(f, g, \{L_i, P_j, \dots, P_{|P|}\})$  or  $\text{RTEST2}(f, g, L_i, \{P_j, \dots, P_{|P|}\})$  then
  (* Loop  $L_i$  does not carry dependences at this point. *)
   $D \leftarrow D \cup \{L_i\}$ 
   $placed \leftarrow \text{true}$ 
 $S_2$  : else if  $j = |P| + 1$  or  $P_j \notin D$  or not  $\text{RTEST2}(f, g, P_j, \{L_i, P_{j+1}, \dots, P_{|P|}\})$  then
  (* Loop  $L_i$  cannot be safely permuted any deeper. *)
   $placed \leftarrow \text{true}$ 
end if
end while
 $S_3$  :  $P \leftarrow (P_1, \dots, P_{j-1}, L_i, P_j, \dots, P_{|P|})$ 
end for

```

Figure 3.3: The Range Test algorithm.

loop-invariant variables and indices of loops not in \mathcal{L} . The function MIN is similar. The implementations of MIN and MAX will be described in the next subsection.

3.3.4 Computing f_j^{\min} and f_j^{\max}

There are several methods in which one can compute the minimum or maximum of an expression for a subset of loop indices. One simple but powerful method is to substitute the lower or upper bound of each index, depending on the index's monotonicity. Figure 3.5 shows how the Range Test computes the maximum of an expression for a given set of loops. The algorithm for computing the minimum is very similar; simply switch the monotonically non-decreasing and monotonically non-increasing cases. It can be proven that the result of these functions meets Definition 1; that is, they are the minimum or maximum of f in the subspace spanned by indices i_{j+1}, \dots, i_n .

```

boolean function RTEST1( $f, g, \mathcal{L}$ )
    (* Apply Theorem 1 *)
 $R_1$  : return MAX( $f, \mathcal{L}$ ) < MIN( $g, \mathcal{L}$ )
 $R_2$  : or MAX( $g, \mathcal{L}$ ) < MIN( $f, \mathcal{L}$ )
end function

boolean function RTEST2( $f, g, L_j, \mathcal{L}$ )
    return  $f$  and  $g$  are both mono. non-decreasing or mono. non-increasing for  $L_j$ 
 $R_3$  : and RTEST2x( $f, g, L_j, \mathcal{L}$ )
 $R_4$  : and RTEST2x( $g, f, L_j, \mathcal{L}$ )
end function

boolean function RTEST2x( $f, g, L_j, \mathcal{L}$ )
    (* Apply Theorem 2 or 3 *)
     $s \leftarrow$  MAX( $f, \mathcal{L}$ )
     $t \leftarrow$  MIN( $g, \mathcal{L}$ )
    if  $t$  is mono. non-decreasing for  $L_j$  then
         $t \leftarrow t$  with  $i_j$  substituted by  $i_j + R_j$ 
    else
         $t \leftarrow t$  with  $i_j$  substituted by  $i_j - R_j$ 
    endif
    return ( $s < t$ )
end function

```

Figure 3.4: Algorithm for disproving carried dependences for loop L_j in respect to loops \mathcal{L} . Loop L_j is assumed to have index i_j . The word *monotonically* has been abbreviated to “mono”.

Although the algorithm for computing the maximum in Figure 3.5 is powerful, a naive implementation of it, which computes the monotonicity of the intermediate maximum expression y for each index, can be inefficient. This is because each monotonicity computation for y requires a symbolic expression comparison, which can be quite expensive. To avoid these costs, the Range Test attempts to determine the monotonicity states of y from the precomputed monotonicity states of the input access expression f and the the loop bounds. (The monotonicity states of all array accesses f and all loop bounds are computed only once, at the beginning of dependence testing of the program.) More specifically, the Range Test initially sets the monotonicity states of y to the monotonicity states of f , then update y ’s monotonicity


```

expression function MAX( $f, \mathcal{L}$ )
   $y \leftarrow f$ 
  for each  $L_k \in \mathcal{L}$  from innermost to outermost loops do
    if  $y$  is mono. non-decreasing for  $i_k$  then
       $y \leftarrow y$  with  $i_k$  substituted with  $Q_k$ 
    else if  $y$  is mono. non-increasing for  $i_k$  then
       $y \leftarrow y$  with  $i_k$  substituted with  $P_k$ 
    else
       $y \leftarrow +\infty$ 
    end if
  end for
  return  $y$ 
end function

```

Figure 3.5: Algorithm for calculating the maximum value of function f for fixed values of indices i_j of loops L_j , where $L_j \notin \mathcal{L}$.

states after each substitution, using the monotonicity states of the substituted loop bound. For those cases where the Range Test couldn't determine the new monotonicity of y for index i from the old monotonicity of y for i and the monotonicity of the substituted loop bound for i , it marks the monotonicity of y for i as unknown. Later, when it finds the monotonicity of y for i to be marked as unknown when it is substituting for i , it computes the monotonicity for i using an expensive symbolic expression comparison. We have found this optimization to be very effective in practice. For many array accesses, the monotonicity states of y never need to be computed with symbolic comparisons.

3.3.5 Time complexity

Since the Range Test spends nearly all of its time performing symbolic expression comparisons, its time complexity can be characterized by the number of symbolic comparisons performed. These comparisons occur explicitly in the functions RTEST1 and RTEST2X and implicitly in the monotonicity tests of MIN and MAX. Since the Range Test may call RTEST1 and RTEST2 as many as $O(n^2)$ times, where n is the loop nest depth, and RTEST1 and RTEST2

call MIN and MAX, which performs at most $O(n)$ symbolic comparisons to determine monotonicity for each index, the Range Test performs at most $O(n^3)$ symbolic comparisons for one pair of array accesses ($A(f(\vec{i}))$ and $A(g(\vec{i}'))$). In practice, only a few permutations are examined and at most a constant number of symbolic comparisons are done by the monotonicity tests of MIN and MAX. So, the average number of symbolic comparisons done by the the Range Test is near $O(n)$.

Unfortunately, determining the costs of symbolic expression comparison is much more difficult. The worst case performance of symbolic comparisons is exponential on the size of the expressions compared and upon the number of variables in the program. However, the average case performance is much better.

3.4 Symbolic range propagation

To provide a facility for comparing symbolic expressions, we have developed a technique called *Range Propagation*. Range Propagation will be described in detail in Chapter 4. However, since the Range Test is built on top of the expression comparison facilities of Range Propagation, we will give a brief sketch of Range Propagation in this section.

Range Propagation consists of two parts: the range propagation algorithm and an expression comparison facility. The range propagation algorithm collects and propagates variable constraints through a program. The expression comparison facility uses these variable constraints to determine arithmetic relationships between two symbolic expressions.

The range propagation algorithm centers on the collection and propagation of symbolic lower and upper bounds on variables, called ranges, through a program unit. Abstract interpretation [16] is used to compute the ranges for variables at each point of a program unit. That

is, the algorithm “executes” the program by following the control flow paths of the program, updating the current ranges to reflect the side effects of the statements encountered along these paths, until a fixed point is reached.

We compare two expressions by calculating the integer range spanned by their difference, then determining whether this range is always positive or always negative. This integer range is calculated by repeatedly substituting ranges for variables in the difference expression then simplifying the expression down, until all variables are eliminated.

For example, suppose we wish to compare $x * y + 1$ with y , where $x = [y : 10]$, (meaning $y \leq x \leq 10$), and $y = [1 : \infty]$. First, we calculate the difference, which is $x * y - y + 1$. Then, we substitute $[y : 10]$ for x in $x * y - y + 1$, getting $[y : 10] * y - y + 1$. Simplifying this expression down, we get the range $[y * (y - 1) + 1 : 9 * y + 1]$. Since the simplified range still contains variables, we substitute $[1 : \infty]$ for y , getting $[[1 : \infty] * ([1 : \infty] - 1) + 1 : 9 * [1 : \infty] + 1]$. After simplification, this becomes $[1 : \infty]$. From this range, we can now see that $x * y + 1 > y$.

3.5 Generalizing the Range Test

In our description of the Range Test, we made some assumptions on the form of loop-nests to ease its presentation. More specifically, we assumed that all array accesses are one-dimensional, loops have a positive stride, none of the array accesses nor loop bounds contain loop-variant variables that are not loop indices, and that the enclosing loops between the two accesses being tested are perfectly nested. In this section, we will describe how one can remove these assumptions.

3.5.1 Multidimensional arrays

We handle multidimensional arrays simply by applying the Range Test to each dimension of the array subscript, then intersecting all the sets of loops that we found to carry dependences.

3.5.2 Negative strides

In our presentation of the Range Test, we assumed that all loop strides are always positive. For always negative strides, we use a modification of Theorems 2 and 3 and functions MIN and MAX to disprove dependences.

The only modifications that need to be made to functions MIN and MAX functions is to swap the substitutions of P_k with the substitutions of Q_k when loop L_k has an always negative stride, (see Figure 3.5). The bounds must be swapped simply because when loop L_k has a negative stride, P_k would be larger than Q_k .

For the Theorems 2 and 3, one just needs to swap the terms “monotonically non-decreasing” and “monotonically non-increasing” in the theorems when loop L_j has an always negative stride. This swapping of terms is required because of the definition of direction vectors for loops with negative strides. That is, a dependence with a dependence direction $d_j = '<'$, where loop L_j has a negative stride, means that there is a dependence between two iterations i_j and i'_j of L_j , where $i_j > i'_j$. To ensure that the theorems stay correct, (see the proof of Theorem 2), one must invert the monotonicity condition on $g_j^{min}(i_1, \dots, i_j)$ for index i_j .

For strides that cannot be proven to be always positive or always negative, our implementation fails and marks the loops with these strides as loops that carry dependences.

3.5.3 Loop-variant variables

Handling loop-variant variables in expressions is not difficult. Loop variant variables are variables whose value may change within a loop and that are not a loop index for an enclosing loop. One just needs to modify functions MIN and MAX to eliminate all variables that are loop-variant for the loops in \mathcal{L} from their results. A loop-variant variable can be eliminated by substituting it with the range that Range Propagation has computed for it. Function RTEST2x must also be modified to eliminate any loop-variant variables for loop L_j from expressions s and t , (see Figure 3.4).

3.5.4 Loops that aren't perfectly nested

Handling imperfectly nested loop nests is simple. Just modify functions MIN(f, \mathcal{L}) and MAX(f, \mathcal{L}) to always substitute the indices of loops that enclose only the array access $A(f(\vec{i}))$. This would ensure that any computation of f_j^{\min} (or f_j^{\max}) would also include the maximum (or minimum) of all the values that f can take for all iterations of the loops that enclose only access $A(f(\vec{i}))$. We perform the same optimization to g_j^{\min} and g_j^{\max} to eliminate loops that enclose only access $A(g(\vec{i}'))$.

3.6 Examples

In this section, we will provide examples of important loop nests, taken from the Perfect Benchmarks [6], that the Range Test can determine to be parallel but which conventional data dependence tests cannot.

One example is a loop nest taken from subroutine FTRVMT from the code *OCEAN*. This loop nest accounts for 44% of the code's sequential execution time on an Alliant FX/80. A

```

D0 j1 = 0, i2k - 1
  exj = ...
  D0 jj = 0, x(j1)
    D0 mm = 0, 128
      js = 258*i2k*jj + 129*j1 + mm + 1
      js2 = js + 129*i2k
      h = data(js) - data(js2)
      data(js) = data(js) + data(js2)
      data(js2) = h * exj
    ENDDO
  ENDDO
ENDDO

```

Figure 3.6: Simplified version of loop nest FTRVMT/109 from *OCEAN*.

L_i	P_j	Stmt	Test	Comparison results			
mm		S_1	R_1	$258 * i2k * jj + 129 * j1 + 129$	$<$	$258 * i2k * jj + 129 * j1 + 129 * i2k + 1$	
jj	mm	S_1	R_1	$+\infty$	\nless	$129 * j1 + 129 * i2k + 1$	
jj	mm	S_1	R_2	$+\infty$	\nless	$129 * j1 + 1$	
jj	mm	S_1	R_3	$258 * i2k * jj + 129 * j1 + 129$	$<$	$258 * i2k * jj + 129 * j1 + 387 * i2k + 1$	
jj	mm	S_1	R_4	$258 * i2k * jj + 129 * j1 + 129 * i2k + 129$	$<$	$258 * i2k * jj + 129 * j1 + 258 * i2k + 1$	
j1	jj	S_1	R_1	$+\infty$	\nless	$129 * i2k + 1$	
j1	jj	S_1	R_2	$+\infty$	\nless	1	
j1	jj	S_1	R_3	$+\infty$	\nless	$129 * j1 + 129 * i2k + 130$	
j1	jj	S_2	R_3	$258 * i2k * jj + 129 * i2k$	$<$	$258 * i2k * jj + 387 * i2k + 1$	
j1	jj	S_2	R_4	$258 * i2k * jj + 258 * i2k$	$<$	$258 * i2k * jj + 258 * i2k + 1$	
j1	mm	S_1	R_1	$258 * i2k * jj + 129 * i2k$	$<$	$258 * i2k * jj + 129 * i2k + 1$	

Table 3.1: Trace of Range Test for loop nest FTRVMT/109 shown in Figure 3.6.

