HYBRID ANALYSIS OF MEMORY REFERENCES AND ITS APPLICATION TO AUTOMATIC PARALLELIZATION

A Dissertation

by

SILVIUS VASILE RUS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2006

Major Subject: Computer Science

HYBRID ANALYSIS OF MEMORY REFERENCES AND ITS APPLICATION TO AUTOMATIC PARALLELIZATION

A Dissertation

by

SILVIUS VASILE RUS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Lawrence Rauchwerger
Committee Members,	Nancy Amato
	Narasimha Reddy
	Vivek Sarin
Head of Department,	Valerie Taylor

December 2006

Major Subject: Computer Science

ABSTRACT

Hybrid Analysis of Memory References
and Its Application to Automatic Parallelization. (December 2006)
Silvius Vasile Rus, B.S., Babes-Bolyai University
Chair of Advisory Committee: Dr. Lawrence Rauchwerger

Executing sequential code in parallel on a multithreaded machine has been an elusive goal of the academic and industrial research communities for many years. It has recently become more important due to the widespread introduction of multi-cores in PCs. Automatic multithreading has not been achieved because classic, static compiler analysis was not powerful enough and program behavior was found to be, in many cases, input dependent. Speculative thread level parallelization was a welcome avenue for advancing parallelization coverage but its performance was not always optimal due to the sometimes unnecessary overhead of checking every dynamic memory reference.

In this dissertation we introduce a novel analysis technique, Hybrid Analysis, which unifies static and dynamic memory reference techniques into a seamless compiler framework which extracts almost maximum available parallelism from scientific codes and incurs close to the minimum necessary run time overhead. We present how to extract maximum information from the quantities that could not be sufficiently analyzed through static compiler methods, and how to generate sufficient conditions which, when evaluated dynamically, can validate optimizations.

Our techniques have been fully implemented in the Polaris compiler and resulted in whole program speedups on a large number of industry standard benchmark applications.

To Ixel, Ileana and Vasile

ACKNOWLEDGMENTS

First, I would like to thank Lawrence Rauchwerger, my academic advisor for teaching me how to be a researcher. He taught me how to read scientific papers, how to extract the essence from a free roaming idea, and how to design, run and understand scientific experiments. I owe Lawrence for never having to worry a single day about funding throughout my entire Ph.D. student life. Above all, he did well what any good advisor should do: give good advice. Perhaps not all was taken, but all was heard, listened to and learned from.

I want to thank Nancy Amato for her advice, but even more so for making time to read my papers even when she had her own close deadlines. She is a model to follow regardless where I go from here. I would like to thank my committee members Narasimha Reddy and Vivek Sarin for taking the time to advise me on my research.

I am grateful to Jay Hoeflinger for his collaboration on my first paper. A part of my experimental setup was built on top of code written by him. I thank Marvin Adams for giving me the opportunity to work on an important nuclear engineering project and for teaching me some of the reasons why physicists want more computing power. Francis Dang helped me run experiments as a colleague and later as the administrator of the on-site supercomputer. Hao Yu helped me understand some more difficult parts of my research topic. Guobin He, Dongmin Zhang and Marinus (Maikel) Pennings helped me with compiler development and also helped me validate my ideas before putting them into practice. I benefited much from talking to, listening to, and being around other students in my group: Alin Jula, Julio Antonio Carvallo de Ochoa, Nathan Thomas, Tim Smith, Gabriel Tănase, Steven Saunders, William McLendon and Antoniu Pop.

I would like to thank the people I worked with at IBM Research, Gheorghe Almási, José Moreira and Manish Gupta, for giving me the opportunity to be a small part of the BlueGene/L project. I learned to be a better person from their attitude, professionalism and dedication.

I thank the Computer Science Department and the GAANN program at the Department of Education for supporting me through a teaching assistantship and a fellowship respectively. I would like to thank the administrative staff at the Computer Science Department, and especially Kay Jones, Sandra Morse and Elena Rodriguez, for making complicated procedures seem easy.

Long before I became a Ph.D. student, several great people shaped my mind. My first teacher, Raveca Găurean, cultivated the values seeded by my parents: respect, correctness and common sense. My math teachers, Gheorghe Simionaş, Nicolae Sanda, Petru Ivănescu, and Csaba Varga, made me understand that memorizing a formula makes you its slave, while knowing how to create it makes you its master.

My close friend, Merrill, made my stay in Bryan/College Station better, lighter on the heart. He is not only a friend, but also a fellow scientist, my landlord for the last two years and a surrogate parent. Ben Krieger was my first host in College Station and helped me learn the ways of Texas and Aggieland. Deborah Lord and Alin Jula were my good friends from the beginning.

My mother and father deserve the credit for making me into who I am. They have supported me wholeheartedly throughout my life. They have believed in me and have inspired me with their successful and happy lives. I am forever thankful to them, as well as to my brother, Marius, my uncle Tudor and aunt Maria for their continuous support.

To my wife, Ixel, I owe the magic of being able to smile every day. She has given me strength and hope. Her vision of life opened my eyes and changed my life forever.

TABLE OF CONTENTS

CHAPTER Page

I INTRODUCTION 1

A. Parallel Computers 1
B. Automatic Parallelization 2
C. Hybrid Analysis 4
D. Contribution 6
E. Organization 7

II FUNDAMENTALS AND PREVIOUS WORK 8

A. Fundamentals of Automatic Parallelization 8
1. Scalar Data Flow and Data Dependence 8
2. Array Data Flow and Data Dependence 10
B. Current State of the Art in Automatic Parallelization . . . 12
1. Compiler Analysis 12
a. Data Flow Analysis 12
b. Data Dependence Analysis 15
c. Analysis of Array References 18
2. Compiler Transformations 20
a. Removing Flow Dependences 20
b. Removing Storage Related Dependences 21
c. Results in Automatic Parallelization 21
3. Run Time Parallelization Techniques 22
a. Instrumentation of Memory References 22
b. Optimization Predicate Extraction 24
c. Partially Parallel Loops and Communication Schedules 25
d. Inspector Executor vs. Speculative Optimization . 26

III HYBRID MEMORY REFERENCE ANALYSIS 27

A. An Overview of Hybrid Analysis Applied to Parallelization 28
B. Proposed Memory Reference Representation:USR 30
1. Program Model 33
2. Background: the Linear Memory Access Descriptor . . 33
3. Abstraction of Set Operations 35

CHAPTER

a. Set Intersection 35
b. Set Difference 36
c. Set Union 37
4. Abstraction of Loops 38
5. Abstraction of Control 40
6. Abstraction of Subprograms 40
7. Formal Definition 42
C. Hybrid Memory Reference Analysis using USRs 43
1. Memory Classification Analysis 44
a. Classification of References in Straight Line Code 45
b. Classification of References in Conditional Blocks 46
c. Classification of References in Loops 47
d. Interprocedural Classification of References 50
2. Dependence Testing as Verification of USR Identities . 51
D. Hybrid Dependence Analysis 52
1. Symbolic Representation: the PDAG 53
2. Symbolic Analysis Algorithms 54
a. Syntax Directed Predicate Extraction 54
b. Extracting PDAGs from USR Approximations . . 57
c. Predicate Extraction from Finite Valued USRs . . 59
d. Extracting PDAGs from LMAD Equations 60
3. Testing Monotonicity and Disjoint Intervals 60
4. Reference Pattern Library: Extensible Compiler . . . 61
5. Fallback: Reference-based Dependence Tests 62
6. Case Study 62
E. Other Applications of Memory Reference Analysis 65
1. Array Data Flow Analysis 65
a. Region Array SSA 66
b. Transformations Based on Data Flow Analysis . . 68
2. Efficient Recompilation 68
3. Program Verification and Symbolic Debugging 70
F. Related Work 71
G. Conclusions 73

IV SYMBOLIC VALUE ANALYSIS 75

A. Motivation 75
1. Background and a Motivating Example 77
2. Our Solution: The Value Evolution Graph 78

CHAPTER

B. The Value Evolution Graph (VEG) 80
1. Formal Definition 81
2. Value Evolution Graph Construction 83
3. Queries on Value Evolution Graphs 85
4. VEG Conditional Pruning 86
C. VEG-based Memory Reference Analysis 87
1. Using the VEG in Memory Classification Analysis . . 89
2. Memory Reference Sequence Classification 90
3. VEG Applications to Classic Compiler Optimizations 92
a. Dataflow Analysis 92
b. Privatization 93
c. Dependence Analysis 93
4. Recognition of Pushbacks and
Other Parallelizable Prefix Computations 94
a. Pushback Sequences 95
b. Other Parallelizable Sequences 96
D. Case Studies 97
1. ADM/DKZMH_do60 100
2. TRACK/EXTEND_do400 100
E. Related Work 102
1. Recurrence Recognition, Classification, and Parallelization 102
2. Analysis of Memory Referenced by Recurrences without Closed Forms 103
F. Conclusions and Future Work 105

V ENGINEERING A HYBRID AUTOMATIC PARALLELIZER . 106

A. Automatic Parallelizer Overview 106
B. Static vs. Dynamic Parallelization 107
C. Dynamic Optimization Strategy 109
1. Inspector/Executor 109
2. Speculation 110
a. Checkpointing 110
3. Inspector/Executor vs. Speculative Execution 111
D. Transformations to Remove Dependences 112
1. Hybrid Privatization 114
a. Hybrid Copy In 116
b. Hybrid Copy Out and Last Value Assignment . . 116

CHAPTER

2. Hybrid Reduction Parallelization 118
3. Pushback Sequence Parallelization 119
E. Automatic Detection of Array Bounds 119
F. Case Study: DYFESM/MXMULT_do10 120
1. Discussion 129

VI COMPILER DESIGN AND IMPLEMENTATION 131

A. Making General Applications Fit our Program Model . . . 131
1. Bringing Programs to Block Structured Form 132
2. Alias Disambiguation 134
a. Commons 135
b. Equivalence 136
c. Type Mismatches across Subprograms 136
3. Language and Programming Style Issues 137
a. Array Bound Declarations 138
b. Multiple Subprogram Entries 138
c. Data Statements 139
d. Lazy Initialization Code 139
B. USR Design and Implementation 140
1. USR Optimization 141
a. Optimization Based on Minimal Evaluation Cost Form 142
b. Partial Invariants 143
c. Approximation with LMAD Lists 144
d. Language Specific Optimization 145
C. PDAG Design and Implementation 147
D. Complexity of Hybrid Analysis 149
1. Compile Time Complexity 149
a. Memory Classification Analysis 149
b. PDAG Extraction 151
2. Run Time Complexity 152
a. USR Evaluation 152
b. PDAG Evaluation 153

VII EMPIRICAL EVALUATION 156

A. Methodology 156
1. Hardware and Software Environment 157
2. Input Data Sets 159

CHAPTER

3. Performance Metrics 161
B. Hybrid Analysis Automatic Parallelization Results 162
C. Evaluation of Run Time Tests 164
D. Comparison to Other Parallelizing Compilers 165
1. The Intel® Compiler 165
2. The *IBM Toronto Lab* Parallelizing Compiler 167
3. The SUIF Research Compiler 168
E. Discussion of Important Loops 170
1. Value Evolution Graph 173
2. USR Based Memory Reference Analysis 173
3. PDAG Based Efficient Run Time Tests 174
a. ADM/APSI 174
b. MDG 175
c. DYFESM 175
d. OCEAN 175
4. Dynamic Parallelization, Privatization and Reduction 176

VIII CONCLUSIONS AND FUTURE WORK 177

A. Contributions 178
1. Program Representation 178
2. Program Analysis 179
B. Future Work 179
1. Extending Hybrid Analysis 179
a. Hybrid Pointer Analysis 180
b. Hybrid Optimization for High Level Languages . 181
2. Applications of Hybrid Dataflow Analysis 182
a. Generation of Communication Schedules 182
b. Compiler-based Cache Coherence 182
c. Symbolic Debugging and Verification 183
d. Other Uses 183
3. Dynamic Compilation Based on Input Sensitivity . . . 184

REFERENCES 185

APPENDIX A 218

APPENDIX B 224

VITA 237

LIST OF TABLES

TABLE		Page
I	Static dependence tests.	17
II	MCA partitions for the privatization problem on array A in Fig. 11(a).	44
III	Extracting evolutions from the program.	83
IV	Uses of memory reference sequence classification for the parallelization of the outer loop of a doubly nested loop.	92
V	Loops parallelized. CP = Conditional Pushback, SL(U) = Stack Lookup (and Update), P-CW = Privatization based on Contiguous Writes, P-VEG = Privatization using the VEG directly.	98
VI	Comparison to recent work on memory referenced through recurrences without closed forms.	102
VII	Comparison of parallel code generation strategies.	107
VIII	Comparison of run time privatization strategies.	115
IX	Run time tests actually executed to decide whether the dependence structure on array MX prohibits or allows parallelization. %S represents the time spent in the test as a percentage of the execution time of the loop.	129
X	Attribute grammar for generating Fortran code for USRs.	140
XI	USR evaluation cost model.	142
XII	USR approximation using LMAD lists.	145
XIII	Compile-time analysis statistics in seconds for both MCA and PDAG extraction for parallelization. Column 4 and 5 show the total number of USR and PDAG nodes created (operator or leaves).	151

TABLE

XIV Run time test dynamic overhead reduction through HDA: ratio between the number of actual memory references and the number of PDAG operations performed at run time. Only the applications with run time tests are shown. 153

XV Experiment environments. 160

XVI Loop parallelization in PERFECT codes. % = percentage of total application execution time. DD Test = type of data dependence test required (CT = compile time, RT = run time, SE = simple logical expressions, IT = interval trees, UE = USR evaluation, LRPD = LRPD run time test) Priv = type of privatization required (A = array privatization). Red = type of reduction required. PB = pushback required. IP = loop contains subprogram calls. EX = execution type (IE = inspector/executor, SP = speculative execution). Intel = parallelized automatically by the Intel Compiler (version 9.0, *-parallel -par_threshold100*). 171

XVII Loop parallelization in SPEC codes. (Legend in Table XVI.) 172

LIST OF FIGURES

FIGURE		Page
1	Example of an input-sensitive memory reference pattern and corresponding code after parallelization.	5
2	Data dependences prevent parallelization. Privatization can eliminate storage related dependences.	9
3	Generic memory references through arrays. (a) different locations are referenced in each iteration respectively; no data flow is possible. (b) there is a data flow from each iteration i to iteration $i+1$ on array element $i+1$.	11
4	Organization of the compiler techniques needed for the automatic parallelization of loops for shared memory machines.	13
5	(a) Kernel to make matrix symmetric. (b) Geometric and (c) algebraic interpretations of the data dependence test for the *write* vs. *read* operations.	15
6	Run time parallelization based on instrumenting every memory reference.	23
7	Inspector / executor vs. speculative execution. (a) original code, (b) inspector/executor parallelization and (c) speculative parallelization. . .	24
8	Extraction of an independence predicate from an independence equation. The black nodes represent simple conditions and logical operations that are easier to evaluate at run time than it is to solve the original independence problem. (a) Original code. (b) Independence equation as intersection of *read* and *write* reference sets. $\cap$ and $\cup$ stand for set intersection and union respectively and # means predication. (c) The original problem was divided into two subproblems. $\wedge$ and $\vee$ stand for logical *and* and *or* respectively. (d) Intermediate result. (e) The final result is an accurate independence predicate which is inserted in the generated code (f) and that will be evaluated efficiently at run time.	29

FIGURE

9 Program model: (a) Sample program – Erathostenes' sieve. (b) Call Graph and Control Dependence Graphs (CD edges shown as solid lines). CD siblings are connected by dotted lines given by the post-dominance relation in the original Control Flow Graph. Subprogram call relations are shown as dashed lines. 32

10 USR intersection ($\cap$): (a) when the result is an LMAD and (b) when the result cannot be represented as an LMAD. 35

11 USR difference ($-$): (a) when the result is an LMAD and (b) when the result cannot be represented as an LMAD. 36

12 USR union ($\cup$): (a) when the result is an LMAD and (b) when the result cannot be represented as an LMAD. 37

13 USR expansion ($\otimes^{\cup}$): (a) when the result is an LMAD and (b) when the result cannot be represented as an LMAD. 38

14 USR gate (#): (a) when the result is an LMAD and (b) when the result cannot be represented as an LMAD. 39

15 USR translation ($USR \bowtie CallSite$): (a) when the result is an LMAD and (b) when the result cannot be represented as an LMAD. 41

16 USR formal definition. $\cap, \cup, -$ are elementary set operations: intersection, union, difference. $Gate\#USR$ represents reference set USR predicated by condition $Gate$. $\otimes^{\cup}_{i=1,n} USR(i)$ represents the union of reference sets $USR(i)$ across the iteration space $i = 1 : n$. $USR(formals) \bowtie Call\ Site$ represents the image of the generic reference set $USR(formals)$ instantiated at a particular call site. 42

17 Classification of references in straight line code. (a) Sample code. (b) MCA partitions. 45

18 MCA algorithm for successive statements. 46

19 Classification of references in conditional blocks. (a) Sample code. (b) MCA partitions. 47

20 MCA algorithm for mutually exclusive conditional blocks. 47

FIGURE

21 Classification of references in loops. (a) Sample code. (b) MCA partitions. 48

22 MCA algorithm for loops. (a) accurate but possibly slower, (b) approximative but faster. 48

23 Case study to compare the algorithms in Fig.22. (a) Sample code showing the Static Single Assignment numbers and ϕ functions for variable x. RW for the body of the outer loop using (b) the accurate algorithm and (c) the approximating algorithm. 49

24 Classification of references at subprogram call sites. (a) Sample code. (b) MCA partitions. 50

25 MCA algorithm for a subprogram call site. 50

26 Algorithm to compute the set of memory locations that carry cross iteration dependences (expressed as a USR). 51

27 PDAG formal definition. $\wedge$, $\vee$, $\neg$ are the elementary logical operators *and*, *or*, *not*. $\otimes_{i=1,n}^{\wedge} PDAG(i)$ holds true if and only if each of $PDAG(i)$ holds true, $i = 1, n$. $PDAG(formals) \bowtie Call\ Site$ represents the instantiation of a generic $PDAG$ at a particular call site. A specialized *library routine* may be employed to produce the value of the predicate. If a test based on simple comparisons and logical operations cannot be found, we fall back to a $reference\ based\ test$. . . . 53

28 **Algorithm Solve**: Extraction of a sufficient run time test as a PDAG from a dependence equation $D = \emptyset$. Details on the implementation of the subalgorithms are presented in the Appendix. We accumulate PDAGs in increasing order of complexity when the partial solutions are sufficient but not necessary, using the *logical or* operator $\vee$. 55

29 Algorithms to extract a PDAG from a USR identity based on USR syntax. 56

30 A Hybrid Analysis extreme: in general, no test can solve this problem faster than the reference-by-reference LRPD test. 57

31 Extraction of an independence predicate using approximation. 58

FIGURE

32 Algorithms to extract a PDAG from a USR identity based on USR approximation. 59

33 Example of a case where a sorting based test is more accurate than applying the *Solve* algorithm. 60

34 Example extracted from DYFESM, loop SOLVH_do20. The loop at line 2 can be executed in parallel if and only if there are no cross-iteration dependences on arrays W and A. 62

35 PDAG extraction from the parallelization problem for the loop at line 2 in Fig. 34 (partial PDAGs are shaded). The numeric labels represent dependence equations $DS = \emptyset$, where DS is the corresponding node. For instance, equation 2 in block (b) has as DS the dependence set for variable W. 63