```

      mrsij0 = 0
      DO mrs = 0, (num*num+num)/2 - 1
        mrsij=mrsij0
        DO mi = 0, num - 1
          DO mj = 0, mi - 1
            S1 :      mrsij = mrsij + 1
            S2 :      xrsij(mrsij) = xij(mj)
          ENDDO
        ENDDO
      ENDDO
      mrsij0=mrsij0+(num*num+num)/2
    ENDDO

```

Figure 3.7: Simplified version of loop nest OLDA/100 from *TRFD*.

simplified version of this loop is shown in Figure 3.6. Conventional data dependence tests cannot prove that these loops do not carry any dependences because of the $258 * i2k * jj$ term in the subscripts for array **data**. The Range Test, on the other hand, can do so.

Table 3.1 shows a trace of the Range Test when it is proving that there are no loop-carried dependences between the array access functions $f(j1, jj, mm) = 258 * i2k * jj + 129 * j1 + mm + 1$ and $g(j1, jj, mm) = 258 * i2k * jj + 129 * j1 + mm + 129 * i2k + 1$. The final column of this table shows the symbolic expressions compared at the given statement of the Range Test algorithm (S_i) and the RTEST functions (R_j). The results of these comparisons were calculated from the variable constraints determined by the range propagation algorithm, which are $i2k \geq 1$, $0 \leq jj \leq x(j1)$, $0 \leq j1 \leq i2k - 1$, and $0 \leq mm \leq 128$. Since the upper bound $x(j1)$ of loop jj is not monotonic, the test used $+\infty$ as an approximation of this bound. For this pair of access functions, the Range Test had to use Theorems 1, 2, and 3 and permute the $j1$ loop inside the jj loop to prove that there are no loop-carried dependences.

Another important loop nest, which needs a dependence test for symbolic, nonlinear expressions, can be found in subroutine OLDA from the code *TRFD*. A simplified version of this loop

nest is shown in Figure 3.7. This loop nest accounts for 69% of the code’s sequential execution time on an Alliant FX/80.

To parallelize this loop nest, induction variable substitution must be used to replace the induction variable `mrsij` at statement S_1 with the statement:

$$\text{mrsij} = (\text{mi} * 2 - \text{mi} + \text{mrs} * (\text{num} * 2 + \text{num})) / 2 + \text{mj} + 1.$$

After this substitution, conventional data dependence tests cannot prove that there are no self-dependences for `xrsij` at S_2 because of the nonlinear array subscript (after forward-substituting the value of `mrsij`). The Range Test, on the other hand, would have no difficulties in proving that this array has no self-dependences.

3.7 Measurements

To measure the effectiveness and speed of the Range Test, we compared its results with the Omega Test [42]. Roughly, the Omega Test is a variant of integer Fourier-Motzkin analysis [19, 47] with optimizations to make the common cases fast. For affine array subscripts and loop bounds, the Omega Test is an exact data dependence test. The Omega Test handles non-affine expressions using uninterpreted function symbols. Our implementation of the Omega Test uses the Omega Library version 0.91 [35].

Since uninterpreted function symbols are the Omega Test’s solution to non-affine expressions, the functionality of uninterpreted function symbols needs some further explanation. An uninterpreted function symbol is simply a variable with one or more arguments, (e.g., $f(i, j)$), representing a side-effect-free function. The Omega Test accepts affine expressions extended to also contain uninterpreted function symbols, (e.g., $i + 2 * f(i)$). The current implementation

of the Omega Test only allows the loop indices of enclosing loops to be the arguments of uninterpreted function symbols. Two identical uninterpreted function symbols can be cancelled out or combined together if all their arguments are equal, (e.g., $f(i) - f(i') = 0$ if $i = i'$). Because our implementation of our interface to the Omega Test adds no constraints on the values that these uninterpreted function symbols can take, the Omega Test can apply no other kind of simplification on uninterpreted function symbols.

Our interface to the Omega Test handles non-affine expressions by translating them into uninterpreted function symbols. For example, to set up a dependence test between $A(n * i + j)$ and $A(n * i + j + 1)$, the interface would translate the non-affine expression $n * i$ into the uninterpreted function symbol $f(i)$. Thus, the interface would feed the constraint $f(i) + j = f(i') + j' + 1$ to the Omega Test. We also use uninterpreted function symbols to handle loop-variant variables. That is, variables whose values change for some loop but are not loop indices.

3.7.1 Effectiveness

To measure the effectiveness of the Range Test and Omega Test, we counted the number of loops found parallel by these techniques as well as the number of loop-carried dependences eliminated. The results of these measurements is shown in Table 3.2. These results were run on a subset of the Perfect Benchmarks, two National Center of Supercomputing Applications (NCSA) codes, and most of the Fortran codes in the Spec92 benchmarks. We ran some very simple data dependence tests before running either the Range or Omega tests. The most important of these simple tests were the GCD test and a simple test that eliminated dependences between $A(i)$ and $A(i)$ for a loop with index i . We counted an eliminated loop-carried dependence arc multiple times if that arc carried dependences for multiple loops.

<i>Code</i>	<i>Number of lines</i>	<i>Both tests</i>		<i>Range only</i>		<i>Omega only</i>	
		<i>Par. Loops</i>	<i>L.C. Deps.</i>	<i>Par. Loops</i>	<i>L.C. Deps.</i>	<i>Par. Loops</i>	<i>L.C. Deps.</i>
ARC2D	4694	2	216	0	0	0	0
BDNA	4887	29	462	0	0	6	46
FLO52	2368	8	37	1	4	0	0
MDG	1430	13	128	6	23	1	9
OCEAN	3285	136	1725	76	884	0	0
TRFD	634	6	60	6	35	0	21
CLOUD3D	14438	1	200	0	12	0	0
CMHOG	11286	11	17	16	16	0	0
DODUC	5334	2	68	0	0	0	0
FPPPP	2718	4	176	0	0	0	0
HYDRO2D	4461	3	6	3	6	0	0
MDLJDP2	4136	0	9	0	6	1	3
MDLJSP2	3885	0	9	0	6	1	3
NASA7	1204	17	108	0	39	0	2
ORA	453	2	51	0	0	0	9
SU2COR	2514	44	1059	0	0	2	568
SWM256	487	12	24	0	0	0	0
TOMCATV	195	3	13	0	0	0	5
WAVE5	7628	124	864	15	174	0	2

Table 3.2: Number of parallel loops or eliminated loop-carried dependences detected by the Range Test and the omega test.

We broke our results into three categories. The *Both tests* category displays the number of loops that were found parallel and the number of loop-carried dependences eliminated by both the Range and Omega tests. The *Range only* category displays the number of loops found parallel and loop-carried dependences eliminated by the Range Test but not the Omega Test. Similarly, the *Omega only* category displays the number of loops found parallel and loop-carried dependences eliminated by the Omega Test but not the Range Test.

All of the advanced restructuring techniques developed and implemented in Polaris were used before dependence testing. These techniques include partial inlining, interprocedural symbolic constant propagation with procedure cloning, array privatization, generalized induction variable substitution, and reduction recognition. Because of this, dependence arcs from reductions, induction variables, and private arrays and scalars have already been eliminated when the

Range and Omega Tests were executed. Details of these advanced techniques can be found in [9, 7].

From Table 3.2, we can see that there are cases where the Range Test does better, and cases where the Omega Test does better. This should not be surprising, because the Omega Test has difficulties with non-affine expressions while the Range Test was designed to handle such cases. On the other hand, the Omega test is exact for affine expressions while the Range Test is not.

To get a better understanding why one test was more successful than the other for some cases, we examined every loop-carried dependence eliminated by only the Range Test or only the Omega Test. For the Range Test, almost every dependence arc that only it eliminated were dependences between non-affine array accesses. Most of these cases were due to the linearization of arrays from partial inlining or from induction variable substitution. The only cases where additional dependence arcs eliminated by the Range Test were not from non-affine array accesses were all the additional loop-carried dependences eliminated for WAVE5. For these cases, the Range Test used the constraint $1252 \leq n \leq 50080$, which was generated by Range Propagation from a conditional statement just before the loop, to break dependences between accesses pairs such as $A(i)$ and $A(i + n)$ or $A(i + n)$ and $A(i + 51332)$, where $1 \leq i \leq 1252$

The Omega Test sometimes did better than the Range Test for several reasons. The most common reason is coupled subscripts. Coupled subscripts are dependences between multidimensional array accesses where one can disprove dependences when one examines all the dimensions together, but can't disprove dependences by testing each dimension, one by one. Almost all of the coupled subscripts that we've seen were one of the access pairs below:

- Between $A(i, j)$ and $A(j, j)$ where $i < j$,
- Between $A(i, j)$ and $A(j, i)$ where $i < j$,

- Between $A(i, i + j)$ and $A(i + j, i)$,
- Between $A(i, c)$ and $A(c, i)$ where c is an integer.

Other coupled subscripts were simple variants of the above four types. Coupled subscripts accounted for about half of the additional loop-carried dependences eliminated by only the Omega Test for codes BDNA and SU2COR, and all of the additional loop-carried dependences for codes MDG, NASA7, and TOMCATV.

The other case where the Omega Test sometimes did better than the Range Test were for those cases where the ranges of two array accesses overlapped, but this overlap is solely due to a dependence between the two accesses for the same loop iteration. This case occurred for all the additional loop-carried dependences eliminated by only the Omega Test for ORA, MDLJDP2, and MDLJSP2. One example, taken from ORA, is a dependence between $A(i + 400)$ and $A(i + 50 * j + 300)$, where $1 \leq i \leq 19$. A dependence exists between these two accesses only when $j = 2$ and $i = i'$, where i' is the value of the i index for the second access. Another example, taken from MDLJDP2 and MDLJSP2, is a dependence between $A(l)$ and $A(4 * f(i, j, k) + l)$, where $1 \leq l \leq 4$, and $f(i, j)$ is actually a complicated non-affine expression. Now the Range Test is powerful to analyze this non-affine expression, which is $f(i, j, k) = n^2 * i + n * j + k - n^2 - n$, to determine that $f(i, j, k) \geq 0$. However, it is not smart enough to see that the coefficient 4 on the term $4 * f(i, j, k)$ guarantees that there can only be a dependence between iteration l' and l'' of the loop with index l if and only if $l' = l''$, since $1 \leq l \leq 4$. On the other hand, the Omega Test can prove this, even though it does not know the values that f can take.

Another reason for why the Omega Test sometimes did better than the Range Test was that our interface to the Omega Test was able to replace complex loop-invariant expressions with symbolic constants but our interface to the Range Test did not. For example, for the code

BDNA, the dependence tests need to disprove dependences between $A(i)$ and $A(i + 2 * x(1))$, where $1 \leq i \leq x(1)$. Our interface to the Omega Test replaces the loop-invariant $x(1)$ term with a symbolic constant t , getting $A(i + 2 * t)$, before feeding it to the Omega Test. Since these transformed accesses are affine, the Omega Test disproves the dependence between the two accesses. However, the Range Test receives these accesses in their raw form. Since the current implementation of Range Propagation cannot determine constraints on the $x(1)$ term, it cannot prove that $i \leq x(1) < 1 + 2 * x(1) \leq i + 2 * x(1)$ since it doesn't know that $x(1)$ is always positive. Thus, the Range Test fails for this pair of accesses. However, if some pre-processing pass was written to replace all loop-invariant expressions with symbolic constants, (e.g., replace all $x(1)$ s with t), the Range Test would also succeed. Because of this, we do not consider this case to be a shortcoming to the Range Test, just its interface. This case occurred for half of additional loop-carried dependences eliminated by only the Omega Test for BDNA and SU2COR. (The other half were coupled subscripts.)

The final reason why the Omega Test sometimes did better than the Range Test for our measurements was because the Omega test can always break cross-iteration dependences for loops with only one iteration while the Range Test cannot. Being that there is no benefit in parallelizing single iteration loops, we do not consider this to be a weakness of the Range Test. This case occurred for the cross-iteration dependences eliminated only by the Omega Test for TRFD and WAVE5.

Overall, the Range Test was able to determine that more loops were parallel than the Omega Test. Additionally, for the Perfect Benchmarks and the NCSA codes, most of the loops that were identified as parallel by only the Range Test were loops that take up a significant fraction of the program's execution time while the loops identified only as parallel by the Omega Test

<i>Code</i>	<i>Preprocessing</i>	<i>Range test</i>	<i>Omega test</i>	<i>Rest of dd testing</i>
ARC2D	300	20	49	160
BDNA	620	41	110	280
FLO52	250	17	18	83
MDG	200	43	74	140
OCEAN	930	400	300	100
TRFD	170	46	69	24
CLOUD3D	735	52	140	760
CMHOG	1200	180	250	3800
DODUC	600	8	39	450
FPPPP	1000	7	1800	1000
HYDRO2D	100	0.8	3	15
MDLJDP2	110	3	8	18
MDLJSP2	110	3	10	18
NASA7	110	31	33	100
ORA	13	0.9	0.3	2
SU2COR	1200	350	4800	2700
SWM256	23	0.7	0.4	5
TOMCATV	18	0.6	0.3	5
WAVE5	1300	48	71	540

Table 3.3: Time taken by the Range and Omega tests on real programs. Timings for the rest of dependence testing as well as all the analyses performed before dependence testing are also included.

were all insignificant. (We were unable to determine the importance of the loops in the SPEC benchmarks). Thus, we believe that the Range Test would have a greater impact than the Omega Test in identifying significant amounts of loop-level parallelism in real programs.

3.7.2 Speed

One of the biggest arguments made by compilers developers against symbolic data dependence tests such as the Range Test is that they are too slow. Although the Range Test is slower than other dependence tests such as the GCD or Banerjee’s Inequalities test, since it manipulates symbolic expressions rather than integers, we believe that the Range Test is efficient enough for use in commercial parallelizing compilers.

To support this assertion, we have measured the execution times taken by the Range and Omega tests to perform the dependence testing for the measurements displayed in Table 3.2 and discussed in the previous section. These timings are displayed in Table 3.3. These timings were collected on a Sparc 10. Our measurements were collected from Polaris, which was compiled with g++ 2.6.3 with the -O flag. The columns *Range Test* and *Omega test* show the times taken by the Range and Omega Tests to perform the dependence tests of the previous experiment.

To give the reader an idea of the significance of these timings compared to the rest of the compiler, we also included timings for all the preprocessing performed before dependence testing and timings of the rest of the dependence testing pass. The preprocessing phase includes the time to parse the Fortran codes as well as several restructuring techniques, including the advanced techniques described in the previous subsection. Because some of these advanced techniques may significantly increase code size, (e.g., partial inlining and interprocedural constant propagation with procedure cloning), the time spent applying these advanced techniques and dependence testing may be much greater than other parallelizing compilers. The timings of the rest of dependence testing pass includes timings of the simple dependence tests described in the previous subsection, as well as timings of the functions that determine and iterate over all possible dependences that need to be tested in a program unit, that create a dependence graph, and that identify parallel loops.

From Table 3.3, one can see that although the Range Test does take a significant amount of time in a parallelizing compiler, it does not dominate that execution time. In the worst case, it only took about a third of the execution time of Polaris. In the average case, it took much less. Additionally, it was on average about twice as fast as the Omega Test. In a few cases, it

was much faster. Thus, we feel confident in claiming that the Range Test is efficient enough to be incorporated in commercial parallelizing compilers.

3.8 Related work

The Range Test was developed, independent of other dependence tests, to handle the symbolic array subscripts we encountered in actual programs. Early ideas of such a test were described in [23, 34]. The most distinguished feature of the test may be the fact that it is now available in an actual compiler, which has proven to parallelize important programs to an unprecedented degree.

The following discussion compares our test to one of the most effective state-of-the-art tests and points out related ideas of other projects.

Mathematically, the Range Test can be thought of as an extension of a symbolic version of the Triangular Banerjee’s Inequalities test with dependence direction vectors [3, 49], although our implementation differs. The only drawback of our test, compared to the Triangular Banerjee’s test with directions, is that it cannot test arbitrary direction vectors, particularly those containing more than one ‘<’ or ‘>’ (e.g., $(<, <)$). The permutation of loop indices partially overcomes this drawback. (These permutations can be thought of as permutations of the dependence direction vectors tested.) We have found that this limited set of direction vectors, along with the permutation of loop indices, was sufficient to parallelize all of the relevant loop nests in our test suite. One advantage of the Range Test is that the worst case of the number of direction vectors tested is better than Banerjee’s Inequalities with directions, since we test at most $O(n^2)$ direction vectors while Banerjee’s Inequalities with directions may test as many as $O(3^n)$ direction vectors.

Haghighat and Polychronopoulos presented a dependence test to handle nonlinear, symbolic expressions [26]. Their algorithm is essentially a symbolic version of Banerjee’s Inequalities test. However, their test did not include the extensions to Banerjee’s Inequalities to test dependence direction vectors and to handle triangular loops, nor does it include our extension to handle nonlinear expressions containing i^c terms, as in Figure 3.7 after induction variable substitution, where i is a loop index and c is an integer constant greater than 1. We have seen several examples in the Perfect Benchmarks that need all these extensions to be identified as parallel. The same authors presented ideas to calculate the set of variable constraints holding for each statement of the program unit, then to use these constraints to prove or disprove symbolic inequalities for dependence testing. More specifically, they use an algorithm by Cousot and Halbwachs [17] to compute the set of variable constraints. We also determine variable constraints and perform symbolic inequality tests, although we use different techniques, (i.e., Range Propagation). We will compare these two methods to compute variable constraints later in Chapter 4.

In a separate paper, Haghighat and Polychronopoulos [27] describe a technique to prove that a symbolic expression is strictly increasing or decreasing. By using this technique, self-dependences for an array reference can be eliminated. Their example can prove that all the loops in Figure 3.7, after induction variable substitution, are parallel. However, as described, the test only handles self-dependences. The subroutine OLDA in *TRFD* has other important loop nests that has multiple array accesses with nonlinear subscript expressions similar to the subscripts from Figure 3.7.

Maslov [36] presents an alternate way to handle symbolic, non-linear expressions. Instead of testing these expressions directly, his algorithm partitions the expression into several independent subexpressions, then tests these partitions using conventional data dependence tests. Es-

essentially, it delinearizes array references. For example, it converts an array reference $A(n*i+j)$, where $1 \leq j \leq n$, into a two-dimensional array $A(j, i)$. The greatest strength of this technique is that it can convert non-linear expressions into linear ones, allowing exact data tests like the Omega Test [42] to be applied. Because of this, there are situations where Maslov's algorithm proves independence whereas we cannot, such as the array references $A(n*i+j)$ and $A(i+n*j)$, where $1 \leq i \leq j \leq n$. However, the delinearization algorithm cannot handle expressions containing terms of the form i^c , as in Figure 3.7 after induction variable substitution. Furthermore, the algorithm requires some additional symbolic capabilities; the compiler must be able to calculate symbolic gcd's and modulus, and the compiler must be able to sort the set of symbolic coefficients (c_j 's). Performing this symbolic sort can be particularly difficult, since one may be unable to determine that some of the coefficients are less than others (i.e., the c_j 's may not have a total ordering).

3.9 Conclusions

We have developed a symbolic data dependence test, called the Range Test, that can identify parallel loops in the presence of nonlinear array subscripts and loop bounds. We have shown that the Range Test can prove that two very important loop nests in the Perfect Benchmarks are parallel, whereas conventional data dependence tests cannot. In our experiments, we have found that the Range Test can prove independence for many of the other parallel loops that contain symbolic non-linear array subscript expressions.

We have implemented the Range Test in Polaris. Currently, the Range Test is the only major data dependence test implemented in Polaris. To determine its effectiveness, we have compared its ability to eliminate loop-carried data dependences and identify parallel loops with

the Omega Test, an exact data-dependence test for linear array subscripts. We have found that the Range Test identifies a significant number of parallel loop nests that the Omega Test could not. Two conclusions can be derived from this result. First, there are a significant number of important parallel loops that contain nonlinear expressions in real programs. Second, the Range Test can determine that some, if not all, of these loops are parallel.

With the aid of memoization [37], or the caching of already tested array subscript pairs, we believe that the execution time of the Range Test is acceptable, even when applied as the only test. Our timings of the Range Test support this belief. However, there is no reason why the Range Test cannot be invoked only when other dependence tests fail due to nonlinear expressions. By doing so, one would gain the power of the Range Test at little cost.

To implement the Range Test, one needs a facility to compare arbitrary expressions. The next chapter will describe Range Propagation, which provides such a facility.

Chapter 4

SYMBOLIC RANGE PROPAGATION

4.1 Benefits from an expression comparator

One of most useful of symbolic analysis techniques is the ability to compare arbitrary symbolic expressions, using constraint information derived from the program. Many transformations in a parallelizing compiler can benefit from such an ability. Examples are symbolic dependence testing, detection of zero-trip loops, dead-code elimination, determination of array sections referenced by an array access, and loop trip-count estimation. We will examine two of these examples, symbolic data dependence testing and detection of zero-trip loops, in detail.

The ability to compare symbolic expressions is probably most useful for symbolic data dependence tests. In fact, our nonlinear data dependence test called the Range Test, which was described in Chapter 3, is built upon this ability.

The Range Test proves that two array accesses do not have a loop-carried dependence by proving that the range of possible values referenced by one access does not overlap with the range of possible values for the other access. It proves this by determining that the symbolic upper bound of one of these ranges is less than the symbolic lower bound of the other range. This requires the ability to compare symbolic expressions.

The real strength of the Range Test is that it can prove independence for non-affine (non-linear) array references, which most other dependence tests cannot do. For example, the Range Test can prove that all the loops in Figure 4.1 are parallel but, to our knowledge, no other dependence test can do so. This loop nest was taken from TRFD, a code in the Perfect Benchmarks, and accounts for 28% of the code's serial execution time. For the Range Test to prove that there are no dependences for the writes to array A , it needs to compare non-affine symbolic range bounds derived from statements $S1$ and $S2$, under the constraints imposed by the enclosing loops, (e.g., $0 \leq i \leq n - 1$, $n \geq 1$). For example, the Range Test had to compare $((i + 1) * (n^2 + n)) / 2 + i + k + 2$ to 0 for this example. We have seen several other important loop nests from real applications that needed the Range Test with Range Propagation to effectively parallelize them.

The ability to compare symbolic expressions is also needed to prove that a loop is not a zero-trip loop. By knowing that a loop is not a zero-trip loop, one can peel off an iteration of a loop or generate a last-value assignment for an induction variable eliminated by induction variable substitution without enclosing this iteration or last-value assignment with a conditional statement that tests whether the loop was a zero-trip loop. Eliminating this conditional is desirable because conditional statements often weaken the results of global analyses, (e.g., constant propagation and induction variable substitution). For example, both loop peeling