36 PDAG extraction from the parallelization problem for the loop at line 2 in Fig. 34 (continued from Fig. 35). 65

37 (a) Scalar code, (b) scalar SSA form, (c) array code and (d) improper use of scalar SSA form for arrays. 66

38 (a) Sample code in Array SSA form (not all gates shown for simplicity). (b) Array SSA forms: (top) as proposed by *citeknobe.popl.98*, (center) with reduced accuracy and (bottom) using aggregated array regions. . . 67

39 Dynamic optimization through recompilation after specialization. (a) Sequential code. (b) Dynamic compilation through specialization, (c) when *ind(:)* is found to be a permutation, and (d) when *ind(:)* is found to contain repeating values. 69

40 Symbolic value analysis for comparison of addresses. 76

41 (a) Code sample, (b) in GSA after closed form substitution, (c) Value Evolution Graphs. 78

42 (a) Sample code in GSA, (b) VEG for f_1, f_2, f_3; VEG for p_1, p_2, p_3, p_4, p_5 – (c) before pruning, (d) after pruning based on GSA Paths, and (e) based on range tracing. 86

FIGURE

43 Aggregation of WF across lines 3-11 for this code snippet extracted from PERFECT/TRACK/EXTEND_do400. The **LMAD** column shows the cases in which the descriptor can be represented as an LMAD. The **Node** column lists the label of the node that roots the corresponding USR in the figure on the right. The last columns show LMAD-based under- and over-estimates (as sets) for each USR. 88

44 Details on how the VEG information is used. 91

45 The integration of the search for contiguous write-first sequences in the Memory Classification Algorithm. We have modified the abstract intepretation phase at loop header level. The *McaLoopBlock* algorithmn is presented in Fig. 22. 91

46 Increasing (a), contiguous (b), and consecutive (c) reference patterns in a loop. 92

47 Integration in the Hybrid Analysis framework. 93

48 Code extracted from *DKZMH_do60*. 100

49 Code extracted from *EXTEND_do400*. 101

50 Loop parallelization. (a) Original sequential loop. (b) After parallelization using OpenMP directives. 107

51 Run time loop parallelization. (a) Original sequential loop. (b) After parallelization using OpenMP directives. 108

52 Example of a loop that can be run in parallel after removing dependences through privatization and after reduction and pushback parallelization. 112

53 In order to parallelize the outer loop, privatization of array W may or may not be needed depending on the values of subscript arrays *begin* and *end*. 114

54 Privatization with copy-in. Only the ends of the array must be copied. . 116

FIGURE

55 In order to parallelize the outer loop, reduction parallelization on array W may or may not be needed depending on the values of subscript arrays *begin* and *end*. 118

56 In order to parallelize the outer loop, reduction parallelization on array W may or may not be needed depending on the values of subscript arrays *begin* and *end*. 119

57 Code extracted from loop MXMULT_do10 in benchmark application DYFESM (PERFECT suite) and schematic representation of the main data structure, array MX. 121

58 Dependence set as a USR for array MX in loop MXMULT_do10. Only RW vs. WF dependences shown. Triangle = intersection, inverted triangle = union, ellipse = recurrence, empty diamond = difference (second term designated by dotted line), diamond with conditional = gate, hexagon = translation across subprogram boundary, and rectangle = list of LMADs. 123

59 Sufficient PDAG consisting of only simple expressions extracted from the dependence test on array MX in loop MXMULT_do10. Ellipse = logical AND across an iteration space, triangle = logical AND, hexagon = translation across subprogram boundaries, rectangle = conditional expression. 124

60 Dependence set as a USR for array MX in loop MXMULT_do10. Only RW vs. RW dependences shown. They are the ones that can be removed by parallelizing the reduction operation. In addition to the symbol explanation given in Fig. 58, the dotted ellipse means a partial iteration space, in this case *1,2,...,iss-1*. 125

61 (a) Output dependence set descriptor as USR and (b) necessary and sufficient PDAG as call to a run time library. 127

62 Parallel code for loop MXMULT_do10. Variables *mxmult_do10_is_indep*, *mxmult_do10_mx_nopriv*, *mxmult_do10_mx_nored* and r_43 are precomputed before the loop. The call to *rtlmadi_..._copy_out_o_1* computes USR *r_57*, which is then used in the call to *usr_copy_out*. . 128

FIGURE

63 Code Restructuring. (a) Original code and (b) corresponding CDG with a nontrivial cycle. (c) After insertion of control variables *peFlag* and *peSite1*. (d) Corresponding CDG with only a trivial cycle and (e) code generated from the CDG (without any jump statements such as GOTO and RETURN). 133

64 Unification of COMMON structures disambiguates aliases. (a) Original code. (b) After common unification. 135

65 Unification of EQUIVALENCE-ed names disambiguates aliases. (a) Original code. (b) After equivalence unification. 136

66 Array bounds issues. 137

67 Lazy initialization pattern. 139

68 Access pattern that can be approximated using LMADs. 144

69 (a) Similar dependence tests for arrays A and B can be stored using a single PDAG. (b) The per-iteration MCA partition descriptors will appear in sub-PDAGs in dependence questions at both loop levels. 148

70 Cascade of sufficient run time tests in increasing order of complexity. . . . 154

71 SGI Altix 3700 system computational brick. 157

72 Intel Core Duo processor in an Apple MacBook notebook computer. . . . 158

73 Hybrid Analysis results on the Altix (PERFECT and Previous SPEC) and the O350 (SPEC2000) systems respectively. In the top graph, the white bars (4 processors CT) correspond to speedups obtained using only compile-time methods and measured on 4 processors. 163

74 Polaris/HA normalized execution times on a dual core processor. Sequential, parallel code executed on 1 and 2 threads respectively. All times are normalized to the sequential execution time. Intel Core Duo 1.83MHz, 2x256 MB RAM 5300 / Mac OSX Tiger 10.4.7, XCode 2.2.1, Intel Compiler 9.1.24 -openmp -O. ADM was compiled with *-O0* because the compilation with *-O* resulted in erroneous execution. 164

FIGURE

75 Comparison of Polaris/HA vs Intel Compiler. Speedup after automatic parallelization on a dual core processor vs. a run of the original application. 166

76 Speedup comparison between 2 processor runs using Polaris/HA on the SGI O350 and the IBM Toronto Lab parallelizing compiler on a Power machine. 167

77 Comparison of automatic parallelization using HA against SUIF Interprocedural Automatic Parallelizer: (a) Speedup on four processors, (b) Granularity as average duration of a parallelized section expressed as percentage of the execution time, and (c) Coverage as percentage of the execution time. 168

78 Parallelization example. (a) Original sequential code. (b) After automatic parallelization. 221

79 Parallelization report after the compile time phase of Hybrid Analysis for DYFESM. 223

80 Interprocedural analysis framework as collection of information in a bottom-up traversal of the program. The first argument to the polymorphic method *get_info* selects the correct code to process the given information. 226

CHAPTER I

INTRODUCTION

This dissertation presents new compiler technology for the hybrid (static and dynamic) analysis of memory reference patterns in programs and its application to the automatic translation of legacy sequential applications into equivalent multithreaded ones.

A. Parallel Computers

The timelines of computing and supercomputing have overlapped almost from the beginning. There was always a need for more performance than could be offered by the fastest computer. First large scientific problems, then military and business applications pressured manufacturers into creating more and more powerful machines. This led naturally to parallelism. More machines of the same type, working together, can solve the same problem faster and address larger problems.

Although first common only in dedicated large scale systems such as mainframes, parallelism has become mainstream recently with the widespread availability of multicore processors. In the 1990's, the main driver of system performance increase had been frequency scaling. Severely limited by overheating and the processor-memory gap, frequency scaling has been overtaken by parallelism over the past few years. Indeed, parallelism is now everywhere, from game devices and laptops to workstations to servers to supercomputers.

This dissertation follows the style of *IEEE Transactions on Parallel and Distributed Systems*.

B. Automatic Parallelization

In order to use parallel hardware efficiently, sequential programs must be divided into concurrent parts (parallelized). These parts must be synchronized so that the outcome of their execution is consistent with that of a sequential run. Parallelization can be performed either by the programmer or automatically in software or hardware.

The programmer can write explicitly parallel programs using parallel languages, parallel directives as extensions to a sequential language, or by using parallel libraries. Unfortunately, writing parallel applications explicitly has several drawbacks. First, there is a great amount of legacy sequential software which would have to be reengineered. Second, most programmers have not been trained to write parallel programs. Additionally, parallel software design tools are not as developed as their sequential counterparts, which results altogether in higher software development costs.

Automatic parallelization methods use hardware and software mechanisms to run sequential applications in parallel efficiently, without programmer assistance.

At the level of machine instructions, this problem was solved partially with the introduction of out-of-order processors, either static (explicitly parallel instruction computing) or dynamic (superscalar). While the static ones require a compiler to detect small sets of instructions that can be executed concurrently, the dynamic ones analyze and extract parallel instruction sets at run time. Unfortunately, the performance improvement of instruction level parallelism does not scale beyond a small constant factor dependent on the structure of the sequential program.

At the level of repetitive structures such as loops, parallelization performance can scale with the data set size. Since parallelization at this level requires analysis of a large window of instructions, it has been performed mostly in software, using compilers (though sometimes with hardware support). Loop parallelization as com-

piler optimization has been the subject of a large amount of research over the past few decades. However, the performance of the automatic parallelizers provided by parallel computer manufacturers is in most cases well below the opportunities offered by the hardware.

There are two important reasons why compiler analysis fails to extract efficient parallelism from sequential applications. First, the behavior of the application may be input dependent. The same loop may or may not be parallelizable depending on a value read from a file at run time. In such a case, the compiler must make the sometimes overly conservative decision of generating sequential code, even when the actual run time values would not prevent parallelization. Second, the compiler may lack the symbolic representation and analysis power to understand the behavior of the application with respect to parallelism, and thus fails to detect it even in input independent cases.