```

DO i = 0, n - 1
  DO j = 0, i
    DO l = 0, i - j
S1 :      x = ((i2 + i + 2 * j) * (n2 + n + 2)) / 4 + l + 1
          A(x) = ...
    ENDDO
  DO k = 0, n - i - 2
    DO l = 0, k + i + 1
S2 :      x = ((i2 + i + 2 * j) * (n2 + n + 2)) / 4
          + i - j + k * (i + 1) + (k2 + k) / 2 + l + 2
          A(x) = ...
    ENDDO
  ENDDO
ENDDO

```

Figure 4.1: A loop nest, extracted from TRFD, that contains non-affine array references. Loop peeling and induction variable substitution were performed to place the loop nest into this form.

and induction variable substitution was needed to get the loop nest in Figure 4.1 into the form shown. Without the ability to detect zero-trip loops, these transformations would have inserted conditional statements that would have prevented the outermost loops from being parallelized.

In response to this need for comparing arbitrary symbolic expressions under constraints derived from the program unit, we have developed Range Propagation. Range Propagation centers upon the computation and manipulation of *ranges*. A range, denoted $[a : b]$, consists of a symbolic lower bound a and a symbolic upper bound b , where $a \leq b$. Constraints are represented by a mapping from program variables to ranges. For example, assigning the range $[a : b]$ to a variable x denotes the constraint $a \leq x \leq b$. We will always assume that x does not occur in either a or b . This mapping from variables to ranges will be called a *range dictionary*.

Range Propagation consists of two major parts: an expression comparison algorithm and a range propagation algorithm. The expression comparison algorithm determines which inequality relationships (e.g., $<$, $=$, \geq) hold between two expressions, given a set of constraints in a

range dictionary. The range propagation algorithm determines the constraints that hold at each point in a program. It does this by computing the ranges that initially hold for each variable, then propagating these ranges through the program unit. Since the range propagation algorithm needs to compare symbolic expressions to simplify ranges, we will discuss the expression comparison algorithm first.

4.2 Comparing expressions using symbolic ranges

In this section, we will first give an overall sketch of our expression comparison algorithm, which determines the relationship between two arbitrary symbolic expressions under the constraints given by a range dictionary. We will then discuss how two major components of the algorithm are implemented. A cost analysis of this algorithm will then be performed.

4.2.1 Algorithm

This algorithm assumes that the compiler can manipulate and simplify arbitrary symbolic expressions. Efficient simplification techniques for parallelizing compilers can be found in [26, 28, 32].

Roughly, we determine the inequality relationship between two symbolic expressions p and q , in respect to a range dictionary R , by forming the difference d between p and q , then repeatedly replacing (substituting) each variable x in d with the range of x , until we reach a point where the lower bound of d is a non-negative integer or the upper bound of d is a non-positive integer. We can then determine the relationship between p and q from the bounds of this final value of d . For example, $p > q$ if the lower bound of the final value of d is a positive, non-zero integer.

The expression comparison algorithm, which implements the intuitive description above, is shown in Figure 4.2. Statement *S1* determines the difference d between p and q . (It is assumed that d has been fully simplified.) Statement *S2* determines the best order to replace variables in d with their ranges. This order is stored in an ordered list S , which contains all variables that may eventually need to be replaced in d , including variables that are not initially in d . Statement *S3* repeatedly removes a variable x from the list S and replaces x in d with its range $R(x)$, until list S is empty or d is comparable. (We say that a range d is comparable when its lower bound is a positive integer or its upper bound is a negative integer.) Statement *S4* handles the case where the while loop ran out of variables to replace before it could make d be comparable. This loop simply replaces all remaining variables in d with the unconstrained range $[-\infty : +\infty]$. (We do this rather than failing because there are cases where d would be comparable if the variables were eliminated, (e.g., $d = [\max(1, x) : +\infty]$.) Statement *S5* returns the inequality relationship between p and q by examining the lower and upper bounds of d . If it cannot determine any relationships, it returns the *unknown* inequality relationship.

For example, suppose we wish to compare $x * y + 1$ with y , where $x = [y : 10]$ and $y = [1 : \infty]$. First, we calculate the difference, which is $d = x * y - y + 1$. Then, we replace x with $[y : 10]$, getting $d = [y^2 - y + 1 : 9 * y + 1]$, (we'll describe how we computed this new value for d in the next subsection). Since d is still not comparable, we replace y with $[1 : \infty]$, getting $d = [1 : \infty]$. Since the lower bound of d is greater than zero, we have the relationship $x * y + 1 > y$.

To implement the algorithm in Figure 4.2, one must implement the functions: `replacement_order()` and `replace_var()`. The function `replacement_order()` attempts to determine the best order to replace variables with their ranges, while the function

INPUT: Symbolic expressions p and q and range dictionary R ,
 which maps each variable x to a range $R(x) = [a : b]$.
OUTPUT: Inequality relationship between p and q
 (i.e, $<$, \leq , $=$, $>$, \geq , or $?$ (unknown)).

```

S1 :  $d \leftarrow [q - p : q - p]$ 
S2 :  $S \leftarrow \text{replacement\_order}(d, R)$ 
S3 : while ( $d$  is not comparable and  $S$  is not empty) do
     $x \leftarrow S.\text{head}$ 
     $S \leftarrow S.\text{tail}$ 
     $d \leftarrow \text{replace\_var}(d, x, R(x), R)$ 
end while
if ( $d$  is not comparable) then
S4 : foreach variable  $x$  in  $d$  do
     $d \leftarrow \text{replace\_var}(d, x, [-\infty : +\infty], R)$ 
end foreach
endif
S5 : return inequality relationship for  $d$ 
  
```

Figure 4.2: The symbolic expression comparison algorithm.

`replace_var()` performs the actual replacements. The next two subsections will describe these functions in detail.

4.2.2 Replacing a variable with its range

One of the essential components of the expression comparison algorithm is `replace_var()`, which replaces variables with their ranges. There are several methods that such a replacement can be performed. In this section, we will describe two methods: a faster but less accurate algorithm, which substitutes each occurrence of the variable with its range, and a slower but more accurate algorithm, which determines the new range of the expression from its monotonicity.

4.2.2.1 Range substitution method

The most intuitive method to replace a variable with its range is to physically substitute each occurrence of the variable in the given expression with the variable's range, then simplify the

$$[[a : b] : c] \Rightarrow [a : c] \quad (4.1)$$

$$[a : [b : c]] \Rightarrow [a : c] \quad (4.2)$$

$$[a : b] + c \Rightarrow [a + c : b + c] \quad (4.3)$$

$$[a : b] * c \Rightarrow \begin{cases} [a * c : b * c] & \text{if } c \geq 0 \\ [b * c : a * c] & \text{if } c \leq 0 \\ [-\infty : \infty] & \text{otherwise} \end{cases} \quad (4.4)$$

$$[a : b] / c \Rightarrow \begin{cases} [a/c : b/c] & \text{if } c > 0 \\ [b/c : a/c] & \text{if } c < 0 \\ [-\infty : \infty] & \text{otherwise} \end{cases} \quad (4.5)$$

$$[a : b]^i \Rightarrow \begin{cases} [a^i : b^i] & \text{if } i \text{ is even and } a \geq 0 \\ [b^i : a^i] & \text{if } i \text{ is even and } b \leq 0 \\ [a^i : b^i] & \text{if } i \text{ is odd} \\ [-\infty : \infty] & \text{otherwise} \end{cases} \quad (4.6)$$

$$\min([a : b], c) \Rightarrow [\min(a, c) : \min(b, c)] \quad (4.7)$$

$$\max([a : b], c) \Rightarrow [\max(a, c) : \max(b, c)] \quad (4.8)$$

Table 4.1: Rewrite rules for simplifying expressions containing ranges. Variables a , b , and c are symbolic integer-valued expressions. The variable i is an integer.

resulting expression. This simplification transforms the resulting expression into a range whose bounds do not contain ranges. As an example, suppose we wish to replace the range $[1 : y]$ for variable x in the difference range $d = [x^2 - x : \infty]$. The resulting range can be computed by first substituting $[1 : y]$ for x in d , getting $[[1 : y]^2 - [1 : y] : \infty]$, then simplifying this range down to $[1 - y : \infty]$, (assuming that $y \geq 1$).

By far the most complicated part of this method is the simplification of the substituted expression. We simplify an expression by applying rewrite rules to each subexpression, from the innermost subexpressions outward. These rewrite rules transform a subexpression whose arguments are range expressions into a range subexpression whose arguments do not contain ranges. These rewrite rules are displayed in Table 4.1. All these rewrite rules assume that we are working with integer-valued symbolic expressions and that all ranges' lower bounds are less than or equal to their upper bounds. The rest of this chapter will also make these assumptions.

As an example, consider the simplification of $[[1 : y]^2 - [1 : y] : \infty]$, taken from the previous example. The steps to simplify this expression are:

$$\begin{aligned}
[[1 : y]^2 - [1 : y] : \infty] &\Rightarrow [[1^2 : y^2] - [1 : y] : \infty] && \text{by rule (6)} \\
&\Rightarrow [[1^2 : y^2] + [-y : -1] : \infty] && \text{by rule (4)} \\
&\Rightarrow [[1^2 - y : y^2 - 1] : \infty] && \text{by rule (3)} \\
&\Rightarrow [1^2 - y : \infty] && \text{by rule (1)} \\
&\Rightarrow [1 - y : \infty]
\end{aligned}$$

Several of the rewrite rules in Table 4.1 require the comparison of symbolic expressions to zero. For example, to simplify expression $x * [1 : 10]$, one must determine the sign of x . We handle this by recursively calling the expression comparison algorithm in Figure 4.2. The costs of performing these recursive expression comparisons can be greatly decreased by caching and reusing the signs of variables, (i.e., memoization).

Because this method makes recursive calls to the expression comparison algorithm, it could go into an infinite loop. To guarantee termination, we do not allow the range for variable x to be substituted in any recursive calls to the expression comparison algorithm while the expression comparison algorithm is trying to replace x in some expression. Instead, we substitute the unconstrained range $[-\infty : \infty]$ for x . This prevents any more recursive expression comparisons, since the simplification of ranges containing $[-\infty : \infty]$ does not require them. (We did not include the rewrite rules for expressions containing $[-\infty : \infty]$, since they are easy to determine.) So, if there are v integer variables in a program, there can be at most v invocations of the expression comparison algorithm on the stack at one time.

It is often desirable to have MIN and MAX expressions in the range expressions for variables. To uncover more opportunities for simplification, these MIN and MAX subexpressions should

$$\min(\min(a, b), c) \Rightarrow \min(a, b, c) \quad (4.9)$$

$$\min(a, b) + c \Rightarrow \min(a + c, b + c) \quad (4.10)$$

$$\min(a, b) * c \Rightarrow \begin{cases} \min(a * c, b * c) & \text{if } c \geq 0 \\ \max(a * c, b * c) & \text{if } c \leq 0 \end{cases} \quad (4.11)$$

$$\min(a, b) / c \Rightarrow \begin{cases} \min(a / c, b / c) & \text{if } c > 0 \\ \max(a / c, b / c) & \text{if } c < 0 \end{cases} \quad (4.12)$$

$$\min(a, b)^i \Rightarrow \begin{cases} \min(a^i, b^i) & \text{if } i \text{ is even and } \min(a, b) \geq 0 \\ \max(a^i, b^i) & \text{if } i \text{ is even and } \min(a, b) \leq 0 \\ \min(a^i, b^i) & \text{if } i \text{ is odd} \end{cases} \quad (4.13)$$

Table 4.2: Rewrite rules for simplifying min expressions. The rewrite rules for max expressions is similar. Variables a , b , and c are assumed to be arbitrary symbolic integer expressions. Variable i is an integer.

be moved outward so that they enclose the entire expression, (although they shouldn't be pulled out of ranges). Thus, when we substitute a range containing MINs and MAXs into an expression, we must pull these MINs and MAXs outward to the top levels of the expression. This can be done by using rewrite rules similar to those given earlier for simplifying range expressions. These rules are displayed in Table 4.2.

In addition to the range, MIN, or MAX specific simplifications given earlier, we apply conventional symbolic simplification techniques to the expression. These techniques include constant folding, distribution of products-of-sums, and the combination and cancellation of common symbolic terms [14, 26, 28, 32]. We also use advanced techniques developed by Haghighat [28] to simplify expressions containing integer divisions. Where they fail, we multiply out the divisions in the difference range, being careful to include the truncation errors of the divisions. For example, we convert the expression $d = a/b + c$ to the range $d' \sim d * b$ where $d' = [a - b + 1 + b * c : a + b * c]$, assuming that $a > 0$ and $b > 0$. (One can see that this range is correct, since $b * (a/b) = b * ((a - a \bmod b) / b) = a - a \bmod b = a - [0 : b - 1] = [a - b + 1 : a]$.)

4.2.2.2 Monotonicity replacement method

One problem with the range substitution method from the previous subsection is that it can generate overly conservative lower or upper bounds when it replaces a variable in an expression that has multiple occurrences of that variable. This occurs because both bounds of the range of the variable to be replaced are used to compute the lower or upper bound of the resulting range expression, although it is impossible for a variable to be equal to both its lower and upper bounds, (assuming that its lower bound does not equal its upper bound). For example, when $[1 : y]$ was substituted for x in $x^2 - x$ in the example from the previous subsection, the final lower bound of the resulting expression $(1 - y)$ was formed by taking the lower bound of $[1 : y]$ for the x^2 term while taking the upper bound of $[1 : y]$ for the $-x$ term. Since there is no legal value of x in the range $[1 : y]$ where $x^2 - x$ would take on the value $1 - y$, this bound is overly conservative.

In this subsection, we will present a variable replacement method that can determine the exact lower and upper bounds for an expression as a whole, (e.g., the lower and upper bounds of $x^2 - x$), if it succeeds. (If it fails, we can still use the variable substitution method.) We will assume that we are replacing variable x in expression $f(x)$ with range $[a : b]$.

The exact lower and upper bounds for $f(x)$ after replacing x are the minimum and maximum values that $f(x)$ can take for any value of x in the range $[a : b]$. Determining these minimum and maximum values are simple if we can prove that $f(x)$ is *monotonic* for x . The expression f is *monotonically non-decreasing* for x within range $[a : b]$, if $f(x) \geq f(x')$ whenever $x \geq x'$ for $x, x' \in [a : b]$. Similarly, f is *monotonically non-increasing* for x within range $[a : b]$, if $f(x) \leq f(x')$ whenever $x \geq x'$ for $x, x' \in [a : b]$. Now, if $f(x)$ is monotonically non-decreasing

for x , then the range of values taken by $f(x)$ is $[f(a) : f(b)]$. Similarly, if $f(x)$ is monotonically non-increasing for x , then the range of values taken by $f(x)$ is $[f(b) : f(a)]$.

Determining whether $f(x)$ is monotonically non-decreasing or monotonically non-increasing is not difficult. One can prove that $f(x)$ is monotonically non-decreasing for x by proving that $f(x + 1) - f(x) \geq 0$. This inequality can be easily proven by recursively calling the expression comparison algorithm in Figure 4.2. Similarly, one can prove that $f(x)$ is monotonically non-increasing for x by proving that $f(x + 1) - f(x) \leq 0$.

The monotonicity replacement method, when used by itself, can be very expensive. (A back-of-the-envelope calculation estimated it to be at least $O(v!)$, where v is the number of variables in the program.) Because of this, we do not use the monotonicity replacement method to determine the sign of the forward difference expression $f(x + 1) - f(x)$. Instead, we use the range substitution method given in the previous subsection. This greatly improves the worst case performance of this technique while preserving much of its accuracy. This also guarantees that this method will eventually halt, since the range substitution method eventually halts. Detailed analysis of its time complexity will be done later in this section.

As an example, suppose we wish to replace the range $[1 : y]$ for variable x in the difference range $d = [x^2 - x : \infty]$. To calculate the new lower bound of d , we must first determine whether the lower bound is monotonically non-increasing or monotonically non-decreasing for x . This is done by calculating the forward difference $(x + 1)^2 - (x + 1) - (x^2 - x) = 2 * x$, then determining whether $2 * x$ is greater-equal or less-equal to zero by a recursive call to the expression comparison algorithm. Now, $2 * x$ is greater than zero, since $2 * x \Rightarrow 2 * [1 : y] \Rightarrow [2 : 2 * y]$ has a positive lower bound. Hence, $x^2 - x$ must be monotonically non-decreasing and the range of $x^2 - x$ is $[1^2 - 1 : y^2 - y] = [0 : y^2 - y]$. After doing a similar computation for the upper bound of d ,

we determine the new range of d to be $[0 : \infty]$. Note that this lower bound is better than the lower bound $1 - y$ computed from the range substitution method in the previous subsection

One complication in using monotonicity for replacing ranges for variables is that either the range we are replacing with or the given expression may contain MIN or MAX expressions. Replacing a variable in a MIN or MAX expression causes problems because the monotonicity of the arguments of the MIN or MAX may differ. Having MINs or MAXs in the range we are replacing for a variable causes problems because we wish to have all MINs and MAXs in the resulting expression to be the outermost terms so to aid simplification.

With a few additional rules, these MIN or MAX expressions can be easily handled. To replace a range for variable x in $\min(f(x), g(x))$, one simply performs the monotonicity replacement method for x on $f(x)$ to get $f(a)$, the same method for x on $g(x)$ to get $g(b)$, then return the expression $\min(f(a), g(b))$. Pulling MIN and MAX expressions out of an resulting expression is also simple. If $f(x)$ is monotonically non-decreasing, then $f(\min(a, b))$ is transformed into $\min(f(a), f(b))$. Similarly, if $f(x)$ is monotonically non-increasing, then $f(\min(a, b))$ is transformed into $\max(f(a), f(b))$. The rules for handling expressions with MAXs are similar.

As an example, suppose we wish to replace x with $[1 : \min(10, y)]$ in $d = [\max(x^2 - x, -x) : \infty]$. Since the previous example has shown that $x^2 - x$ is monotonically non-decreasing, its lower bound is $1^2 - 1 = 0$. Similarly, since $-x$ is monotonically non-increasing, its lower bound is $-\min(10, y)$, which can be rewritten to $\max(-10, -y)$. Thus, the new value of d is $[\max(0, \max(-10, -y)) : \infty] = [\max(0, -y) : \infty]$.

4.2.3 Determining a replacement order

The order in which one replaces ranges for variables is very important. A poorly chosen replacement order may require more replacements and result in less accurate ranges than a well chosen replacement order. For example, suppose we wish to compute the relationship between x and y , where $x = [1 : y]$ and $y = [1 : \infty]$. If we replace x then replace y in $d = x - y$, we get:

$$\begin{aligned} d &= x - y \\ &= [1 : y] - y \\ &= [1 - y : 0] \end{aligned}$$

and we can determine that $x \leq y$, since the upper bound of d is zero. On the other hand, if we replace y first, we get:

$$\begin{aligned} d &= x - y \\ &= x - [1 : \infty] \\ &= [-\infty : x - 1] \\ &= [-\infty : [1 : y] - 1] \\ &= [-\infty : y - 1] \\ &= [-\infty : [1 : \infty] - 1] \\ &= [-\infty : \infty] \end{aligned}$$

In this case, we would not be able to determine any relationship between x and y .

To determine a good order to replace variables, we create a Range Dependence Graph (RDG), then determine an ordering for its vertices. Each vertex in the RDG represents a variable that may need be replaced at some point to make the difference range (d) to be

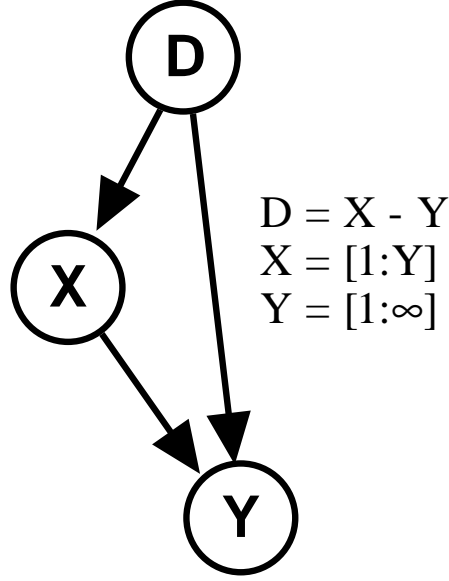


Figure 4.3: An example of a Range Dependence Graph (RDG).

comparable. The RDG also contains a vertex for the initial value of d . An edge exists between vertices x and y if and only if the range for variable x contains the variable y . An example of an RDG is shown in Figure 4.3. RDGs may contain cycles.

A good replacement order is computed by determining the strongly-connected components¹ (SCCs) in the RDG, topologically sorting² these SCCs³. This topological ordering of the SCCs gives an good overall replacement order of the variables between SCCs in the RDG, but does not specify any replacement order for variables within a SCC in respect to each other. To find a replacement order for variables within a SCC, we topologically sort the vertices in each SCC ignoring back-edges.⁴ We use an arbitrary vertex with an edge originating from outside the

¹A strongly-connected component of a graph is a maximal subgraph of that graph where each vertex in the subgraph can reach all other vertices in the subgraph.

²A topological ordering of a directed acyclic graph is an ordering such that if there is a path from vertex u to vertex v then u occurs before v in this ordering.

³By definition of SCCs, one can always topologically sort the SCCs of a graph in respect to each other.

⁴Back-edges are edges in a graph, which if deleted would result in an acyclic graph.

SCC as the root (start) node for the internal topological sort of a SCC. Efficient algorithms for finding SCCs and back-edges, and performing topological sorts can be found in [15].

As an example, we will compute the replacement order for the RDG graph shown in Figure 4.4. The algorithm first computes the strongly connected components of the graph, then topologically sorts them into the order (SCC1, SCC2, SCC3, SCC4). It then topologically sorts the vertices in each SCC, ignoring back-edges. The orders for SCC1 and SCC4 are trivial, $((t)$ and (z) respectively). The order for SCC2 is computed by choosing an arbitrary vertex with an incoming edge, in this case the vertex u , then topologically sorting SCC2 ignoring the back edge $w \rightarrow u$, getting the order (u, v, w) . The ordering for SCC3, which is (x, y) , is computed similarly. The final replacement order is then formed by concatenating these individual orderings in the order of the SCC ordering. This results in the order (t, u, v, w, x, y, z) .

For acyclic RDGs, it can be seen that the computed replacement order is optimal in the number of replacements. However, it may not be optimal for cyclic RDGs. This may be because we chose the wrong root node to start the topological sort in the SCC, or may be that the optimal replacement order may require a variable to be visited more than once. To partially overcome this problem, we repeat the replacement order of each multiple vertex SCC in the final replacement order. The SCC's variable order and its duplicate variable order are adjacent to each other in the final replacement order. For example, the replacement order for Figure 4.4 would be $(t, u, v, w, u, v, w, x, y, x, y, z)$. We have found this heuristic to be effective for many of the cyclic RDGs we have seen in practice.

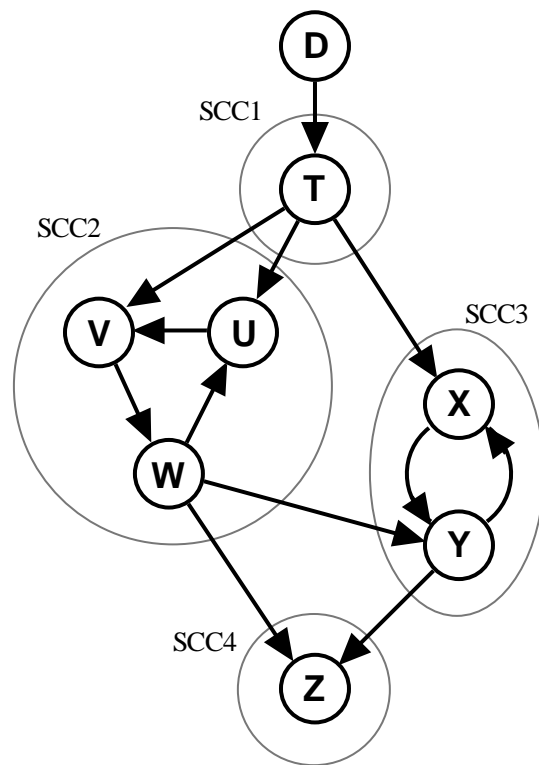


Figure 4.4: An example of a cyclic Range Dependence Graph.

4.2.4 Time complexity

In our experience, nearly all the time spent by the expression comparison algorithm in Figure 4.2 is performing variable substitutions and expression simplifications inside the `replace_var()` function. Thus, the overall time complexity of the expression comparison algorithm would be a function of the number of variable replacements performed and the costs of substitutions and simplifications.

The cost of a single variable replacement, (i.e., `replace_var`), is equal to the costs of performing a constant number of variable substitutions and expression simplifications, ignoring the costs incurred by recursive calls to the expression comparison algorithm. Now, the cost of performing a variable substitution or expression simplification are functions of the sizes of the expressions involved, (more specifically, the number of nodes in the abstract syntax trees that represents the expressions). Typically these functions are of the order $O(n)$ or $O(n \log n)$, where n is the maximum of the sizes of the initial and final expressions. (See Havlak for a detailed analysis of the costs of simplifying symbolic expressions encountered by parallelizing compilers [32].)

Using these costs of substitutions and simplifications, one can compute the cost of the expression comparison algorithm. Unfortunately, each substitution or simplification can cause a multiplicative growth in the size of the final expression, causing subsequent substitutions and simplifications to run longer. Let c be the maximum amount that an expression may grow after a substitution or simplification. Then after i invocations of `replace_var`, each performing a constant number of substitutions and simplifications, the final expression may be of length $c^i n$. Thus the worst case time complexity of the expression comparison algorithm is $O(\sum_{i=1}^r c^i n \log c^i n)$, where r is the total number of variable replacements performed by the

expression comparison algorithm. After simplifying this down and treating variable n as a constant, the worst case time complexity of the expression comparison algorithm is $O(rc^r)$.

Computing the number of variable replacements, (i.e., variable r), is a bit more complicated, since each variable replacement may make recursive calls to the expression comparison algorithm, which can perform several variable replacements itself. The worst case of number of replacements made by the expression comparison algorithm is $O(v!)$. This is because each call to the expression comparison algorithm performs $O(v)$ variable replacements and each variable replacement may make a constant number of recursive calls to the expression comparison algorithm. Now, because the range substitution method has an optimization to ensure termination, where recursive calls to the expression comparison algorithm cannot replace variables that are currently being replaced, these recursive calls can replace only $v - 1$ of the v variables. Solving this recurrence, one gets $O(v!)$.

Although the worst case of the number of variable replacements is $O(v!)$, if one makes a reasonable assumption, the maximum number of variable replacements can be shown to be $O(v)$ or $O(v^2)$. This assumption is that the only recursive comparisons performed by the range substitution method will be those to determine the uncached signs of variables. We feel justified to make such an assumption because it (nearly) holds for all expressions that are polynomials in a fully simplified, sums-of-products form possibly divided by an integer and enclosed by one or more MINs or MAXs;⁵ nearly all expressions derived from real programs are of this form. Alternately, one may modify the algorithm to guarantee that the assumption holds.

If we use the range substitution method to replacing variables with ranges, no recursive calls to the expression comparison algorithm would be needed if the the signs of all variables

⁵This assumption may not hold for exponentiation terms. (See rewrite rules (6) and (13) from Tables 4.1 and 4.2.) However it does hold for most of the cases we've seen.

in a program is known and stored in the sign cache. In this case, only $O(v)$ replacements would be needed, where v is the number of integer variables in the program, or more accurately, the number of variables in the RDG. However, if none of the variable signs are cached, then the algorithm may perform up to $O(v)$ recursive calls to the expression comparison algorithm, (each performing $O(v)$ replacements), to determine the signs of the $O(v)$ variables. Thus, the expression comparison algorithm may perform at most $r = O(v^2)$ variable replacements.

For the monotonicity replacement method, as many as $O(v)$ recursive calls to the expression comparison algorithm would be performed to determine the monotonicity of the $O(v)$ variables. By design of this method, each of these recursive calls can only invoke the range substitution method. Thus, this method will perform at most $O(v^2)$ variable replacements if all the sign caches were already filled. If none of the sign caches were filled, it would still make at most $O(v^2)$ replacements, since the filling of the sign caches only add $r = O(v^2)$ additional replacements.

In summary, the expression comparison algorithm takes at most $O(rc^r)$ time, where c is the maximum multiplicative growth of the difference expression after a substitution and simplification and r is the number of variable replacements. At most $r = O(v^2)$ replacements are made by the algorithm if one assumes that the range substitution method would only make recursive calls to the expression comparison algorithm to determine the sign of uncached variables, or at most $r = O(v!)$ if one does not.

4.2.5 Performance

The previous subsection derived bounds on the time taken by the expression comparison algorithm. Unfortunately, the worst case bounds are very bad. That is, the algorithm takes at most $O(rc^r)$ time. Luckily, for real programs the number of variable replacements (r) performed is

usually very small and little growth typically occurs in the difference expression, (i.e., c is close to 1). In this subsection, we will present quantitative results that support this assertion.

Table 4.3 displays some statistics on the number of replacements and the average times taken by the expression comparison algorithm in Figure 4.2. These statistics were collected from the expression comparisons performed by the range propagation algorithm described in Section 4.3 and by the Range Test, when run on six of the Perfect Benchmark codes and two application codes from the National Center of Supercomputing Applications. Because most of the time of the algorithm is spent in performing variable replacements, the table breaks down the statistics by number of variable replacements per comparison. This number of variable replacements includes replacements performed by recursive calls to the expression comparison algorithm. The *No. Comparisons* column displays the number of expression comparisons that needed this number of variable replacements. The *Avg. time taken* column shows the average wall-clock time in milliseconds spent by one of these comparisons on a Sparc 10. The *Total* row displays the total number of expression comparisons performed and the average amount of time taken by them.

Two remarks should be made concerning Table 4.3. First, special-case code was used to optimize the case where no replacements are required. Second, the large execution time for comparisons that make 13 variable replacements is mostly due to a costly simplification algorithm for integer divisions. If disabled, the average time decreases to about 250 milliseconds. This does worsen the accuracy of the expression comparator for these cases, however.

Table 4.3 shows that the typical invocation of the expression comparison algorithm is moderately inexpensive. It also shows that the small number of variable replacements performed by a typical comparison is the main reason for its low cost. Thus, the expression comparison

<i>No. replacements per comparison</i>	<i>No. comparisons needing n replacements</i>	<i>Avg. time (in mS) taken per comparison</i>
0	26007	0.2
1	13829	7.7
2	953	14.0
3	295	21.1
4	135	27.6
5	59	46.8
6	7	72.9
7	6	105.0
9	5	150.0
13	8	947.5
Total	41304	3.6

Table 4.3: Average time and number of replacements made by the expression comparison algorithm in Figure 4.2 for six Perfect Benchmarks codes.

algorithm should be inexpensive enough to use in a production compiler, despite its exponential worst-case complexity.

4.3 Propagating ranges

The range propagation algorithm centers on the collection and propagation of ranges through a program unit. Abstract interpretation [16] is used to compute the ranges for variables at each point of a program unit. That is, the algorithm “executes” the program by following its control flow paths, updating the current ranges to reflect the side effects of the statements encountered along these paths, until a fixed point is reached. This section will describe how to compute ranges for FORTRAN 77 programs. Similar algorithms can be designed for other languages.

4.3.1 Basic operations

To compute the ranges for each statement in the program unit, some basic operations to merge or join ranges are required. These basic operations, which are displayed in Table 4.4, are the

$$[a : b] \cup [c : d] \Rightarrow [\min(a, c) : \max(b, d)] \quad (4.14)$$

$$[a : b] \cap [c : d] \Rightarrow [\max(a, c) : \min(b, d)] \quad (4.15)$$

$$[a : b] \nabla [c : d] \Rightarrow [\text{if } a = c \text{ then } a \text{ else } -\infty : \text{if } b = d \text{ then } b \text{ else } \infty] \quad (4.16)$$

$$[a : b] \triangle [c : d] \Rightarrow [\text{if } a \neq -\infty \text{ then } a \text{ else } c : \text{if } b \neq \infty \text{ then } b \text{ else } d] \quad (4.17)$$

Table 4.4: Basic operations used by the range propagation algorithm.

union (\cup), intersection (\cap), widening (∇), and narrowing (\triangle) operations. The union operator merges ranges coming from multiple points in the control flow of the program. The intersection operator adds new constraint information to a range. This new constraint information typically originates from conditional tests or loop bounds. The widening operator, which is essentially an overly conservative union operator, is used at selected points of the program unit to guarantee termination. The narrowing operator is used to regain some of the information lost by the widening operator. Our definitions of the narrowing and widening operators were influenced by the operators given by Bourdoncle [11].

The range propagation algorithm uses a special range, denoted as \top , which represents an undefined value. The union operator, when applied upon \top and a range x , has the following identity.

$$x \cup \top = \top \cup x = x$$

The application of the widening and narrowing operators on \top and x have the same identity. For the intersection operator, we have the following identity.

$$x \cap \top = \top \cap x = \top$$

As mentioned in Section 4.2, the expression comparison algorithm can be very expensive when the variables' ranges contain MINs and MAXs. Because of this, we attempt to eliminate

the new MINs and MAXs introduced by the union and intersection operators in Table 4.4 by using the expression comparison algorithm to determine the MIN (or MAX) argument with the smallest (or greatest) value. Also, since expression comparison is often slowest for range bounds that contain both MINs and MAXs, we disallow MIN expressions to occur in range lower bounds and MAX expressions to occur in range upper bounds. More specifically, if the union operator is unable to eliminate the MIN (or MAX) from its lower (or upper) bound, it replaces the bound with $-\infty$ (or $+\infty$).

4.3.2 Algorithm

The ranges for a program unit are computed in two very-similar phases: a widening phase then a narrowing phase. For each of these phases, mappings of variables to their ranges, (i.e., range dictionaries), are associated with each statement and each control-flow edge in the program unit. For both phases, iterative data-flow analysis [2] is used to compute the final values of these range dictionaries. For the widening phase, the ranges for all variables for all program entry points is initially defined to be $[-\infty : \infty]$. All other statements and control flow-edges are initially assigned a range dictionary whose ranges are undefined (\top). The initial ranges for the narrowing phase are the final ranges computed by the widening phase.

We compute the ranges for a statement or control flow edge as follows. The range dictionary for a statement is the union of the ranges of all the entering control-flow edges for that statement. The range dictionaries for the exiting control-flow edges of a statement are computed by modifying a copy of the statement's range dictionary with the side effects of the statement. The modifications made for each kind of statement is as follows: An assignment statement sets the range for the left-hand-side variable to the range computed from the right-hand-side expression.

A conditional statement intersects the entering range dictionary with ranges derived from the conditional's test. A similar intersection operation is done for the bounds of `DO` loops.

To guarantee that the algorithm eventually reaches a fixed point and halts, a widening operator is applied on the entering range dictionaries of all entry statements of loops in the control-flow graph, when in the widening phase. The arguments to the widening operator are the entering ranges for the current visit to the statement and the entering ranges for the previous visit to the statement. The widening operator is only applied for such nodes on the third and later visits to these nodes, to ensure that the current and previous range dictionaries are fully defined, (i.e., no ranges inside them are \top or were formed by a union with \top).

The narrowing phase is identical to the widening phase, except that its initial ranges are the ranges computed from the widening phase, and the narrowing operator is applied at loop-header nodes instead of the widening operator.

Variable modifications by a statement must also be taken into account when computing the ranges for the statement's exiting control-flow edges. When a statement modifies a variable, we eliminate all occurrences of that variable in the exiting control-flow edges' range dictionaries. We do this because the ranges are in terms of these variables before they are modified, and would thus be incorrect after this statement.

A simple way to perform this is to replace all occurrences of that variable with the variable's range, (i.e., call `replace_var`). However, a more accurate result can be achieved for variable modifications caused by a special class of assignment statement called an *invertible assignment* [17]. Invertible assignments are assignment statements where the variable on the left-hand-side also occurs on the right-hand-side, and where this assignment can be rewritten so that the old value of this variable equals some function of the new value of this variable; that

is, if $x_{new} = f(x_{old})$ then one can determine some inverse f' of f such that $x_{old} = f'(x_{new})$. For example, $x = x + 1$ is an invertible assignment, where the inverse of $x + 1$ equals $x - 1$. For variable modifications caused by invertible assignments, one can replace all occurrences of the variable with the inverse of the assignment's right-hand-side. For example, if $y = [1 : x]$ before the invertible assignment $x = x + 1$, then the new range for y is $[1 : x - 1]$.

4.3.3 Example

Figure 4.5 gives an example of the ranges generated by the range propagation algorithm for a small code fragment. Only the ranges for variable x are shown. We will describe how these ranges were computed for only a few statements in the code fragment. The range after statement S_5 is determined by simply computing the range of $x + 1$, which is $[1 : n] + 1 = [2 : n + 1]$. (Remember that the range of x is not allowed to contain x .) The range inside the **THEN** part of the **IF** statement at S_3 was calculated by intersecting the old range $[1 : 2 * n]$ with the range $[-\infty : n]$, resulting in $[\max(1, -\infty) : \min(2 * n, n)] = [1 : n]$. The range after the **ENDIF** statement at S_8 was calculated by taking the union of the two ranges ($[2 : n + 1]$ and $[2 : 2 * n]$) entering the statement, forming $[\min(2, 2) : \max(n + 1, 2 * n)] = [2 : 2 * n]$. Finally, the range at the top of the loop at S_2 was computed with the help of the widening and narrowing operators. Without a widening operator, the values of the range after S_2 would take on the successive values: $[1 : 1]$, $[1 : 2]$, $[1 : 4]$, \dots , preventing the algorithm from terminating. The widening operator prevents this by assigning x to the conservative value $[1 : 2] \nabla [1 : 4] = [1 : \infty]$. However, one can see that the upper bound of this range is overly conservative, since the upper bound from S_1 (1) and the upper bound from S_9 ($2 * n$) are both less than ∞ . This overly conservative value is corrected in the narrowing phase, where the old value of the range after

$S_1 :$	$x = 1$	$x = [-\infty : \infty]$
$S_2 :$	CONTINUE	$x = [1 : 1]$
$S_3 :$	IF $(x \leq n)$ THEN	$x = [1 : 2 * n]$
$S_4 :$	IF $(A(x) \geq 1.0)$ THEN	$x = [1 : n]$
$S_5 :$	$x = x + 1$	$x = [1 : n]$
$S_6 :$	ELSE	$x = [2 : n + 1]$
$S_7 :$	$x = 2 * x$	$x = [1 : n]$
$S_8 :$	ENDIF	$x = [2 : 2 * n]$
$S_9 :$	GOTO S_2	$x = [2 : 2 * n]$
$S_{10} :$	ENDIF	$x = \top$
		$x = [1 : \infty]$

Figure 4.5: Computation of ranges for an small code segment.

S_2 is narrowed with the range formed by the union of the ranges from S_1 and S_9 , getting $[1 : \infty] \triangle ([1 : 1] \cup [2 : 2 * n]) = [1 : 2 * n]$. (Remember that the narrowing operator replaces all infinite bounds of the old range with the bounds of the new range, but otherwise leaves the range alone.)

4.3.4 Time complexity

The time taken to perform a union or intersection of ranges for a particular statement would be the time c taken to perform a symbolic expression comparison times the number of variables v in the range dictionary, or $O(vc)$. Similarly, the cost for handling variable modifications is also $O(vc)$. The widening operator only cost $O(v)$ time. Thus it takes $O(vc)$ time to compute the range for a particular statement or control flow edge.

Because of the widening operators, each range in the program unit can be modified only a constant number of times. Thus the time taken by the range propagation algorithm is $O((s + e)vc)$, where s is the number of statements in the program and e is the number of control-flow edges in the program. Since $e \geq s$, we can simplify this down to $O(etc)$.

<i>Program</i>	<i>Lines of code</i>	<i>Time taken (in S)</i>
ARC2D	4694	8.2
BDNA	4887	33.4
FLO52	2368	19.4
MDG	1430	7.2
OCEAN	3285	23.1
TRFD	634	38.8
CLOUD3D	14438	130.5
CMHOG	11826	42.2

Table 4.5: Time taken by the range propagation algorithm for six Perfect Benchmarks codes and two NCSA codes.

Table 4.5 displays the speed of the range propagation algorithm on a Sparc 10 for six of the Perfect Benchmarks and two application codes from the National Center of Supercomputer applications (NCSA). The range propagation algorithm was compiled using g++ 2.6.3 with the -O flag. This table shows that the range propagation algorithm is efficient for real codes. The table also shows that there is little correlation between the number of lines in the program and the time taken by range propagation. This is because the algorithm is also sensitive to other factors, such as the number of integer variables, the complexity of the expressions that these variables are assigned to, and the complexity of the program's control flow.

4.4 Related work

The idea for representing program constraints as ranges was first proposed by Harrison [31] for array bounds checking and program verification. In his paper, Harrison describes how one can compute the range of integer values that variables can take in a program unit, using data-flow analysis. Although he does propose simple techniques to handle symbolic ranges, our symbolic analysis techniques are superior. (He restricts the bounds of his symbolic ranges to the form `< variable > + < constant > .`)

Bourdoncle [11] greatly improves the accuracy of the integer range propagation algorithm by Harrison through the use of abstract interpretation [16]. Our use of the narrowing operator was influenced by his algorithm. Bourdoncle’s algorithm is unable to generate symbolic ranges. The accuracy of the ranges generated by his (and Harrison’s) technique can be improved with our monotonicity replacement method in Section 4.2.

Cousot and Halbwachs [17] presents a different method to compute and propagate constraints through a program. In their technique, sets of constraints between variables are represented as a convex polyhedron in the n -space of variable values. Because of this representation, all constraints are restricted to be in the form of affine inequality relationships, (e.g., $5 * x + 2 * y \leq 2$). Abstract interpretation is used to compute the convex polyhedron of variable constraints for each statement and each control flow edge of the program. Two affine symbolic expressions can be compared, in respect to constraints given as a convex polyhedron, by determining what side the convex polyhedron falls on the hyperplane formed by the difference of the two expressions.

Although our range propagation algorithm was heavily influenced by their abstract interpretation algorithm, our representation of variable constraints is quite different. They are more accurate in the computation and propagation of affine variable constraints. However, they cannot handle non-affine variable constraints, such as $a < b * c$. Another strength in our representation is that our analyses can use a sparse data-flow form, such as definition-use chains [2] or Static Single Assignment [18], while theirs cannot. Performing range propagation on such a sparse form can greatly increase its efficiency.

In a complementary paper, Tu and Padua [44] present a symbolic expression comparison and constraint propagation technique, based on an extension of Static Single Assignment (SSA)

form [18]. Expressions are compared by repeatedly substituting variables with their constant symbolic values until the two expressions differ by only an integer constant. The values to substitute are determined by a demand-driven analysis of the program. Phi (ϕ) functions, which are allowed to be substituted in these expressions, are used to represent ranges of values. These methods are potentially more efficient because they are demand-driven and use SSA form.

Our variable replacement and simplification techniques are more powerful than theirs. However, their constraint propagation methods are more efficient since they are demand-driven and use Static Single Assignment form. Because of this, we have incorporated their demand-driven techniques into the range propagation algorithm. Chapter 5 will describe how our demand-driven range propagation algorithm. Additionally, their constraint propagation algorithm is capable of performing flow-sensitive analyses.

4.5 Conclusions

We have presented a powerful, efficient technique to compute the symbolic ranges for a program unit, and have shown how to use these ranges to compare arbitrary, possibly non-affine, symbolic expressions. We have also shown that this technique can benefit several passes in a parallelizing compiler.

We have implemented Range Propagation in Polaris. Range Propagation is currently being used for symbolic data dependence testing by the Range Test, detection of zero-trip loops, computation of symbolic range of values that may possibly be referenced by an array access, and estimation of iteration counts of loops. Range Propagation has enabled the Range Test to effectively parallelize two codes in the Perfect Benchmarks, TRFD and OCEAN, with the aid of other restructuring techniques. Because of this, we were able to achieve speedups close

to the hand-parallelized versions, which were 13 for TRFD and 14 for OCEAN on Cedar, a machine with 32 vector processors, where commercial parallelizing compilers could only achieve a speedup of at most 2. Thus, the ability to compare symbolic expressions, as provided by Range Propagation, can significantly improve the effectiveness of parallelizing compilers.

We also found Range Propagation to be reasonably efficient. A single expression comparison takes only about 3.6 milliseconds, on average. Computing all the ranges for a program unit takes about 7 seconds per a thousand lines of code, an average. In chapter 5, we will present a demand-driven version of Range Propagation which greatly improves the speed of computing ranges for a program.

Chapter 5

DEMAND DRIVEN SYMBOLIC RANGE PROPAGATION

5.1 Motivation for demand-driven analysis

As shown in the previous chapter, Range Propagation is useful for many of the restructuring techniques and analyses of parallelizing compilers. In fact, Range Propagation has been used for data dependence analysis, detecting zero-trip loops for induction variable substitution, loop trip-count estimation, and computing the range of possible values taken by an array access in Polaris. Unfortunately, the compiler usually modifies the program between these techniques, requiring repeated recomputation of the program's ranges. Because of this, a significant fraction of a compiler's execution time can be spent performing range propagation.

Because of these costs, we have developed a demand-driven algorithm for performing range propagation. By demand-driven, we mean an algorithm that computes the range for a particular variable only when that range is requested by the user, as opposed to a conventional data-flow

algorithm that computes all ranges at once. Since many restructuring techniques only use a small subset of the ranges of a program unit, a demand-driven algorithm should greatly reduce the costs of range propagation.

5.2 Notation

A **control-flow graph** (CFG) of a program is a directed graph with a special vertex named *start* and where every vertex is reachable from *start*. Each vertex in the CFG corresponds to a statement in the program. An edge exists between two vertices in the CFG if and only if the statement corresponding to the second vertex may be immediately executed after the statement corresponding to the first vertex. An edge is said to be a **back-edge** if the order of the source of the edge is larger than the order of the sink of the edge, under a depth first ordering of the CFG.¹ Vertex *u* **dominates** vertex *v* if and only if every path from *start* to *v* pass through vertex *u*. Vertex *u* **strictly dominates** vertex *v* if and only if *u* dominates *v* and *u* does not equal *v*. Vertex *u* is the **immediate dominator** of vertex *v* if and only if *u* strictly dominates *v* and there is no vertex *w* such that *u* strictly dominates *w* and *w* strictly dominates *v*. See Aho, Sethi, and Ullman [2] for more details on these definitions.

Similar dominance relationships can be defined for the control-flow edges in the program. For example, a control-flow edge dominates a statement if all paths from *start* to that statement pass through that control-flow edge. In this chapter, we would say that a control-flow edge is an **immediate dominating control-flow edge** (or $\text{icdom}(s)$) of a statement (*s*) if that edge is the immediate dominator of the statement.

¹Our definition of back-edges is wider than the definition of back-edges given by others, (i.e., we define more edges to be back-edges than they do). We have defined back-edges differently so that if one deleted all the back-edges from a graph, the graph is guaranteed to be acyclic, even if the original graph is irreducible.

Our demand-driven range propagation algorithm assumes that programs are in **Static Single Assignment** (SSA) form. A program is in SSA form when every variable within it has at most one defining statement. Programs are translated into SSA form by inserting ϕ -functions and renaming variables. A ϕ -function, denoted as $v \leftarrow \phi(w_1, w_2, \dots, w_n)$, is special assignment to a variable v that is inserted at a join in the control flow where at least two definitions of v reach this join. The ϕ -function has an argument for each entering control-flow edge of this join. The i th argument (w_i) corresponds to the value that the variable assigned by the ϕ -function (v) would take if the control-flow of the program took the i th control-flow edge to reach the join node, (i.e., $v = w_i$). An efficient algorithm to translate programs into SSA form is described in [18]. In this chapter, we will assume that the function $\mathbf{def}(v)$ would return the single statement in the program that defines v .

5.3 Range propagation

Briefly the range propagation algorithm computes the range of each variable at each point of the program. A range is simply a symbolic lower bound and a symbolic upper bound on the values that a variable may take.

Since one cannot always statically compute the exact range that all variables may take in a program, range propagation computes a conservative approximation of the range of a variable. The lower bound of this approximation is always guaranteed to be smaller than or equal to the actual lower bound while the upper bound of this approximation is always guaranteed to be larger than or equal to the actual upper bound.

We break up the problem of computing the ranges of a variable at a particular statement into two sub-problems, the computation of the *control ranges* of the variable and the computation