All parallelization problems can be solved at run time, when needed input values become available and when symbolic variables take numeric values. The first proposed run time parallelization methods were based on the instrumentation of virtually each memory access operation. Although accurate, these methods incur high run time overhead which can negate the benefits of parallelization. Moreover, run time methods based on instrumentation perform too much unnecessary analysis work. For instance, run time parallelization analysis for a loop with n iterations will incur $\Theta(n)$ overhead even when the parallelization decision does not depend on the iteration count, but rather on the loop bounds.

C. Hybrid Analysis

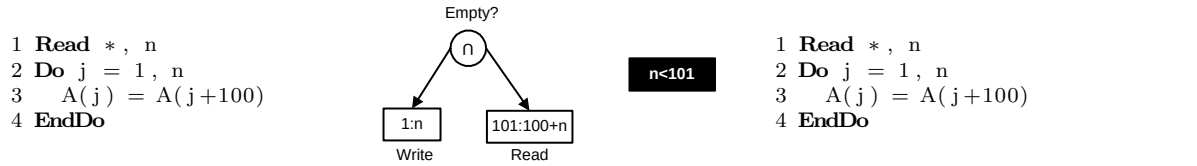
As we have discussed over the previous section, run time techniques are crucial to the effectiveness of automatic parallelization, but they often perform unnecessary work resulting in unnecessary overhead. It is thus imperative to design dynamic analysis methods whose run time overhead is proportional only to the number of variables that affect the parallelization decision. If the decision to parallelize a loop with n iterations depends only of the values of its bounds, a run time test should cause $\Theta(1)$ overhead, and not $\Theta(n)$.

Hybrid Analysis represents the process of extracting at compile time sufficient conditions for optimizing transformations that could not be verified statically, and of validating them at run time in the presence of actual values. This dissertation presents a general framework for the Hybrid Analysis (HA) of memory reference patterns and its application to parallelization – Hybrid Dependence Analysis (HDA). Hybrid Analysis can also be applied to compiler optimization problems beyond parallelization, such as constant propagation or liveness analysis.

Let us illustrate the way Hybrid Analysis works through an example.[1] Consider the loop in Fig. 1. In order to generate code for its parallel execution, the compiler must prove that the set of *read* references [101:100+n] and the set of *write* references [1:n] are disjoint, i.e., that their intersection is empty.

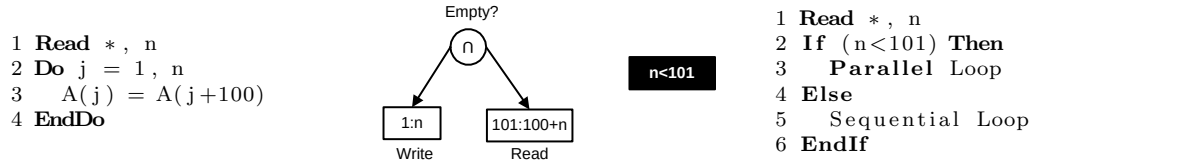
An accurate compile time parallelization decision based on symbolic calculus cannot be made, because the validity of the decision depends on the input value n, which is not known before run time. The conservative static decision (a) would thus

[1]This particular example presents a very simple dependence problem and is discussed here for illustration purposes only. The actual cases that can be handled by Hybrid Analysis are much more complex, as will be shown over the remainder of this chapter.



(a) Compile time overly conservative decision: do not execute in parallel.

(b) Run time parallelization based on instrumenting every memory reference.



(c) Hybrid Analysis: extract condition at compile time, evaluate it at run time.

Fig. 1. Example of an input-sensitive memory reference pattern and corresponding code after parallelization.

be not to execute the loop in parallel.

Run time analysis methods [1] take a different approach (b). They instrument every dynamic memory reference and then make optimization decisions at run time based on dynamic analysis of the produced trace. Although accurate, they often perform a large amount of unnecessary work, since their overhead complexity is proportional to the number of memory operations and sometimes data set size of the program.

Let us observe that in the case in Fig. 1 (a), condition ($n < 101$) is sufficient to prove the *read* and *write* sets disjoint. Moreover, static analysis may have already extracted this condition but could not use it for a decision because it contained input value n. On the other hand, the instrumentation based run time analysis (b) ignores

this partial symbolic result and ends up performing much more work than necessary.

Hybrid Analysis (c) combines the advantages of static and dynamic methods. It starts at compile time by performing symbolic calculus and extracts conditions under which certain optimization transformations are legal. At run time, it evaluates these optimization correctness predicates and switches on the optimization when they hold true. Optimization based on Hybrid Analysis has the same applicability as any dynamic optimization method, but in general most of the work is performed at compile time using scalable, symbolic calculus methods. In other words, the complexity of Hybrid Analysis is often independent of the number of dynamic operations or data set size of the program.

D. Contribution

We believe that this dissertation makes the following contributions:

- It introduces Hybrid Analysis, a novel compiler technique that bridges seamlessly static and dynamic analysis of memory reference patterns, and which has been applied succesfully to dependence analysis and Array SSA.
- It presents a new representation for sets of memory references, the Uniform Set of References (USR), that relies on partial symbolic aggregation to reduce the complexity of associated optimization problems by orders of magnitude.
- It presents an implementation of the Hybrid Analysis framework and its application to parallelization, Hybrid Dependence Analysis, in a research compiler. The implementation and experimental results prove that automatic parallelization works for scientific applications. Our techniques have been fully implemented in the Polaris compiler and resulted in whole program speedups of at least 2 on 4 processors on 18 out of 22 industry standard benchmark applications.

E. Organization

Chapter II introduces the fundamentals and discusses the state of the art in compiler-based automatic parallelization. Chapter III presents the memory reference analysis framework and a generic hybrid data dependence analysis method, which are crucial to implementing an automatic parallelization tool. Chapter IV presents the lower level symbolic analysis of scalar values, which is used throughout the memory reference analysis framework. Chapter V describes the engineering of the automatic parallelization tool and presents a detailed case study. Chapter VI discusses related compiler implementation issues. Chapter VII presents a comprehensive evaluation of our hybrid optimization techniques. Chapter VIII summarizes the contributions of the dissertation and discusses future research on Hybrid Analysis. A user manual and a developer/reference manual are available as the first and second appendices respectively. Partial results of this dissertation have been published in refereed workshop and conference proceedings and journals [2, 3, 4, 5, 6, 7].

CHAPTER II

FUNDAMENTALS AND PREVIOUS WORK

A. Fundamentals of Automatic Parallelization

Automatic parallelization represents the algorithmic transformation of a sequential program into an equivalent parallel counterpart. Efficient automatic parallelization is conditioned by three factors: a small number of synchronization points (high granularity), an even distribution of work among different threads (load balancing), and a good affinity of data to processing units (data locality).

This dissertation focuses on the detection of loop level parallelism at high granularity levels, which translates into a low number of synchronization points. Although our results could be improved by addressing load balancing and locality issues, our techniques precondition all such optimizations and have been successful on their own to the efficient parallelization of a large number of applications. We are thus focusing on the automatic parallelization of large loops, possibly spanning multiple subprograms and possibly containing complex control structures.

1. Scalar Data Flow and Data Dependence

Fig. 2 presents three sequential code fragments (column 1). Let us assume that, for each case, we want to execute in parallel the two statements enclosed in a rectangle. When the two statements are executed on different threads, there are no guarantees of relative ordering between them. In the example on the first row, the assignment of 8 to X may happen on thread 2 after the value stored in X is used on thread 1. In this case, thread 1 will wrongfully use value 5 stored previously in X. There is a fundamental producer-consumer relation between these two statements. Such a

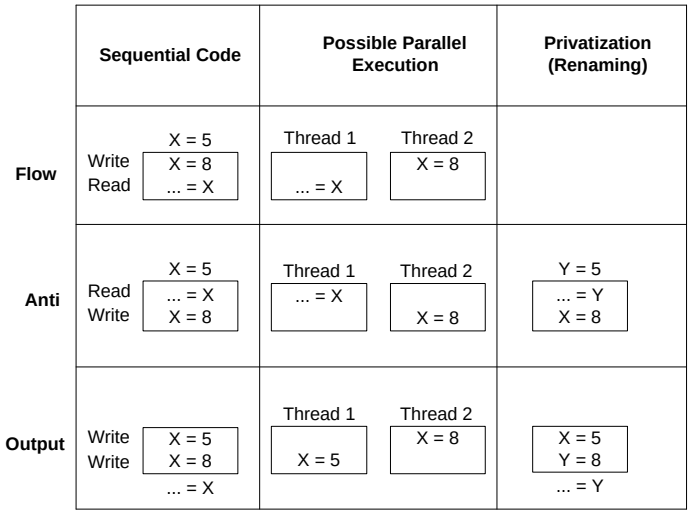


Fig. 2. Data dependences prevent parallelization. Privatization can eliminate storage related dependences.

relation cannot be broken, because a value cannot be consumed before it is produced, thus the two statements cannot be executed in parallel. The statements are said to be *flow* dependent. This relation is also known in literature as a RAW (read after write) dependence or race condition.

The second row presents a different situation. The two statements selected for parallelization do not share any values. In the sequential program the value of X in the first two statements is 5, and in the third one it is 8. Statements 2 and 3 are therefore not flow dependent. However, the second column presents a scenario in which the result of the parallel program is different from the sequential one. This consequence is caused by the fact that the two flows of values 5 and 8 share a single memory location. The two operations are said to be *anti* dependent. This relation is also known as WAR (write after read). By using an additional memory location Y for the first two references to X, we can separate the flow of data into two, one using X and the other using Y. These two flows can then be carried concurrently by separate threads. We will use the terms *renaming* and *privatization* interchangeably

to denote the transformation that removes storage related dependences. The first term is used more with respect to Instruction Level Parallelism while the second is generally associated with Thread Level Parallelism, where the dependence is usually removed by creating *private* versions of the conflicting storage object for each thread.

The third row presents a similar scenario, although here the storage related dependence is between two *write* operations at the same memory location. This type of data dependence is known as *output* or WAW (write after write). Analogous to *anti* dependences, *output* dependences can be removed through renaming/privatization

The last class of data dependences is known as *input* or RAR (read after read). These dependences are not important from the point of view of the correctness of parallelization for shared memory machines, because data access is handled by the hardware. However, input dependences are important to the study of data locality as well as to the automatic generation of communication primitives in distributed memory systems. They will not be discussed further in this dissertation.

2. Array Data Flow and Data Dependence

The analysis of scalar data flow and data dependence is crucial to extracting Instruction Level Parallelism (ILP). However, the performance gain of ILP does not scale with the data set. Thread Level Parallelism can achieve scalable performance by employing a larger number of computation resources, e.g. more processors. A larger data set can thus be processed in the same time budget by allocating more hardware execution threads, while a fixed size problem can be solved faster. In order to achieve this scalability, the compiler must be able to understand how large data structures are referenced. In general, scalable programs reference memory through parameterized objects named containers. The most used containers are *arrays*, which are collections of fixed size objects stored consecutively in memory. In a compiler, each element of an

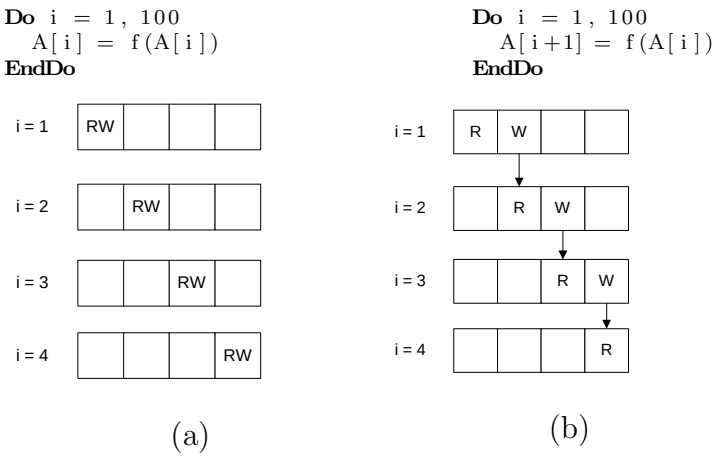


Fig. 3. Generic memory references through arrays. (a) different locations are referenced in each iteration respectively; no data flow is possible. (b) there is a data flow from each iteration i to iteration $i+1$ on array element $i+1$.

array is identified by the array name and its position in the array (array subscript). The subscript can be either a single integer or an expression describing a point in a multidimensional, rectangular, integral coordinate space, which can be linearized to describe a memory location relative to the beginning of the array.

Fig. 3 presents two examples of accessing memory through arrays using loops. The array subscript formula is in general a symbolic expression that contains the loop index, in this particular case i and $i+1$. In the first example (a), the only location in array A referenced in some iteration i is at offset i. Therefore, the locations referenced in two different iterations will always be different. This means there cannot exist data flow between the operations in any two different iterations, thus any two iterations can be executed in parallel concurrently. The loop is said to be *parallelizable*.

On the contrary, in the second example (b), each iteration i writes location $i+1$ which will then be read in iteration $i+1$. There is thus data flow from the operation in each iteration i to the following iteration. This data flow imposes a strict execution order on iterations: 1, 2, ..., 100. In other words, the loop cannot be *parallelized*.

When arrays are referenced using subscript expressions that are affine combination of the loop index, and when the loop bounds are known at compile time, compilers can make the decision on whether a loop can or cannot be parallelized. The analysis is based on the formulas of the subscripts of significant references, e.g., *read* vs *write*. Dependence relations are represented symbolically, e.g., $i \rightarrow i + 1$. However, in other cases the necessary information is not available at compile time either because it is input dependent or because the compiler cannot perform complex symbolic calculus. In such cases, decisions can still be made at run time after instrumenting all memory references and building the unfolded, dynamic data dependence relation table.

B. Current State of the Art in Automatic Parallelization

A parallelizing compiler has two main components (Fig. 4). In the analysis phase, the compiler identifies data dependence relations among operations in different iterations. In the transformation phase, it modifies the code to generate and manage a number of parallel threads.

1. Compiler Analysis

a. Data Flow Analysis

There has been a very large amount of research on the analysis of the flow of information (data) in programs. In addition to parallelization, data flow knowledge is crucial to several other optimization techniques such as register allocation, constant propagation, common subexpression elimination, checkpoint size reduction or dead code elimination.

Data flow analysis for scalars has been performed traditionally using monotone dataflow iterative methods [8] and lately using the Static Single Assignment program

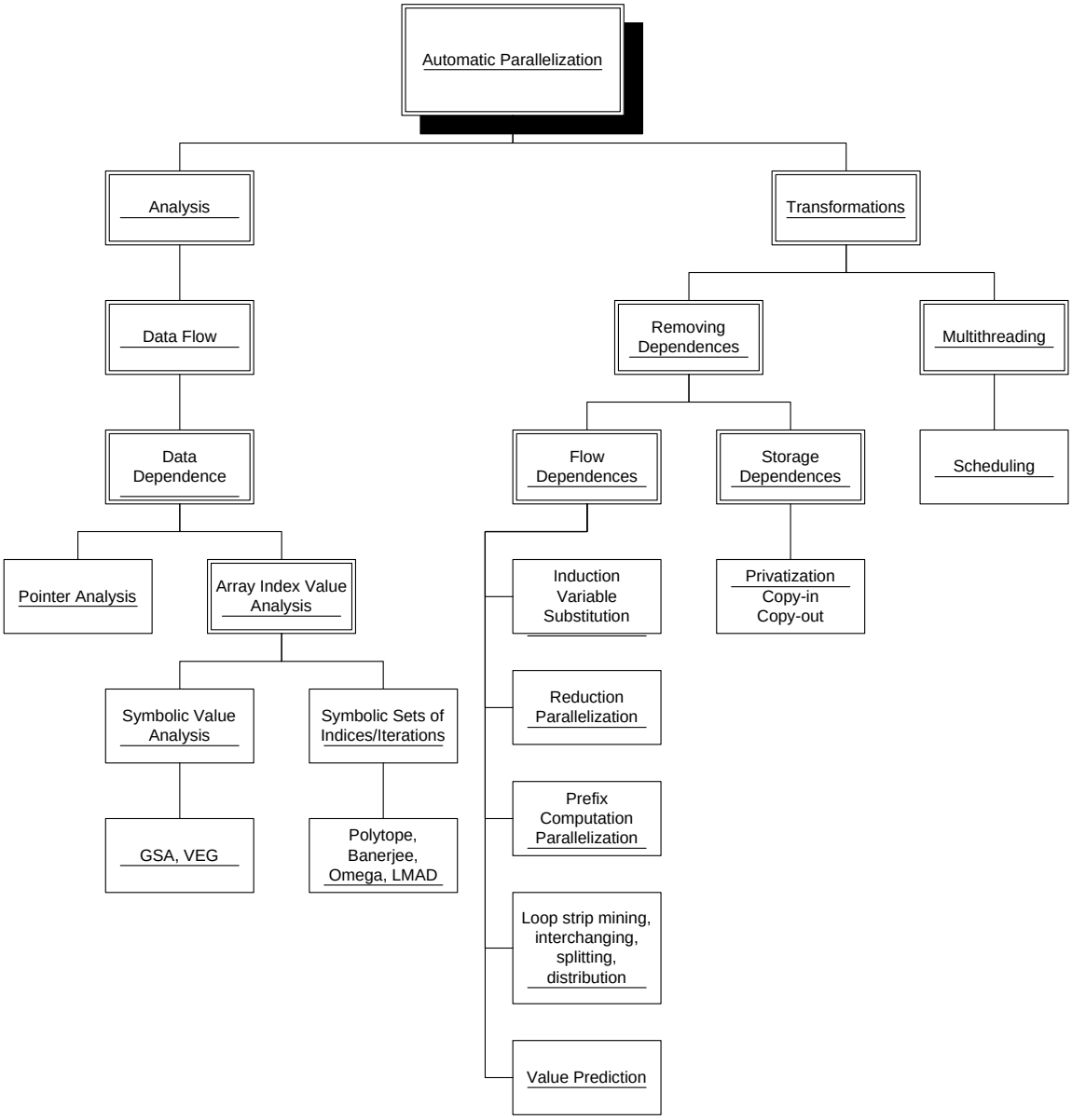


Fig. 4. Organization of the compiler techniques needed for the automatic parallelization of loops for shared memory machines.

representation [9] or similar representations [10, 11, 12].

When memory is referenced through arrays, data flow relations can be computed by analyzing the symbolic subscript values. Individual symbolic values can be compared using value range dictionaries generally based on abstract interpretation of arithmetic operations, control flow and recurrences. Most nontrivial programs contain loop nests possibly spanning several subprograms and containing complex control flow. When the subscripts, inner loop bounds and control predicates are affine combinations of loop invariants and outer loop bounds and indices, data flow equations can be formulated as linear integer programming problems.

Array data flow relations have been represented in several ways. One way is to partition, for a given program context, all memory locations referenced within that context, into RO (read only), WF (write first) and RW (read write). Each set in the partition can be represented using triplet-based representations such as the linear memory access descriptor (LMAD) [13, 14, 15] or gated array region (GAR) [16], or by using linear constraint sets [17] or Presburger formulas [18]. Similar to the RO/WF/RW partition, there are several other equivalent classifications based on whether an operation *may* or *must* modify the data at a particular memory location. Other representations of data flow are Last Write Trees [19] and Array SSA [20], which maintain explicit def-use edges.