```

function get_range( $s$  : statement,  $v$  : variable) : range
  if ( $R(s, v)$  has not been defined) then
     $c \leftarrow$  get_control_range( $s, v$ )
     $d \leftarrow$  get_data_range( $v$ )
     $R(s, v) \leftarrow c \cap d$ 
  end if
  return  $R(s, v)$ 
end function

```

Figure 5.1: The demand-driven range propagation algorithm.

of the *data ranges* of the variable. The final range for the variable is simply the intersection of its control and data ranges. The control ranges of a variable are those ranges computed from the constraints imposed by the control flow of the program, such as from **IF** or **DO** statements. The data ranges of a variable are those ranges computed from the assignments to that variable. We compute the control and data ranges separately because control ranges are much cheaper to compute.

The top-level algorithm for the demand-driven range propagation algorithm is shown in Figure 5.1. This algorithm stores its results in a global structure named R , so that future invocations of this algorithm can reuse these results rather than recomputing.² The algorithm simply checks whether the range for the given variable and statement already exists in R , computes and stores the range if it does not, then returns the range. The range is computed by intersecting the data and control ranges returned by `get_control_range` and `get_data_range`. The formal definition of the intersection operator for ranges can be found in Table 4.4. The next two sections will describe how these control and data ranges are computed.

²This storing and reusing values to avoid recomputation is called **memoization** [37, 1].

5.4 Computing control ranges

5.4.1 Needed functionality

Our demand-driven control range propagation algorithm assumes that the immediate dominating control-flow edge is known for each statement and that the program is in SSA form. The function `icdom(s)` will represent the immediate dominating control-flow edge of the statement *s*. An linear-time algorithm for computing dominators has been developed by Harel [30]. Alternatively, one can approximate the dominating control-flow edges from the statement dominators, which must be computed when translating into SSA form.

To compute the control ranges of a program, we will assume that there exists a function `get_local_control_ranges(e, v)`, which computes and returns the range of a given variable *v* at a given control-flow edge *e* computed from the control constraints imposed by the source statement of that edge. For example, `get_local_control_ranges(e, v)` would return $[a : \infty]$, if *e* is the exiting control flow edge for the then case of the statement `IF (V .GE. A) THEN`. If there are no control flow constraints imposed on that variable for that control-flow edge, then the function returns the unconstrained range $([-\infty : \infty])$.

5.4.2 Algorithm

The algorithm for demand-driven control range propagation is shown in figure 5.2. This algorithm is composed of two mutually recursive functions: a function that computes the control range that holds for the entry of a given statement and a function that computes the control range that holds after taking a given control-flow edge. Intuitively, these functions computes the control range of a given variable and statement (or control-flow edge) by intersecting the ranges that hold for the variable for all the dominating control-flow edges of the given state-