Most data flow analysis methods described in literature are interprocedural and rely on interval analysis to summarize the effect of larger and larger program contexts [21, 22, 23, 24, 25, 26, 27, 28]. They compute data flow relations expressed using sets of references. [13, 14, 15] compute RO/WF/RW partitions, [16, 29] use GARs, [30, 19] compute Last Write Trees, and [17, 31, 32, 33, 34] use linear constraint sets.

[35] computes the flow between data referenced by nonlinear index expressions by defining and identifying relevant properties of index arrays, such as closed form

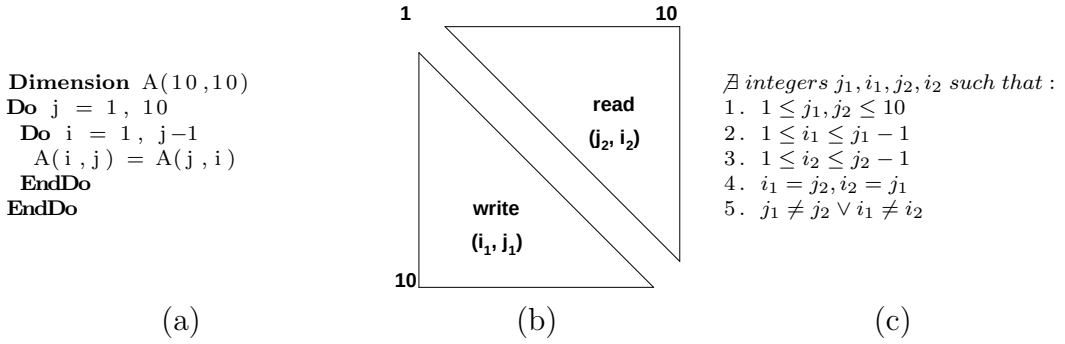


Fig. 5. (a) Kernel to make matrix symmetric. (b) Geometric and (c) algebraic interpretations of the data dependence test for the *write* vs. *read* operations.

values or distance. [36] presents data flow analysis for parallel programs. [37] presents instance-wise iterative array dataflow analysis based on constraint sets. [38] estimates the probability of dataflow edges. [39, 34] present a good discussion of array dataflow work in general.

b. Data Dependence Analysis

Several representations have been proposed to detect or measure data dependences between operations in different iterations of a loop. One way is to measure, for each pair of iterations, the set of memory locations (as LMAD, GAR, linear constraints set etc) on which there are data dependences between the two iterations. For loop nests that access memory in a regular way we can measure the distance as a number of iterations between two dependent operations [40, 41]. [42] presents an application that displays interactively dependence information in human readable form.

Most data dependence tests have a geometric interpretation [43]. They take the subscript space corresponding to two operations in two different iterations and

prove them disjoint. When subscripts are affine combinations of loop indices and loop invariants, the subscript spaces appear as polyhedra in an n-dimensional Euclidian space. Since the subscripts have integer coordinates, only the points with integer coordinates are of interest.

Fig. 5(a) shows a kernel that makes matrix A symmetric by copying the upper diagonal elements to their mirror lower diagonal position. A geometric interpretation of the test (b) can be given by associating integer points in a 2-dimensional space with either the iteration vector of each operation or with the address of the associated memory reference. We can see that the set of points associated with *read* references is disjoint from the set of points associated with *write* references. The corresponding algebraic form (c) can be used to prove disjointness automatically. Each linear constraint describes either a half space (inequations) or a hyperplane (equations). The dependence test proves, using symbolic calculus, that there are no points of integer coordinates in the intersection of these half spaces and hyperplanes.

In conclusion, proving data independence reduces to proving that the intersection of two n-dimensional bodies does not contain any Diophantine (integer coordinates) points. This problem is NP-hard when subscripts are affine and apparently undecidable when subscript expressions are arbitrarily complex. However, in most cases memory is referenced in a simple way, and several models of reduced complexity have proved to cover many practical cases.

The first dependence tests consisted mostly of bound checks and elementary number theory results such as the GCD test [8]. These tests are very simple, thus inexpensive, but cannot handle complex cases. Most later tests rely on Fourier-Motzkin variable elimination (FMVE) to reduce the number of coupled constraints repeatedly. After all variables have been eliminated, the remaining identity may be a tautology (dependences exist) or a contradiction (no dependences). However, variable

Table I. Static dependence tests.

Test name	**Description**
SIV, ZIV, MIV, Delta Test [45]	Constraint propagation for coupled subscripts.
GCD Test[8, 46, 47]	Greatest common divisor test.
Lambda Test [48]	Constraint sets (including coupled subscripts).
Power Test[49]	Fourier-Motzkin variable elimination (FMVE).
SVCT [50]	Single variable per constraint test.
Acyclic Test [50]	Acyclic elimination graph.
SLRT [50]	Simple loop residue test.
Banerjee Test[8, 46, 47]	Approximation, ignores Dophantine requirement.
Symbolic Banerjee Test[39]	Symbolic Banerjee test.
Integer Programming [43, 50, 51]	Fourier-Motzkin variable elimination.
(V)I-test [47, 52, 44]	Polynomial time Banerjee with integers.
Omega Test[18, 53]	Presburger algebra, FMVE with integers.
Range Test [54, 55, 56, 57]	Value range-based nonlinear.
ART Test [58, 14]	Access region test.
Commutativity Test [59]	Find operations that can happen in any order.
Container semantics [60, 61, 62, 63]	Detection of parallelizable container operations.
Index property [64, 35, 65]	Subscript properties, e.g. closed form value.
Monotonicity [66, 67, 68, 55, 69]	Based on monotonicity of the subscript.
Shape/Traversal [70, 71, 72, 73]	Shape/traversal analysis of linked structures.
Pattern matching [74, 75, 76, 77]	Recognition/substitution of a reference pattern.
MHP [78]	May happen in parallel analysis for Java.

elimination may result in loss of information. The Banerjee test [8] returns success (independence) when the intersection of the constraint spaces is completely empty. There are cases when, although the intersection is not empty, it does not contain any points of integer coordinates. In such cases the Banerjee test will report dependences when in actuality there are none. The (V)I-test [44] and the Omega test [18] take into account these issues. The Omega test is the only one that can solve any dependence equation with affine constraints in which the loop bounds are known. Most of these tests also handle cases when the loop bounds and other invariant values are symbolic (but comparable at compile time).

Table I presents a list of the common data dependence tests or classes of tests,

together with a brief explanation and some references to papers where the tests are described and/or compared against others. Here is a list of interesting comparisons and overviews: [79] - comparison of ddtests and their impact as seen statically and dynamically; [80] - summary of constraint-based data dependence analysis; [81] - comparison of I-test and Omega, comparison of Banerjee and FMVE; [82] - evaluation of several dependence tests; [58] - comparison of ART test against GCD, extreme value, FMVE, generalized GCD, Power, Lambda, I-test, Delta, Range; and [83] - dependence test and parallelization results for PERFECT codes using Omega, I-test and Banerjee.

None of the existent symbolic tests can solve arbitrarily complex dependence equations, even when all needed values are available at compile time.

c. Analysis of Array References

Most of the static analysis methods presented above work with sets of array references (addresses) or with sets of iterations. In both cases, they are integer numbers which are generally organized in sets. Since in most cases subscripts are affine, the sets of references or iterations are organized as polyhedra. The LMAD ([13, 14, 15]) and GAR ([16]) represent polyhedra by listing an initial starting point plus a stride and a span in each dimension of the space. Linear constraints [17] describe the half-spaces that bound the polyhedra and are in general more accurate than the triplet-based representations when polyhedra facets are not orthogonal. However, the polyhedra described by linear constraint sets must be scanned [84] in order to describe all the points within, whereas the triplet-based representations can be translated to Fortran or C code directly. Presburger formulas [18] are logical forms that include linear constraint sets and quantification operators $\forall$ and $\exists$.

Certain classes of non-array references can be converted into arrays, or can be

reasoned with as if they were array references. [60, 61, 62, 63] showed that containers in C++ and Java behave under certain circumstances like arrays. [85] showed that even machine code references can be pattern-matched into linear constraint sets. [86] showed how some classes of pointer-based references can be transformed into array references.

The analysis of arrays in loops generally reduces to analyzing their subscripts. They depend on either values produced by recurrences or on values stored in indirection (subscript) arrays.

The first set of analysis methods try to model recurrences in order to extract closed forms for the n-th term of a recurrence. The analysis of recurrences was presented [87] as abstract interpretation, as [88, 89] cycle detection on graphs, and [90] using the inverted chain of recurrences formulas.

When a closed form cannot be extracted, these methods extract closed forms for a *property* rather than a value, specifically for properties that must be checked by a specific optimization technique, such as parallelization. [66, 91] present data flow and data dependence analysis based on the monotonicity of recurrence values with applications to the parallelization of recursive subprograms. [67, 68] present similar data dependence tests based on monotonic value evolutions. [64, 35] present a set of subscript properties that are relevant to parallelization and can be checked automatically under certain circumstances: injectivity, monotonicity, closed-form value, closed-form bounds. [55] presents value-based Java data dependence analysis using index equivalence, additive constant difference and inequality graphs.

The subscript property that received most attention is *value range*. Even when two values cannot be compared directly, they can be proven distinct by proving that their possible value ranges are disjoint. [54, 92, 57, 56] present value range analysis and its applications to data dependence analysis.

[39] presents a comprehensive view of symbolic analysis for optimization (especially parallelization), including value analysis. [93] presents backwards, on demand value analysis using GSA. [94] presents a method for the discovery of regular strides in pointer programs by profiling, and presents applications to prefetching.

2. Compiler Transformations

In general, the parallelization of sequential code on a shared memory system is achieved by inserting explicit calls to multithreading routines that create threads and keep them synchronized when needed. When parallelizing for a machine with distributed memory, the communication of data between threads is also managed through calls to a runtime library. From a correctness perspective, the generation of multithreaded code is straightforward once the necessary data dependence information has been computed. In addition to proving independence, Hybrid Analysis information can be used to detect cases when dependences can be removed through a code transformation.

a. Removing Flow Dependences

In many cases, although there is data flow between different iterations, the code can be transformed into an (quasi-)equivalent parallelizable one. Although order-1 recurrences have data flow from each iteration to the next, some classes of recurrences have closed forms, i.e., algebraic formulas for the n-th term. By replacing all uses of the recurrence term with its closed form, the data flow between iterations is eliminated completely (induction variable substitution) [8].

Reductions are operations of form $X = X\ op\ exp$, where exp is an expression that does not reference X. When a variable is referenced only through reduction operations, the implicit flow dependences can be removed by substituting the algorithm with a

parallel counterpart (reduction parallelization) [95]. Other patterns can be recognized and parallelized, such as prefix computation [96, 75].

Several loop transformations have been devised to modify a loop nest so that a particular level in the nest can be parallelized: loop strip mining, interchanging, peeling, splitting, distribution [97, 98, 40, 99, 100, 101, 102, 103, 104, 39].

A data flow can be broken speculatively by guessing the values of the data [105, 106, 107]. This method seems to work best with linear recurrences controlled by predicate arrays that are almost entirely true or almost entirely false. In such cases closed forms are used speculatively.

b. Removing Storage Related Dependences

In most nontrivial programs, memory is reused across computation threads that do not share information. For instance, it is a common practice to use temporary variables whose liveness range is included in a loop body. There cannot exist a flow of information across iterations through these temporaries. However, the loop cannot be executed in parallel because each iteration will define and use the data at the same memory locations. The compiler can *privatize*, in other words *rename* the memory locations for each iteration, thus disambiguating the data flow [108, 102, 109, 19, 93, 110, 29, 111, 112].

c. Results in Automatic Parallelization

[97] presents testing of vectorization capabilities, including a compile-time/run-time hybrid approach based on validity predicates. [113] describes the automatic parallelization of four PERFECT benchmark codes. [114] gives a general parallelization overview in Polaris. [16] presents automatic parallelization based on GAR summaries. [115, 116] present SUIF best parallelization results with interprocedural analysis. [117]

present parallel performance enhancement techniques (such as load balancing) based on post-mortem analysis. [39] presents a comprehensive set of anlyses and transformations for parallelization. [118] discusses parallelization artifacts that lead to false sharing. [119] presents transformations for large granularity, good decomposition, vectorization, and locality. [120] presents practical parallelization performance issues (beyond data dependence analysis) and show results on several PERFECT benchmarks and other codes. [121] discusses compiler requirements for an automatic parallelization of all PERFECT benchmarks. [122] offers an empirical evaluation of three parallelizing compilers. [91] discusses parallelization of recursive programs. [123, 124] present parallelization, locality improvement and reduction of false sharing. [125] presents automatic parallelization results in a commercial compiler. [126] presents performance results for automatic parallelization among other optimization techniques. [127] presents challenges to automatic parallelization for DSPs, with no caches, multiple address spaces and direct memory access.

3. Run Time Parallelization Techniques

Dependence relations cannot be analyzed at compile time when they depend on input values or when the necessary representation/analysis methods are too complex. Several run time analysis methods have been proposed to perform data dependence tests at run time, in the presence of necessary actual values read from input or computed during program execution.

a. Instrumentation of Memory References

[128, 129, 130, 131, 132, 1, 133] present the Lazy Reduction Privatizing Doall test. The LRPD test consists of two phases. In the *marking* phase, each memory

```
1 Read *, n
2 Do j = 1, n
3   A(ind1(j)) = A(ind2(j))
4 EndDo
```

```
1  Read *, n
2  Do j = 1, n
3    Call MarkRead(ind2(j), trace)
4    Call MarkWrite(ind1(j), trace)
5  EndDo
6  isParallel = Analyze(trace)
7  If (isParallel)
8    Parallel Loop
9  Else
10   Sequential Loop
11 EndIf
```

(a) Instrumentation of each reference is necessary.

```
1 Read *, n
2 Do j = 1, n
3   A(j) = A(j+100)
4 EndDo
```

```
1  Read *, n
2  Do j = 1, n
3    Call MarkRead(j+100, trace)
4    Call MarkWrite(j, trace)
5  EndDo
6  isParallel = Analyze(trace)
7  If (isParallel)
8    Parallel Loop
9  Else
10   Sequential Loop
11 EndIf
```

(b) Instrumentation of each reference is not necessary. A simple loop bound check is sufficient.

Fig. 6. Run time parallelization based on instrumenting every memory reference.

reference[1] is recorded in a shadow data structure. In the *analysis* phase, the shadow data structure is processed to extract dependence information. Fig. 6(a) presents a scenario where marking each memory reference is necessary because every reference is made through indirection. In this case, LRPD is optimal in the sense that independence cannot be decided with less run time overhead. The LRPD can be applied speculatively, or using an inspector/executor strategy (Fig. 7).

[134] presents a test similar to LRPD used to reduce the overhead of dependence profiling for speculative low-level speculative parallelization, i.e. to estimate dependence probabilities. Several speculation methods [135, 136, 107] use a variety of tests conceptually similar to LRPD to validate the data flow or data dependence relations on which they speculate.

[1]Although overhead can be reduced by a constant factor [133], the asymptotic complexity remains proportional to the number of dynamic memory references.

```
1 Read *, n
2 Do j = 1, n
3   A(ind1(j)) =
      A(ind2(j))
4 EndDo
```

(a)

```
1 Read *, n
2 Do j = 1, n
3   Call MarkRead(ind2(j),
                  trace)
4   Call MarkWrite(ind1(j),
                   trace)
5 EndDo
6 isParallel = Analyze(trace)
7 If (isParallel)
8   Parallel Loop
9 Else
10  Sequential Loop
11 EndIf
```

(b)

```
1 Read *, n
2 Call copy(A, saveA)
3 Do j = 1, n
4   Call MarkRead(ind2(j),
                  trace)
5   Call MarkWrite(ind1(j),
                   trace)
6   A(ind1(j)) = A(ind2(j))
7 EndDo
8 isParallel = Analyze(trace)
9 If (NOT isParallel)
10  Call copy(saveA, A)
11  Sequential Loop
12 EndIf
```

(c)

Fig. 7. Inspector / executor vs. speculative execution. (a) original code, (b) inspector/executor parallelization and (c) speculative parallelization.

[94, 136] present the use of profiling to measure the regularity of memory references, with application to prefetching and speculative parallelization respectively. Both methods use the profiling information only as a profitability guide and resort to other checkers for correctness.

b. Optimization Predicate Extraction

The effectiveness of the run time tests based on instrumentation of virtually all memory reference has been limited by their inherent overhead, even when optimized for scalability. Fig. 6(b) presents a case in which LRPD would perform a large number $\Theta(n)$ of markings. However, in this case a simple check of the array and loop bounds would suffice to determine independence.

The alternative is to extract, at compile time, the validity predicates of the optimizing transformation. These predicates can then be verified at run time, usually with significantly less overhead. Vectorizing compilers had introduced [97] simple run time methods to decide when it is profitable to vectorize, e.g., a test on the length of the vector. In [31, 32, 53, 137, 20, 138] the authors had recognized this need to

bridge compile-time and run time analysis.

[31, 32] present the extraction of predicates under which a data flow or data dependence relation will hold true, which translates into less expensive run time tests. Their methods can solve the problem in Fig. 6(b). However, their solutions did not go far enough for significant impact in automatic parallelization. Their method cannot extract predicates when there is a variable number of compile time unknowns, such as indirection arrays or arrays of control variables.

[139] showed how sufficient predicates can be extracted by simplifying Presburger formulas with uninterpreted function symbols. [140] showed how uninterpreted symbols can be used to hide unimportant aspects that appear to, but do not prevent optimization. Although these two approaches are the closest to Hybrid Analysis, we could not extract their exact mathematical formulations so we could not compare their theoretical foundations. Also, these two approaches were not implemented fully in a run time optimizer or parallelizer so there are no empirical proofs of their applicability and effectiveness. We present in this dissertation extensive proof of the applicability and effectiveness of Hybrid Analysis.

c. Partially Parallel Loops and Communication Schedules

Partially parallel loops and communication schedules is not the focus of this dissertation. However, they have been the target of dynamic analysis methods and the techniques we propose here can be applied to these problems as well.

[141, 142, 143, 144, 145, 146] present the inspector/executor scheme for dynamic computation of the communication schedule including schedule reuse and other improvements such as dependence uniformization for regular schedules.

[147, 148, 149, 150, 151, 152] present partial redundancy elimination and aggregation analysis for interprocedural movement of communication primitives and

overlapping of communication and I/O with computation.

[153] presents speculative parallel thread spawning for the superthreaded architecture. [154] presents parallelization of partially parallel loops using dynamic scheduling. [155] introduces the R-LRPD dependence test and cascaded execution of partially parallel loops.

d. Inspector Executor vs. Speculative Optimization

From the point of view of the execution of optimized code, there are two main strategies (Fig. 7). The inspector/executor method validates the optimizing transformation and runs the optimized code only when correct. The speculative execution strategy first makes copies of data subjected to side effects and then runs the optimized code. If the speculative assumption is proved wrong, the original data is restored and the nonspeculative code version is executed instead. Although speculation may impose a large overhead in case of repeated failure, when successful it can be more profitable than the inspector/executor method in the case when the inspector duplicates a significant amount of the original loop. Also, in certain cases speculation is the only viable method because the validity of the transformation cannot be evaluated until the optimized code has finished executing.