```

function get_control_range( $s$  : statement,  $v$  : variable) : range
    if (icdom( $s$ ) is not defined) then
        return  $[-\infty : \infty]$ 
    else
        return get_control_range1(icdom( $s$ ),  $v$ )
    end if
end function

function get_control_range1( $e$  : control-flow edge,  $v$  : variable) : range
    if ( $C(e, v)$  has not been defined) then
         $c \leftarrow$  get_local_control_range( $e, v$ )
         $p \leftarrow$  get_control_range(source( $e$ ),  $v$ )
         $C(e, v) \leftarrow c \cap p$ 
    end if
    return  $C(e, v)$ 
end function

```

Figure 5.2: The demand-driven control range propagation algorithm.

ment (or control-flow edge). The control range of a statement is simply the control range that holds after passing through that statement's immediately dominating control-flow edge, (i.e., $\text{icdom}(s)$). If the statement does not have a immediately dominating control-flow edge, then its result is the unconstrained range ($[-\infty : \infty]$). The control range for a control-flow edge is the control range that is imposed by the edge, (i.e., the result of `get_local_control_range`), intersected with the control range for the source statement of that edge. The result is stored in the data structure C so as to avoid needless recomputation, (i.e., it memoizes).

Although recursive, the functions in Figure 5.2 are guaranteed to terminate. By definition of immediately dominating control-flow edges, the source of $\text{icdom}(s)$ must strictly dominate statement s . Since the graphical representation of the dominator relationship is a tree, the algorithm will eventually reach a statement that does not have a immediately dominating control-flow edge.

5.4.3 Time complexity

The worst case time taken by the algorithm in Figure 5.2 is bounded by $O(c|S|)$, where c is the time taken to perform an intersection, (which equals the time taken to perform a constant number of symbolic expression comparisons), and $|S|$ is the number of statements in the program. However, from the extensive use of memoization, (i.e., storing computed values into C and reusing them), the worst case time taken to compute the range for every variable at every statement is $O(c|S||V|)$, where $|V|$ is the number of scalar variables in the program. Since a non-demand-driven algorithm would also take at least $O(c|S||V|)$ time, (since such an algorithm would have to visit each (statement, variable) pair in the program), the demand-driven algorithm is at most as expensive as a non-demand-driven algorithm, ignoring a constant factor.

5.4.4 Design justification

The algorithm for computing control ranges may prompt the user to ask two questions: Why is the algorithm so conservative, and why does the algorithm use dominating control flow edges as opposed to control dependences?³ In this section, we will answer both of these questions.

Examining the algorithm in figure 5.2, one can see that our algorithm for computing control ranges is very conservative, since it looks at only some of the edges in the program. For example, for the following code fragment:

```
    IF (n > 0) THEN GOTO 100
    ...
    IF (n > 0) THEN GOTO 100
    ...
    STOP
100  CONTINUE
```

³A statement s is **control-dependent** upon another statement t if the execution of statement t may determine whether statement s would be executed or not. A formal definition of control-dependences, as well as how to efficiently compute them, can be found in [18].

the algorithm would not be able to prove that $n > 0$ at the `CONTINUE` statement, since neither of the two exiting control-flow edges of the two *IF* statements dominate the `CONTINUE` statement. We have chosen to look at only dominating control-flow edges for simplicity and efficiency. If instead one tried to compute the control ranges from all control-flow edges, then one would need to perform an iterative data-flow analysis. As we shall see in the following sections, performing iterative analyses on demand is a complex, sometimes expensive procedure. Additionally, for structured programs, the results from our algorithm would be identical to the results of an iterative algorithm, since all statements that determine control flow dominate the statements contained within their bodies. Thus, we believe that our algorithm would be just as accurate as an iterative algorithm for most real programs, since most real programs are sufficiently well structured.

As for the question why we use dominating control-flow edges instead of a control-dependences, we do not use control-dependences since they do not provide the information we need. More specifically, there are some cases where a statement can add a control range to another statement, yet this other statement is not control-dependent upon this statement. For example, in the following code fragment:

```

100  CONTINUE
      IF (i <= n) THEN GOTO 200
      ...
      i = i + 1
      GOTO 100
200  CONTINUE

```

the `CONTINUE` statement labeled 200 is not control-dependent upon the `IF` statement, but the control range of *i* at this `CONTINUE` statement is tightened by the constraint imposed by this `IF`, (i.e., $i > n$).

```

function sparse_icdom( $s$  : statement) : control-flow edge
  if ( $I(s)$  has not been defined) then
     $e \leftarrow \text{icdom}(s)$ 
    if ( $e$  is defined and  $e$  does not add any control ranges) then
       $e \leftarrow \text{sparse\_icdom}(\text{source}(e))$ 
    end if
     $I(s) \leftarrow e$ 
  end if
  return  $I(s)$ 
end function

```

Figure 5.3: Demand-driven algorithm for computing a sparse immediate dominating control-flow edge relationship.

5.4.5 Optimizations

By design of the algorithm, the time taken to compute a single control range is dependent upon the number of control-flow edges that dominate the given statement s . The number of dominating control-flow edges of a statement can be very large ($O(|S|)$). However, only a few of these edges add new constraints, (e.g., edges exiting **IF** or **DO** statements or from **ASSERT** directives). Because of this, our algorithm creates and uses a sparse form of the `icdom` function, where this sparse form returns the most immediate dominating control-flow edge that adds at least one range to one variable.

An algorithm for computing the sparse immediate dominating control-flow edge of a statement is shown in Figure 5.3. This algorithm simply traces back through all the statement's dominating control flow edges, using the `icdom` relationship, until it finds an edge that adds a control range to at least one variable. The global structure I is used to memoize the result of this computation so that the algorithm would not recompute it in future calls.

```

type node_ptr = pointer to d_range_node
type node = structure
    var : variable
    value : range
    old_value : range
    committed : boolean
    prev : set of node_ptr
    next : set of node_ptr
end structure

```

Figure 5.4: Fields of a `node` structure.

5.5 Computing data ranges

The algorithm for computing ranges originating from the program's data flow is much more complex than the algorithm for computing ranges originating from constraints imposed by the control flow. This additional complexity arises from the need to iterate to a fixed point, (i.e., perform data-flow analysis), to compute the data ranges.

5.5.1 Data-flow graph

To allow the algorithm to cleanly and efficiently perform data flow analysis on a program in a demand-driven manner, we create and iterate over a data-flow graph that contains only the information needed to compute the desired range. Each node in this data-flow graph represents a variable and its data range. An edge exists from the node for variable x to the node for variable y if and only if the computation of the range of x depends upon the range of y . One node in this graph, denoted as *root*, is the node for the variable of the requested data range. All other nodes in the graph that we need to iterate over are reachable from *root*.

The fields of a single node of this graph are shown in Figure 5.4. The *var* field stores the name of the variable representing this node. The *value* field contains the variable's current data range. The *old_value* field holds the value of the variable's data range before its latest update.

```

function create_node( $v$  : variable) : node_ptr
     $x \leftarrow$  new node
     $x.var \leftarrow v$ 
     $x.old\_value \leftarrow \top$ 
     $x.value \leftarrow \top$ 
     $x.committed \leftarrow$  false
     $x.prev \leftarrow \emptyset$ 
     $x.next \leftarrow \emptyset$ 
    if (def( $v$ ) is not an assignment statement) then
         $x.value \leftarrow [-\infty : \infty]$ 
         $x.committed \leftarrow$  true
    end if
    return  $x$ 
end function

```

Figure 5.5: Algorithm to create and initialize a data-flow graph node.

The *committed* field indicates whether the range in the *value* field is the fully computed range for the node's variable. The *next* field represents all edges in the data-flow graph that exit from this node. That is, it contains pointers to nodes whose values this node's range depends upon.⁴ Similarly, the *prev* field represents all edges that enter this node.

The *committed* field of nodes need some additional description. When the *committed* field is set for a node, we say that the node is committed, uncommitted otherwise. When a node is committed, one is assured that the range in the *value* field for this node is the correct data range for the node's variable and will not be modified. Because of this, only the ranges of uncommitted nodes in a data-flow graph need to be computed. Conceptually, committing a node can be thought of as memoizing the range of that node.

The algorithm for creating and initializing a single node in the data-flow graph is shown in Figure 5.5. With the exception of the initialization of the *value* field, all the initializations performed by this algorithm are straightforward. The initial range assigned to the *value* field is

⁴By construction of SSA form, each variable may have at most one definition. Because of this, determining what definitions that a particular expression depends upon is simple.

```

function get_data_range(v : variable) : range
    root ← get_node(v)
    // Assert: root.committed = true
    return root.value
end function

function get_node(v : variable) : range
    if (D(v) has not been defined) then
        root ← create_node(v)
        D(v) ← root
        call add_children_to_node(root)
        // Node root is now fully-initialized
        if (all uncommitted nodes reachable from root have been fully initialized) then
            compute_data_ranges(root)
            commit_data_ranges(root)
        end if
    end if
    return D(v)
end function

```

Figure 5.6: The demand-driven data range propagation algorithm.

determined from the single definition point of the given variable, as indicated by **def**(*v*). If the variable’s definition is not an assignment statement, (e.g., the variable is a formal parameter or is an argument to a procedure call or I/O statement), then its range is set to the unconstrained range ($[-\infty : \infty]$) and committed. Otherwise, its value is set to the undefined range (\top).

5.5.2 Algorithm

The top-level of the algorithm for computing data ranges is shown in Figure 5.6. This algorithm simply calls the function **get_node** to build and iterate over a data-flow graph whose root is the node for the given variable, then returns the computed range stored in this root node.

The function **get_node** has two responsibilities. One of these responsibilities is to build a data-flow graph. More specifically, it creates a node for the given variable as well as data-flow subgraphs for the variables that the given variable’s range may depend on. The function

```

procedure add_children_to_node( $x$  : node_ptr)
  if (def( $v$ ) is an assignment statement) then
    for each variable  $w$  in rhs of def( $v$ ) do
       $y \leftarrow \text{get\_node}(w)$ 
       $x.\text{next} \leftarrow x.\text{next} \cup \{y\}$ 
       $y.\text{prev} \leftarrow y.\text{prev} \cup \{x\}$ 
    end for
  end if
end procedure

```

Figure 5.7: Algorithm to create children for a data-flow graph node.

```

procedure commit_data_ranges( $root$  : node_ptr)
  for each uncommitted node  $x$  reachable from  $root$  do
     $x.\text{committed} \leftarrow \text{true}$ 
  end for
end procedure

```

Figure 5.8: Algorithm to commit data ranges in the data-flow subgraph rooted at $root$.

`create_node`, which is shown in Figure 5.5, creates the node for the given variable. The function `add_children_to_node`, which is shown in Figure 5.7, creates nodes for the variables that the current node depends upon and adds edges between this current node and the newly created nodes by updating their *next* and *prev* fields. These other nodes are created by recursive calls to `get_node`. The global array D is used to memoize created nodes for future reuse.

The other responsibility of function `get_node` is to compute and commit the data ranges for the data-flow subgraph that it has built, if this subgraph is sufficiently complete to safely compute these ranges. By sufficiently complete, we mean that none of the nodes in the subgraph of nodes reachable from $root$ are in the midst of being initialized by other invocations of `get_range`. (Remember that `get_range` is a recursive function). We use Tarjan’s algorithm to detect strongly-connected components⁵ to efficiently determine whether the subgraph rooted at $root$ have been fully initialized.

⁵A strongly-connected component of a graph is a maximal subgraph of that graph where each vertex in the subgraph can reach all other vertices in the subgraph.

If the data-flow subgraph rooted at *root* have been fully initialized, the function `get_node` computes and commits the ranges of this subgraph by calling functions `compute_data_ranges` and `commit_data_ranges`. The implementation of `compute_data_ranges` will be described in the next subsection. The implementation of `commit_data_ranges` is shown in figure 5.8. This algorithm sets the *committed* field to true for all uncommitted nodes in the subgraph rooted at *root*. By committing these nodes, we conceptually memoize the ranges contained in these nodes, since future invocations of `get_data_range` would return the ranges in these nodes rather than recomputing them.

5.5.3 Computing data ranges from a data-flow graph

The computation of data ranges from a data-flow graph uses some basic operations. More specifically, the algorithm uses union (\cup), intersection (\cap), widening (∇), and narrowing (\triangle) operators to compute data ranges. Definitions of these operators can be found in Table 4.4 and Section 4.3.1.

5.5.3.1 Algorithm

The main algorithm for computing data ranges is shown in Figure 5.9. This algorithm computes the data ranges of all nodes whose variable's definitions are ϕ -functions in function `compute_data_ranges_phase`, then computes the data ranges of all other nodes in the graph. The range of one of these other nodes is a single element range whose bounds are the right-hand-side expression of the definition of the node's variable.⁶ The function `compute_data_ranges_phase` is called twice since the ranges of ϕ -functions are computed in

⁶Function `compute_data_ranges_phase` also computes ranges for these other nodes. However, the ranges it generates are more conservative.

```

procedure compute_data_ranges(root : node_ptr)
  compute_data_ranges_phase(root, WIDENING_PHASE)
  compute_data_ranges_phase(root, NARROWING_PHASE)
  for each uncommitted node x reachable from root do
    if (rhs of def(v) is not a  $\phi$ -function) then
      b  $\leftarrow$  rhs of def(x.var)
      x.value  $\leftarrow$  [b : b]
    end if
  end for
end procedure

```

Figure 5.9: Algorithm to compute data ranges from the given graph.

two phases: the *widening phase* and the *narrowsing phase*. The differences between these two phases will be described later in this section.

The implementation of function `compute_data_ranges_phase` is shown in figure 5.10. It performs an iterative data-flow analysis upon all uncommitted nodes in the data-flow graph. More specifically, it initially inserts all uncommitted nodes on the priority queue *work_list*. It then repeatedly removes a node from *work_list*, updates that node's data range, then adds all nodes to *work_list* that depend upon its data range, (i.e., the nodes in *x.prev*), if its data range has changed. The algorithm quits only when the *work_list* becomes empty. To minimize the number of updates performed upon the graph's nodes, the nodes in *work_list* should be ordered by a topological order of the data-flow graph, ignoring any back-edges, (that is, in `rPOSTORDER`, as described in [33]).

The data range (*r*) of a node *x*, whose variable's (*v*) definition contains a ϕ -function, is computed by unioning (\cup) the ranges of the arguments of its ϕ -function. This union results in a range whose lower bound is minimum of the lower bounds of the ranges of the arguments of the ϕ -function and whose upper bound is a maximum of the upper bounds of the arguments'


```

procedure compute_data_ranges_phase(root : node_ptr, phase : phase_type)
  work_list  $\leftarrow$  all uncommitted nodes that are reachable from root
  while (work_list is not empty) do
    x  $\leftarrow$  dequeue(work_list)
    v  $\leftarrow$  x.var
    if (rhs of def(v) is a  $\phi$ -function) then
      r  $\leftarrow$   $\top$ 
      for each edge e entering def(v) do
        y  $\leftarrow$  node in x.next associated with edge e
        s  $\leftarrow$  y.value  $\cap$  get_control_range1(e, v)
        r  $\leftarrow$  r  $\cup$  s
      end for
      if (def(v) has an entering back edge in CFG and x.old_value  $\neq$   $\top$ ) then
        if (phase = WIDENING_PHASE) then
          r  $\leftarrow$  x.value  $\nabla$  r
        else // (phase = NARROWING_PHASE)
          r  $\leftarrow$  x.value  $\triangle$  r
        end if
      end if
    else
      b  $\leftarrow$  rhs of def(v)
      r  $\leftarrow$  [b : b]
      for each node y  $\in$  x.next such that y.committed = false do
        r  $\leftarrow$  r with all occurrences of variable y.var replaced with y.value
      end for
    end if
    x.old_value  $\leftarrow$  x.value
    x.value  $\leftarrow$  r
    if (x.old_value  $\neq$  x.value) then
      work_list  $\leftarrow$  work_list  $\cup$  x.prev
    end if
  end while
end procedure

```

Figure 5.10: Algorithm to compute data ranges for nodes whose definitions are ϕ -functions.

ranges. An argument's range (s) is the intersection (\cap) of the argument's current data range and the control range holding for the argument's control-flow edge.⁷

As for a node whose variable's definition is not a ϕ -function, its data range is initially set to a single element range whose bounds are the right-hand-side of its definition. The algorithm then replaces all variables of uncommitted nodes in this range with their ranges.

One may ask why the algorithm eliminates all variables with uncommitted ranges from such nodes' ranges. The algorithm will run correctly if such variables were not eliminated. However, the resulting ranges are not very useful for they may be directly or indirectly self-referential. A self-referential range is a range assigned to a variable v , where after repeated replacements of variables in the range with those variables' ranges, its symbolic value contains variable v . An example of a directly self-referential range is $x = [1 : x + 1]$. An example of indirectly self-referential ranges is the pair of ranges $x = [1 : y]$ and $y = [1 : x + 1]$. Such self-referential ranges are not very useful because self-referential bounds typically add no constraint information to variables. For example, in the range $x = [1 : x + 1]$ the upper bound adds no information, since all it says is $x \leq x + 1$, which is always true. Hence, this range is equivalent to the simpler range $x = [1 : \infty]$. Another problem with directly or indirectly self-referential ranges is that the expression comparison algorithm, which is described in Chapter 4, has difficulties determining a good order to substitute variables with ranges, resulting in more variable substitutions and less accurate results.

To partially offset the loss of accuracy because of elimination of variables with uncommitted ranges from the nodes' ranges, the loop in function `compute_data_ranges` in Figure 5.9 sets

⁷By construction of ϕ -functions, each argument of a ϕ -function corresponds to one of the entering control-flow edges of the ϕ -function's basic block.

the data range of all nodes whose variable's definition is not a ϕ -function to a single element range whose bounds are the right-hand-side of its definition.

5.5.3.2 Widening and narrowing

One problem with the computation of the data ranges in the data-flow graph, as described above, is that its data ranges may not converge to some fixed value. For example, the data range of an induction variable, ($i = i + 1$), inside a loop can take on the successive ranges $[1 : 1]$, $[1 : 2]$, $[1 : 3]$, \dots . To guarantee that such ranges would reach a fixed point, a *widening operator* [16], denoted as ∇ , is applied to selected ranges in the data-flow graph. This widening operator, which takes the old and new values of a range as its arguments, returns a range whose lower and upper bounds equal the two arguments' lower and upper bounds if they are equal, or infinite bounds otherwise. For example, if the widening operator was applied to the old and new ranges of the induction variable of the previous example, (i.e., $[1 : 2] \nabla [1 : 3]$), it would return the value $[1 : \infty]$.

The main disadvantage of applying the widening operator to ranges in a data flow graph is that it results in overly conservative ranges. Because of this, the algorithm only applies the widening operator on the ranges of nodes in the data flow graph that have entering back-edges, (i.e., loop headers). Additionally, it applies the widening operator only on the third or later updates to such nodes, (the test $x.old_value \neq \top$ in the algorithm determines whether an update is the third or later), to allow the data ranges of such nodes to settle a bit before widening.

By construction of the widening operator, function `compute_data_ranges_phase` is guaranteed to terminate when in the widening phase. This is because each update to a node's range

is more conservative⁸ than its previous value and the application of the widening operator on ranges of loop header nodes guarantees that such nodes would reach a fixed point in a finite number of steps. Since the algorithm computes the ranges of all other nodes directly or indirectly from committed ranges and ranges from loop headers, the ranges of these nodes must also reach a fixed point.

Even with a selective application of the widening operator to the nodes' data ranges in the data-flow graph, one may still suffer from overly conservative results. To partially overcome this, `compute_data_ranges_phase` is called twice by `compute_data_ranges`. The second invocation of `compute_data_ranges_phase` applies a special operator called the *narrowing operator* [16, 11], denoted as \triangle , at those nodes where the widening operator was applied in the previous invocation of `compute_data_ranges_phase`. (The widening operator is not applied in this phase.) This narrowing operator allows the currently computed data range to replace any infinite bounds in the old data ranges with finite bounds derived from data-flow analysis. For example, suppose the induction variable in the previous example was computed in a while loop that tested that the induction variable was less than 100. Then the narrowing operator would allow the variable's data range inside this loop to be changed from $[1 : \infty]$, which was computed in the previous invocation of `compute_data_ranges_phase`, to $[1 : 100]$.

5.5.4 Example

As an example how function `get_data_range` works, suppose that we wish to compute the data range for variable i_4 in the Fortran code fragment shown in Figure 5.11. For this example, the function call `get_data_range(i_4)` creates and iterates over the data-flow graph shown in

⁸When we say more conservative, we mean ranges with smaller (or unchanged) lower bounds and larger (or unchanged) upper bounds.

```

c      Assert:  $n \geq 8$ 
       $i_1 = n$ 
100   CONTINUE
       $i_2 = \phi(i_1, i_4)$ 
      IF ( $x(i_2) > 0$ ) THEN
         $i_3 = i_2/2$ 
      ENDIF
       $i_4 = \phi(i_3, i_2)$ 
      IF ( $i_4 \geq 2$ ) THEN GOTO 100

```

Figure 5.11: Example used to show how function `get_data_range` works.

figure 5.12 for this example. Function `get_range_data` invokes `compute_data_ranges` (and `commit_data_ranges`) twice in this example; once to compute the range of i_1 , and once to compute the ranges for i_2 , i_3 , and i_4 . This is because $\{i_1\}$ and $\{i_2, i_3, i_4\}$ are strongly-connected components of the data flow graph and function `get_node` computes and commits all nodes in a created subgraph as soon as it is fully formed, (i.e., is a strongly-connected component). This eagerness to compute and commit data ranges as early as possible is one of the inherent properties of the algorithm. The reason for this eager computation of ranges is that it maximizes the number of committed ranges that a particular range may depend on, which improves the accuracy of the ranges computed by function `compute_data_ranges_phase`.⁹

The ranges listed alongside each node in the graph represent the values that the node's *value* field take while function `compute_data_ranges_phase` iterates to a fixed point. To give a reader an understanding how these ranges are computed, we will describe in detail how the data range was computed for the node for variable i_2 . Initially, on entry to `compute_data_ranges`, the range of i_2 is the undefined range \top . In the widening phase, the first visit to i_2 ¹⁰ sets its value to the union of the range of i_1 intersected with the control range holding for the

⁹Computing and committing data ranges early also minimizes the number of *poisoned* ranges generated. Poisoned ranges will be discussed in Section 5.6.

¹⁰In this example, we will use a variable's name to represent both the variable and the data-flow node for that variable.

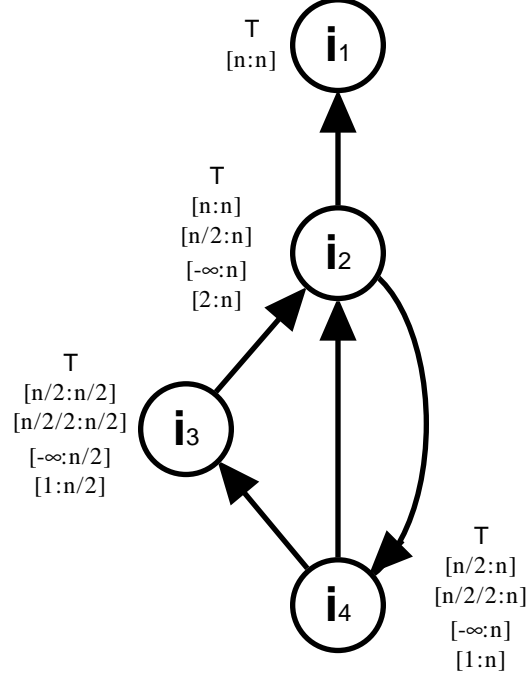


Figure 5.12: The data-flow graph created by function `get_data_range(i_4)` for the example in figure 5.11.

control-flow edge coming from the statement before the `CONTINUE` statement, which would be the unconstrained range $[-\infty : \infty]$ since there is no control-flow constraints on i_1 , and the range of i_4 intersected with the control range holding for the control-flow edge coming from the `IF` statement, which would be $[2 : \infty]$ because of the condition $(i_4 \geq 2)$ in the `IF` statement. That is, the formula of the range of i_2 is:

$$i_2 = (i_1 \cap [-\infty : \infty]) \cup (i_4 \cap [2 : \infty]) = i_1 \cup (i_4 \cap [2 : \infty])$$

(In this formula, we used the techniques described in Section 4.3.1 to simplify the results of unions and intersections.) So the value of the range of i_2 on the first visit to its node is $[n : n] \cup (\top \cap [2 : \infty]) = [n : n]$. Since the range of i_2 have changed, predecessors of node i_2 (i.e., i_3 and i_4), are placed in the work list and the algorithm continues. When the algorithm returns to i_2 , the range of i_3 would have been updated to $[i_2/2 : i_2/2] = [[n : n]/2 : [n : n]/2] = [n/2 : n/2]$

and i_4 would have been updated to $[n/2 : n/2] \cup [n : n] = [n/2 : n]$. At this point, the new range for i_2 is $[n : n] \cup ([n/2 : n] \cap [2 : \infty]) = [n/2 : \infty]$. On the third visit to i_2 , the algorithm determines its range to be $[n : n] \cup ([n/2/2 : n] \cap [2 : \infty]) = [n/2/2 : n]$. Now, since i_2 is a loop header node, the algorithm also applies the widening operator to this result, getting $[n/2 : n] \nabla [n/2/2 : n] = [-\infty : n]$. On the fourth visit to i_2 , the algorithm would find that the range of i_2 is $[-\infty : n] \nabla ([n : n] \cup ([-\infty : n] \cap [2 : \infty])) = [-\infty : n] \nabla [2 : n] = [-\infty : n]$. Since the range of i_2 has not changed, the algorithm will not put i_2 's successors on the work list, causing the widening phase to stop. Function `compute_data_range` would then enter the narrowing phase. In the narrowing phase, the narrowing operator would be applied to the range of i_2 instead of the widening operator. Thus the narrowing phase, on its first visit to i_2 , would compute the range of i_2 to be $[-\infty : n] \triangle ([n : n] \cup ([-\infty : n] \cap [2 : \infty])) = [-\infty : n] \triangle [2 : n] = [2 : n]$. On its second visit of i_2 , the computed range of i_2 would also be $[2 : n]$, causing the narrowing phase to stop.

When function `get_data_range(i_4)` completes, it would have computed and have committed the data ranges $i_1 = [n : n]$, $i_2 = [2 : n]$, $i_3 = [i_2/2 : i_2/2]$ ¹¹, and $i_4 = [1 : n]$. So, the function `get_data_range` would return $[1 : n]$.

5.5.5 Time complexity

Examining the algorithms in Figures 5.6–5.10 one can see that the time taken by the algorithms would be proportional to the size of the generated data-flow graph if one ignored the costs of calling `get_control_range`. By construction of functions `get_node` and `add_children_to_node`, the data-flow graph would have at most $O(|V|)$ nodes and $O(u|V|)$ edges, where $|V|$ is the

¹¹The range $i_3 = [i_2/2 : i_2/2]$ was created in the loop at the end of function `compute_data_ranges`.

number of scalar variables in the program and $u \leq |V|$ is the maximum number of variables used in any assignment statement in the program in SSA form.

By inspection, one can easily see that functions `get_node` and `add_children_to_node` together take at most $O(|V| + u|V|)$ time and function `commit_data_ranges` takes at most $O(|V|)$ time. Computing the time complexity of `compute_data_ranges` is a little more involved. Its cost depends upon the number of times each node in the graph are visited by the algorithm. Because of the use of the widening operator, each node at the start of a cycle in the data-flow graph can change at most four times. Thus each node in the data-flow graph is visited at most a constant number of times. Thus, function `compute_data_ranges` performs at most $O(|V|)$ union operations and $O(u|V|)$ intersection operations and calls to `get_control_range`. Hence, the worst case time complexity of performing a single invocation of `get_data_range` is $O(cu|V| + c|E||V|)$ ¹², where c is the cost of performing an union or intersection, (which equals the time taken to perform a constant number of expression comparisons), and the $c|E||V|$ term is the time complexity for computing all control ranges for all control-flow edges, where $|E|$ is the number of control-flow edges in the program.

Because of the use of memoization, the time complexity of the computation of several data ranges is identical to the time complexity of computing a single range. This is because of the use of memoization in the function `get_node` prevents nodes from be created multiple times and because the nodes' *committed* field prevents the function `compute_data_ranges` from computing the ranges for a node more than once. Hence the sum of the number of edges in all created data-flow graphs can be at most $O(u|V|)$. Additionally, the sum of the number of all nodes in all created data-flow graphs is also at most $O(u|V|)$.

¹²Under big-O notation, $O(A + B) \equiv O(\min(A, B))$.

5.6 Handling union and intersection operations

In our presentation of the algorithms to compute control and data ranges, we have overlooked the complexities associated with making the union or intersection of two ranges. Both of these operations form the bounds of the resulting range by taking the minimum or maximum of the bounds of their two arguments. Difficulties arise because simplifying these minimum and maximum expressions typically requires symbolic expression comparisons, which in turn can perform several `get_range` operations. (We call the expression simplification algorithm to simplify minimums and maximums for two reasons; simpler ranges cause the expression comparison algorithm to run faster, and the widening operator used in the algorithm in Figure 5.9 is much less likely to set simplified range bounds to $\pm\infty$.) One of these recursive calls may request a range that is currently being computed, causing the program to go into an infinite recursion.

5.6.1 Control ranges

Handling this recursion in the control range computation algorithm in Figure 5.2 is not difficult. One only needs to initialize $C(e, v)$, (i.e., the memoized control range of variable v at control-flow edge e), to the unconstrained range $([-\infty : \infty])$ and allow the intersection operation to only use control ranges to compare bounds when simplifying the range. Initially setting the control range of the current variable and statement to $[-\infty : \infty]$ ensures termination, since this assignment guarantees that any recursive invocation of `get_control_range` will not attempt to compute the control range for this variable, statement pair. Because the intersection operator can generate many recursive calls to `get_control_range`, the worst case time complexity of the algorithm in Figure 5.2 increases to $O(c|S||V|)$.

5.6.2 Data ranges

A simple way to handle the recursion caused by union and intersection operations performed in the algorithm `compute_data_ranges_phase` in Figure 5.10 is to allow these operations to use only control ranges to simplify their results. Since the computation of control ranges will never invoke `get_data_range`, the algorithm is guaranteed to terminate. Additionally, the worst case time complexity of the algorithm remains unchanged.

However, by using only control ranges to perform unions and intersections when computing data ranges, the resulting data ranges may lead to overly conservative results. This is because the widening operator may replace partially unsimplified range bounds with $\pm\infty$. Thus, it is desirable to be able to use data ranges in these simplifications as well. Unfortunately, avoiding infinite recursions is complex. We handle this problem by assigning a variable's node a timestamp when we create it in `create_node`. This timestamp, which is associated with a particular invocation of `get_data_range`, is used to identify when a node belongs to an older invocation of `get_data_range`. The functions `compute_data_ranges`, `compute_data_ranges_phase` and `commit_data_ranges` are allowed to only visit nodes created by the current invocation of `get_data_range`. Additionally, if the function `compute_data_ranges_phase` attempts to access the data range of a uncommitted node created by a previous invocation of `get_data_range`, it would use the range $[-\infty : \infty]$ for that node's data range. Also, any node that uses the range of a node created by an older invocation of `get_data_range` is marked as *poisoned*. Poisoned nodes are nodes that cannot be memoized, (i.e., their values cannot be stored in R in Figure 5.1), nor committed, since their data ranges may be overly conservative. Instead, they are deleted from the data-flow graph on the exit of `get_data_range`. Any node that uses the data range of a poisoned node is itself poisoned.

<i>Code</i>	<i>Number of lines</i>	<i>Computing data ranges (s)</i>	<i>Computing control ranges (s)</i>	<i>Merging data and control ranges (s)</i>
ARC2D	3573	2.4	1.3	1.4
BDNA	5960	3.8	5.4	4.3
FLO52	3348	3.9	4.9	2.7
MDG	1487	0.7	1.9	1.1
OCEAN	3142	8.0	4.6	25.3
TRFD	965	1.7	4.9	17.6

Table 5.1: Time taken in seconds to compute all data ranges, all control ranges, and merge all data and control ranges on a Sparc 10.

5.7 Performance

To show that our demand-driven range propagation algorithm is efficient for real programs, even when called many times, we have measured the time taken to compute all control and data ranges in a program. All optimizations described in this chapter have been implemented in these algorithms. These times are displayed in Table 5.1. The *code* column displays the name of each Fortran code examined. These codes were taken from the Perfect Benchmarks, which is a suite of Fortran 77 programs representing applications in a number of areas in engineering and scientific computing [6]. The *Number of lines* column displays the number of lines in each code after being converted into SSA form. The *Computing control ranges* and *Computing data ranges* columns give the total times taken to compute every control and data range respectively in each of the codes. The *Merging data and control ranges* column shows the time taken to intersect all the control and data ranges in the program, as done by the algorithm in Figure 5.1. The total time to compute all the ranges in the program is just the sum of these columns. All timings are user times measured on a Sparc 10, using g++ 2.6.3 with the flag -O.

As Table 5.1 has shown, computing all data and control ranges is very efficient. However, merging them together can be expensive. This cost of merging them together arises from the

intersection operation in Figure 5.1. In our experience, much of this time in computing and merging control and data ranges is spent computing the intersection and union of ranges. This cost arises from the fact that the symbolic expression comparison algorithm, as described in Chapter 4, is used to simplify the intersection and union of ranges, and this algorithm can be expensive. However, unions and intersections are very cheap if one of their arguments is $[-\infty : \infty]$

Table 5.1 has shown that the demand-driven range propagation algorithm is efficient for computing all ranges in a program. However, it says little about the costs of computing a single range. As subsections 5.4.3 and 5.5.5 have shown, the demand-driven range propagator may need to compute the values of other control and data ranges to determine the value of a certain range. In the worst case, every data and control range may need to be computed.

To determine the efficiency of the demand-driven range propagation algorithm for computing a single range, we have measured the number of control and data ranges computed by the algorithm to determine this range. Since the running time of the algorithm is proportional to the number of ranges it needs to compute, the fraction of computed control and data ranges computed out of the set of all control and data ranges should roughly indicate what fraction of the execution times shown in Table 5.1 that a typical range computation takes.

We have collected both the average and the maximum number of data and control ranges computed for a single invocation of `get_control_range` from Figure 5.2 and `get_data_range` from Figure 5.6. Tables 5.2 and 5.3 displays these results. These numbers were computed by requesting each control or data range in a program, then counting the number of control and data ranges created by each request. All memoized ranges, (i.e., the ranges stored in R , C , and D , from functions `get_range`, `tt get_control_range`, `get_data_range`), were cleared before each

<i>Code</i>	<i>No. control ranges</i>	<i>No. computed</i>	
		<i>Avg.</i>	<i>Max.</i>
ARC2D	10227	1.9	5
BDNA	27285	2.7	8
FLO52	35094	2.6	11
MDG	4152	2.4	7
OCEAN	94819	2.4	37
TRFD	9994	3.1	21

Table 5.2: Average and maximum number of control ranges computed when computing a single control range.

request for a control or data range. Ideally, the average and maximum number of control ranges created should be one when a single control range is requested. Also, the ideal average and maximum number of data ranges created should be one and the ideal average and maximum number of control and poisoned ranges created should be zero when a data range is requested.

Examining Table 5.2, one can see that typically only two or three control ranges need to be computed to determine the value of a single control range. This low number is mostly due to the use of the sparse `icdom` optimization, as described in section 5.4. This sparse `icdom` relationship usually causes the algorithm in Figure 5.2 to compute the control ranges for only the control-flow statements, (e.g., `IF` and `DO` statements) that enclose the current statement. Since our test programs do not have deeply nested control flow, it is not surprising that the average number of computed control ranges is small.

One can roughly determine the cost of computing a control range by dividing the total time taken by `get_control_range`, as shown in Table 5.1, by the total number of control ranges in the program, as shown in column *No. control ranges* in Table 5.2, then multiplying this result by the number of ranges computed for that control range. Doing this, we find that the average control range computation takes about a few hundred microseconds, and the longest

<i>Code</i>	<i>No. data ranges</i>	<i>Control</i>		<i>Data</i>		<i>Poisoned</i>	
		<i>Avg.</i>	<i>Max.</i>	<i>Avg.</i>	<i>Max.</i>	<i>Avg.</i>	<i>Max.</i>
ARC2D	1002	2.9	31	2.4	20	0.1	24
BDNA	1431	5.5	47	3.9	29	1.3	40
FLO52	1142	22.9	243	9.8	97	2.8	54
MDG	436	4.0	20	2.9	11	1.1	49
OCEAN	1529	10.5	201	6.6	64	0.4	30
TRFD	521	8.4	68	4.4	21	0.2	6

Table 5.3: Average and maximum number of control ranges, data ranges, and poisoned data ranges computed when computing a single data range.

control range computation takes about a few milliseconds. Thus, we feel confident to claim that computing a single control range using a demand-driven algorithm is very efficient.

Table 5.3 displays the average and maximum number of control, data, and poisoned ranges computed per data range.¹³ The overall average and maximum cost of computing a data range can be approximated by adding these averages and maximums respectively. This table shows that computing a data range typically causes the algorithm to compute several data and control ranges, possibly many data and control ranges in the worst case. Additionally, the large discrepancies between the averages and the maximums indicate that the costs of computing a data range may vary greatly. Despite the potentially large number of data ranges computed, the average and maximum data and control ranges computed is still a small fraction of the total number of control and data ranges in the program, so a demand-driven data range computation algorithm is still more efficient than its non-demand-driven counterpart.

We can determine the rough cost of computing a data range by dividing the total time taken by `get_data_range` by the total number of data ranges in the program, then multiplying by the

¹³Poisoned ranges are overly conservative data ranges computed by a recursive call to `get_data_range`. Unlike control and data ranges, poisoned ranges are not memoized, so they may be repeatedly recomputed when computing a data range. We count such ranges multiple times, once per computation, in the table. See section 5.5 for more details.

number of data and poisoned ranges ranges computed for that data range. Doing this, one can determine that the average time taken to compute a data range is a few tens of milliseconds and the worst case time is a few hundreds of milliseconds for real codes.

If one finds the cost of computing a single data range to be too expensive, one can sometimes compute and use only control ranges. We have found that using only control ranges, coupled with symbolic constant propagation and induction variable substitution, provides sufficient information for applications of range propagation by parallelizing compilers on some Fortran programs. This is because such transformations transform most expressions in a program into expressions made up of only symbolic constants and enclosing loop indices, and one only needs to know the constraints imposed upon these loop indices and symbolic constants to compare or compute the ranges of such expressions.

5.8 Related work

The idea for representing program constraints as ranges was first proposed by Harrison [31] for array bounds checking and program verification. In his paper, Harrison describes how one can compute the range of integer values that variables can take in a program unit, using data-flow analysis. Although he does propose simple techniques to handle symbolic ranges, our symbolic analysis techniques are superior. (He restricts the bounds of his symbolic ranges to the form `< variable > + < constant >`.)

Bourdoncle [11] greatly improves the accuracy of the integer range propagation algorithm by Harrison, through the use of abstract interpretation [16]. Our use of the narrowing operator was influenced by his algorithm. Bourdoncle's algorithm is unable to generate symbolic ranges. Neither Harrison's nor Bourdoncle's algorithms are demand-driven nor do they use a sparse

data-flow representation of a program, such as SSA form or definition-use chains, to improve the efficiency of their algorithms.

Cousot and Halbwachs [17] presents a different method to compute and propagate constraints through a program. In their technique, sets of constraints between variables are represented as a convex polyhedron in the n -space of variable values. Because of this representation, all constraints are restricted to be in the form of affine inequality relationships, (e.g., $5 * x + 2 * y \leq 2$). Abstract interpretation is used to compute the convex polyhedron of variable constraints for each statement and each control-flow edge of the program.

They are more accurate in the computation and propagation of affine variable constraints than our algorithm. However, they cannot handle non-affine variable constraints, such as $a < b * c$. Additionally, by using a convex hull representation to compute variable constraints, their algorithm cannot benefit from a sparse data-flow representation of a program. Because of this, their algorithm can be much less efficient than ours. Also, their convex hull representation prevents one from creating a demand-driven version of their algorithm that is not overly complex.

Tu and Padua [45] also present a demand-driven, symbolic expression comparison and constraint propagation technique, based on an extension of SSA called gated SSA form. Their technique compares expressions by repeatedly substituting variables with their constant symbolic values until the two expressions differ by only an integer constant. The values to substitute are determined by a demand-driven analysis of the program. Variants of ϕ -functions, which can be substituted in other expressions, are used to represent ranges of values. (These variants of ϕ -functions are simply ϕ -functions extended to contain conditional predicates that indicate

which of their arguments should be their result.) Rewrite rules are used to simplify expressions containing such ϕ -functions.

One strength of their algorithm is that they can perform flow-sensitive analyses. For example, their algorithm can determine that \mathbf{x} equals \mathbf{a} at **S1** for the following code fragment:

```

    IF (y > 0) THEN
        x = a
    ENDIF
    ...
    IF (y > 1) THEN
S1:      ... = x
    ENDIF

```

Another strength is that their algorithm can use semantical information offered by **DO** loops to compute more accurate constraints, such as perform induction variable substitution on the fly. They are also able to derive constraints on array elements.

Their algorithm also has weaknesses when compared to ours. First, they do not perform memoization. Thus, their algorithm can be much slower than ours when the constraints for many variables are requested. Second, their algorithm cannot derive many of the constraints generated by our `get_control_range` function. Basically, they can only derive constraints equivalent to our data ranges and the control ranges for loop indices.

The differences between our algorithm and theirs is mainly due to that the two applications were designed to handle two different problems. Their algorithm was designed to compare the bounds of array sections for array privatization [43]. Because conditional array definitions and uses occur in a significant fraction of important loop nests, flow-sensitive analysis is essential to successfully perform such comparisons. On the other hand, the bounds being compared are usually very similar to each other, requiring the substitution of only a few variables with thier constant values to make the two bounds equal each other except for a constant offset. On the

other hand, range propagation was originally designed for dependence testing. Our symbolic data dependence test, called the Range Test, often needs to compare expressions that are more complicated and dissimilar to each other than the expressions compared for array privatization. Because of this, more constraint information and a more powerful expression comparator is needed to compare such expressions. Also, dependence testing requests many more constraints, making memoization much more important. In our experience, conditionally defined constants and constraints does not significantly improve the effectiveness of dependence testing. Because of these differences, Polaris includes implementations of both range propagation and Tu's and Padua's gated-SSA demand-driven analysis.

5.9 Conclusions

We have developed a demand-driven range propagation algorithm and have shown it to be efficient for computing a single range as well as computing many ranges. Because of its efficiency, it is feasible to use Range Propagation at many points in a compiler without a serious degradation of the compiler's performance, even though the program may be modified between these points. Since these ranges can be used to perform symbolic expression comparisons and range computation of expressions, and these operations enable powerful symbolic analyses, Demand-Driven Range Propagation can significantly increase the effectiveness of parallelizing and optimizing compilers.

Chapter 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this dissertation, we have identified several symbolic analysis techniques that can significantly improve the effectiveness of parallelizing compilers and have developed algorithms for the two of the most important techniques. These two techniques are the Range Test and Range Propagation. The Range Test is a symbolic data dependence test designed to handle the nonlinear array subscripts and loop bounds seen in real programs. Range Propagation computes symbolic constraints, called ranges, on variables for each point in a program and uses these constraints to compare arbitrary, possibly nonlinear, expressions.

We have implemented both the Range Test and Range Propagation in Polaris, a parallelizing compiler being developed at the University of Illinois. We have found that these two techniques, along with other advanced techniques developed by our group, significantly im-

prove the effectiveness of parallelizing compilers, (see Figure 1.3). Additionally, we found the Range Test and Range Propagation to be reasonably efficient. For example, even if we use the Range Test as the only major data dependence test in Polaris, and we use the slower version of Range Propagation that uses abstract interpretation to compute its ranges, our techniques together take up at most a third of our compiler’s execution time. To improve the efficiency of Range Propagation, we have also developed a demand-driven version of the range computation algorithm.

6.2 Future work

There is still much work that can be done on symbolic analysis for parallelizing compilers as well as on the Range Test and Range Propagation.

Our analysis of the Perfect Benchmarks for needed symbolic analysis techniques can be extended in several ways. First, we did not develop algorithms nor implement about half of the techniques identified in our study. The two most important of these unimplemented techniques are the analysis of subscripted subscripts, (e.g., $A(x(i))$) and run-time tests. Second, we never verified the importance of these identified techniques by measuring the actual effectiveness of the implementations of these techniques on the Perfect Benchmarks. Hence, we do not know whether the identified techniques are necessary and sufficient for parallelizing the Perfect Benchmarks or just necessary. Third, it would be desirable to extend our study to other application codes, such as the SPEC benchmarks.

There is plenty of room for improvement for the Range Test. We can extend the Range Test to handle some of its deficiencies it had compared to the Omega Test. This includes extensions to handle coupled subscripts or to use ranges with non-unit strides. The Range Test can also be

extended to handle some of the identified needed symbolic analysis techniques such as run-time tests or analysis of subscripted subscripts. Extending the Range Test to generate run-time tests should not be difficult. All one needs to do is to record all expression comparisons that it couldn't determine to be true or false and generate a multi-version loop that tests these comparisons and chooses whether to run the parallel or sequential version of the loop nest. We believe that with extensions to Range Propagation, the Range Test may also be able to handle subscripted subscripts, where the compiler can determine constraints on the subscript arrays.

Several improvements can be made to Range Propagation. One improvement would be to extend Range Propagation to compute a program's ranges interprocedurally. Another improvement is to extend Range Propagation to determine and use ranges on array elements. This would allow the Range Test to handle subscripted subscripts. Finally, it would be desirable to merge our Range Propagation with Tu's and Padua's demand driven analysis [45], combining the strengths of both techniques.

BIBLIOGRAPHY

- [1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.
- [2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.
- [3] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer. Boston, MA, 1988.
- [4] Utpal Banerjee. A Theory of Loop Permutations. In *2nd Workshop on Languages and Compilers for Parallel Computing*, pages 54–74. The MIT Press, 1990.
- [5] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, 81(2), February 1993.
- [6] M. Berry, D. Chen, P. Koss, D. Kuck, L. Pointer, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *Int'l. Journal of Supercomputer Applications, Fall 1989*, 3(3):5–40, Fall 1989.
- [7] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The Next Generation in Parallelizing Compilers. *Proceedings of the Workshop on Languages and Compilers for Parallel Computing, Ithaca, New York*, pages 10.1 – 10.18, August 1994.
- [8] William Blume and Rudolf Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Transactions of Parallel and Distributed Systems*, 3(6):643–656, November 1992.
- [9] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu. Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing. *IEEE Parallel and Distributed Technology*, 2(3):37–47, Fall 1994.
- [10] William Joseph Blume. Success and Limitations in Automatic Parallelization of the Perfect Benchmarks Programs. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., July 1992.

- [11] François Bourdoncle. Abstract Debugging of Higher-Order Imperative Languages. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 46–55, June 1993.
- [12] Preston Briggs, Keith D. Cooper, Mary W. Hall, and Linda Torczon. Goal-Directed Interprocedural Optimization. Technical report, Rice University, November 1990. TR90-147.
- [13] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural Constant Propagation. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 152–161, June 1986.
- [14] Thomas E. Cheatham Jr., Glenn H. Holloway, and Judy A. Townley. Symbolic Evaluation and the Analysis of Programs. *IEEE Transactions on Software Engineering*, SE-5(4):402–417, July 1979.
- [15] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.
- [16] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.
- [17] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–97, 1978.
- [18] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- [19] R. J. Duffin. On Fourier's Analysis of Linear Inequality Systems. *Mathematical Programming Study 1*, pages 71–95, 1974.
- [20] Rudolf Eigenmann and William Blume. An Effectiveness Study of Parallelizing Compiler Techniques. *Proceedings of ICPP'91, St. Charles, IL*, II:17–25, August 12-16, 1991.
- [21] Rudolf Eigenmann, Jay Hoeflinger, G. Jaxon, and David Padua. The Cedar Fortran Project. Technical Report 1262, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., April 1992.
- [22] Rudolf Eigenmann, Jay Hoeflinger, Greg Jaxon, Zhiyuan Li, and David Padua. Restructuring Fortran Programs for Cedar. *Concurrency: Practice and Experience*, 5(7):553–573, October 1993.
- [23] Rudolf Eigenmann, Jay Hoeflinger, Zhiyuan Li, and David Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Lecture Notes in Computer Science 589. Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pages 65–83, August 1991.

- [24] Keith A. Faigin, Jay P. Hoeflinger, David A. Padua, Paul M. Petersen, and Stephen A. Weatherford. The Polaris Internal Representation. *International Journal of Parallel Programming*, 22(5):553–286, October 1994.
- [25] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical Dependence Testing. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 15–29, June 1991.
- [26] Mohammad Haghighat and Constantine Polychronopoulos. Symbolic Dependence Analysis for High-Performance Parallelizing Compilers. *Parallel and Distributed Computing: Advances in Languages and Compilers for Parallel Processing*, MIT Press, Cambridge, MA, pages 310–330, 1991.
- [27] Mohammad Haghighat and Constantine Polychronopoulos. Symbolic Analysis: A Basis for Parallelization, Optimization, and Scheduling of Programs. *Proceedings of the Sixth Annual Languages and Compilers for Parallelism Workshop, Portland, Oregon*, August 1993.
- [28] Mohammad R. Haghighat and Constantine D. Polychronopoulos. Symbolic Program Analysis and Optimization for Parallelizing Compilers. *Presented at the 5th Annual Workshop on Languages and Compilers for Parallel Computing, New Haven, CT*, August 3-5, 1992.
- [29] Mohammad Reza Haghighat. Symbolic Analysis for Parallelizing Compilers. Master’s thesis, Univ of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, August 1994.
- [30] D. Harel. A linear time algorithm for finding dominators in a flow graph and related problems. *Proceedings of the 17th ACM Symposium of Theory of Computing*, pages 185–194, May 1985.
- [31] William H. Harrison. Compiler Analysis of the Value Ranges for Variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.
- [32] Paul Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Rice University, May 1994.
- [33] Matthew S. Hecht and Jeffrey D. Ullman. A Simple Algorithm for Global Data Flow Analysis Problems. *SIAM Journal on Computing*, 4(4):519–532, December 1975.
- [34] Jay Hoeflinger. Run-Time Dependence Testing by Integer Sequence Analysis. Technical Report 1194, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., January 1992.
- [35] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library (version 0.91) Interface Guide. Technical report, University of Maryland, February 1995.
- [36] Vadim Maslov. Delinearization: An Efficient Way to Break Multiloop Dependence Equations. *Proceedings of the SIGPLAN ‘92 Conference on Programming Language Design and Implementation*, pages 152–161, June 1992.

- [37] D. Maydan, J. Hennessy, and M. Lam. Efficient and exact data dependence analysis. In *SIGPLAN NOTICES: Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, June 26-28*, pages 1–14. ACM Press, 1991.
- [38] Kathryn S. McKinley. Dependence Analysis of Arrays Subscripted by Index Arrays. Technical report, Rice University, June 1991. TR91-162.
- [39] D. Padua, R. Eigenmann, J. Hoeftlinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin. Polaris: A New-Generation Parallelizing Compiler for MPP's. Technical Report 1306, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., June 1993.
- [40] D. Padua and M. Wolfe. Advanced Compiler Optimization for Supercomputers. *CACM*, 29(12):1184–1201, December, 1986.
- [41] Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. *Proceedings of the 9th ACM International Conference on Supercomputing, 1995*, April 1995.
- [42] William Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8):102–114, August 1992.
- [43] Peng Tu and David Padua. Automatic Array Privatization. In Utpal BanerjeeDavid GelernterAlex NicolauDavid Padua, editor, *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science.*, volume 768, pages 500–521, August 12-14, 1993.
- [44] Peng Tu and David Padua. Demand-Driven Symbolic Analysis. Technical Report 1336, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., February 1994.
- [45] Peng Tu and David Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. Technical Report 1399, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., January 1995.
- [46] Mark N. Wegman and Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.
- [47] H. P. Williams. Fourier's method of Linear Programming and its Dual. *The American Mathematical Monthly*, 93(9):681–695, November 1986.
- [48] Michael Wolfe. Beyond induction variables. In *Proc. ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 162–174, 1992.
- [49] Michael Wolfe. Triangular Banerjee's Inequalities with Directions. Technical report, Oregon Graduate Institute of Science and Technology, June 1992. CS/E 92–013.
- [50] Michael Wolfe and Utpal Banerjee. Data Dependence and its Application to Parallel Processing. *International Journal of Parallel Programming*, 16(2):137–178, 1987.

- [51] Hans P. Zima and Barbara M. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, NY, 1991.

VITA

William Joseph Blume was born on March 6th, 1967 in Maywood, Illinois. He received a Bachelor in Science degree in Computer Science with highest honors in May 1989. From August 1989 to June 1995, he has been a graduate student of the compiler group at the Center of Supercomputing Research and Development at the University of Illinois. He earned his Masters in Science degree in Computer Science in August 1992 and his Doctor of Philosophy in Computer Science in June 1995. He also spent the summers of 1989-1991 at Northrop as an engineering intern. For his last year as a graduate student, he was supported by an ARPA fellowship. After graduation, he will join the California Languages Laboratory Department at Hewlett Packard.