[141, 142, 143, 144, 145, 146] present the inspector/executor scheme with schedule reuse and other improvements such as dependence uniformization for regular schedules. [128, 129, 130, 131, 132, 1, 105, 133, 155, 156, 136, 135, 107, 157] present speculation for thread level parallelization, mostly at loop level. [158] discusses the choice between inspector/executor and speculative execution.

Hybrid Analysis uses both the inspector/executor and the speculative execution strategies based on a cost estimation model.

CHAPTER III

HYBRID MEMORY REFERENCE ANALYSIS

The analysis of memory references is essential to many optimizations: parallelization, locality improvement, and data flow related transformations such as constant propagation or memory exclusion.

Static memory reference analysis relies on a symbolic representation of the expressions that make up the addresses at which memory is accessed. The analysis generally consists of comparisons between symbolic addresses. For instance, in the example in Fig. 1, in order to decide whether the loop can be executed in parallel, we need to prove that $j_1 \neq j_2 + 100, \forall j_1, j_2 \in 1..n$. Two symbolic expressions cannot always be compared because their values may depend on input data which are only known at run time. In this case all subsequent optimization must be conservatively dismissed. However, in many cases the actual addresses turn out to satisfy the assumption of the optimization. For instance, the loop in Fig. 1 cannot be parallelized because, when $n = 101$, there are $j_1 = 101, j_2 = 1, j_1, j_2 \in 1..n$, and $j_1 = j_2 + 100$. However, it may turn out at run time that $n \leq 100$, in which case the loop could have been parallelized.

Dynamic memory reference analysis consists of recording, at run time, the actual value of each accessed memory address. These values are then used to validate optimization assumptions. For instance, in the example in Fig. 1, we can record at run time all the values taken by address expressions j and $j + 100$. In order to find out whether the loop can be run in parallel, we just need to prove the resulting address sets disjoint. This can be done using an *always on* run time test. Generally, the implementation of a run time optimization uses code versioning. An optimized code version is produced at compile time and invoked at run time when all necessary

assumptions hold true.

In general, static analysis is preferred because it uses time and memory proportional to the size of the program. In contrast, although always applicable, dynamic analysis incurs a run time cost proportional to the dynamic number of individual memory references. However, in the example in Fig. 1, checking every dynamic memory reference is unnecessary. We can see that a sufficient condition for parallelization would be $n \leq 100$.

This section presents representations and techniques for hybrid memory references analysis. The analysis process starts at compile time, based on symbolical calculus. When optimization decisions are not reached at compile time, hybrid analysis extracts conditions which can validate them at run time, in the presence of actual values, but with much less overhead than pure dynamic methods. The border between what occurs at compile time and what occurs at run-time depends to a large extent on the power of current compiler algorithms and, with their continuous improvement, can be smoothly shifted towards better performance and less overhead.

A. An Overview of Hybrid Analysis Applied to Parallelization

Hybrid Analysis combines the advantages of static and dynamic methods. It starts at compile time by performing symbolic calculus and extracts conditions under which certain optimization transformations are legal. At run time, it evaluates these optimization correctness predicates and switches on the optimization when they hold true.

The compile-time part of Hybrid Analysis formulates an independence problem in terms of sets of references: the set of memory locations *read* in one iteration must not overlap with the set of memory locations *written* in another iteration.

(a)

```
1 Read *, n
2 Do j = 1, n
3    A(j) = A(j+200)
4    If (x < 0) Then
5       A(j) = A(j) + A(j+100)
6    EndIf
7 EndDo
```

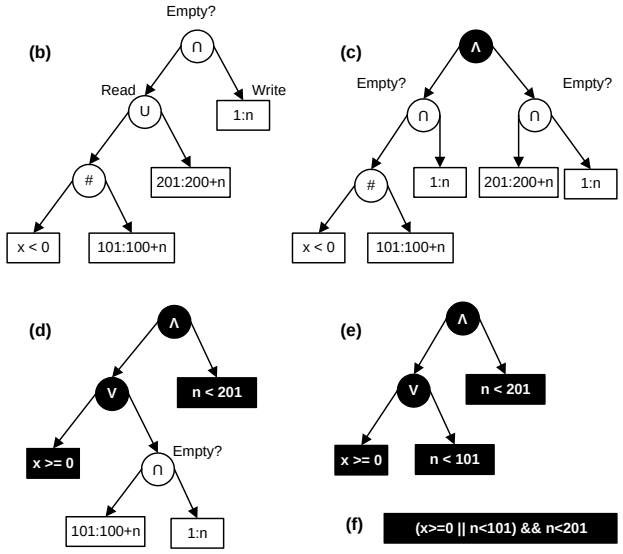


Fig. 8. Extraction of an independence predicate from an independence equation. The black nodes represent simple conditions and logical operations that are easier to evaluate at run time than it is to solve the original independence problem. (a) Original code. (b) Independence equation as intersection of *read* and *write* reference sets. $\cap$ and $\cup$ stand for set intersection and union respectively and # means predication. (c) The original problem was divided into two subproblems. $\wedge$ and $\vee$ stand for logical *and* and *or* respectively. (d) Intermediate result. (e) The final result is an accurate independence predicate which is inserted in the generated code (f) and that will be evaluated efficiently at run time.

Most reference patterns in loops are more complex than those in Fig. 1(a). The relevant sets of references *read* and *write* cannot, in general, be represented as linear intervals. Moreover, parallelization is profitable at large levels of granularity, which correspond to large loops, spanning a large amount of code. Quite often, such loops contain nonlinear patterns, such as indirect memory references or nonlinear control flow. Let us follow the slightly more complex example in Fig. 8(a). The independence question is still represented as whether a set intersection is empty (b), but the set of *read* references is not a simple interval because of the unknown control flow value of

$x < 0$. It is thus not straightforward to extract conditions such as the bound check presented in Fig. 1. However, the problem of proving the *read* and *write* sets disjoint can be divided into two subproblems (c). In order to solve the first subproblem, let us notice that when $x < 0$ is false, the corresponding predicated set becomes empty. When $x < 0$ is true, the subproblem reduces to proving the sets disjoint, which is similar to the simpler problem in Fig. 1. The subproblem on the right in Fig. 8(c) is similar to the simpler problem in Fig. 1. The final result is shown in Fig. 8(f) as a simple logical expression which can be evaluated quickly at run time. Let us point out the important steps taken to solve these problems.

• We represent the set of memory locations that carry cross-iteration dependences as a tree in which the leaves are linear intervals and the internal nodes are operators, such as set union, set intersection and predication (Fig. 8(b)). We formulate the independence problem as testing whether this dependence set is empty.

• We apply a sequence of transformations which convert this problem into an equivalent logical expression that can be evaluated efficiently at run time (Fig. 8(c-e)). These transformations are applied in a recursive descent on the tree representation of the dependence set and are based on set algebra semantics.

B. Proposed Memory Reference Representation:USR

The USR, or *Uniform Set of References*, is a symbolic and compact representation of memory reference sets in a program. It can represent symbolically the aggregation of array memory references at any hierarchical level (on the loop and procedure call graph) in a program. It can represent the control flow (gates), inter-procedural issues (call sites) and recurrences (when array references, i.e., indices or gates, have to be expressed symbolically as a recurrence with no closed form solution or as an

subscripted subscripts). Its evaluation and subsequent optimization decisions can be: initiated and completed at compile time if all symbolic values can be analyzed, compared, or initiated at compile time with partial but insufficient results and completed at run-time.

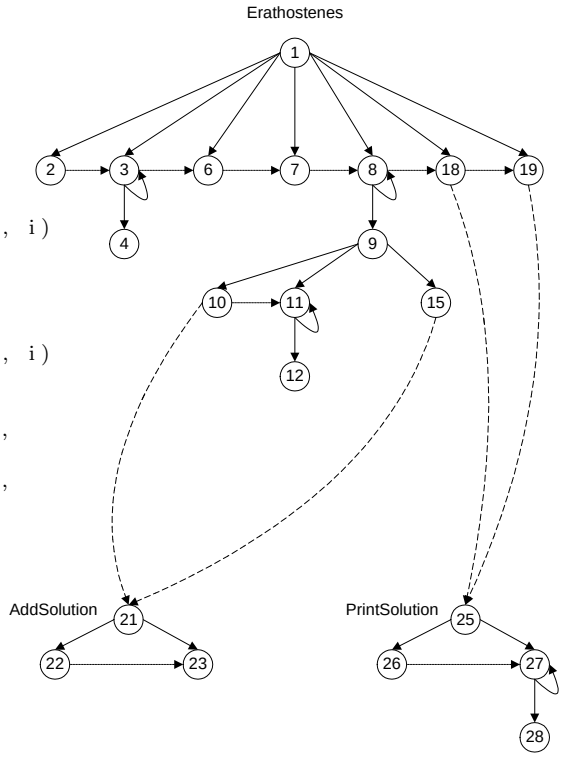
Any nontrivial program consists of several subprograms, complex loop nests and control structures, which determine the shape and size of memory reference patterns. In order to scale across large programs, most analysis techniques use *aggregation*, i.e., a way to represent several addresses by a single symbolic expression. For instance, the interval [n+1:n+4] is the aggregation of {n+1,n+2,n+3,n+4}. In the example in Fig. 1, we know that the set of *read* references for some iteration j is {j+100}. We can express the set of *read* references for the whole iteration space symbolically as [101:n+100].

In addition to aggregating memory reference sets over loops, control structures and subprograms, an analysis process must also operate on these sets according to logic particular to the particular analysis goals. Consider the example in Fig. 10(a). Let us assume that we want to propagate constant 0 stored in some elements of array A from the definition site at line 2 to the use site at line 5, and thus eliminate several unnecessary multiplication operations. Let us assume that we computed the set of addresses at which we write at line 2, [1:10], and we computed the set of addresses from which we read at line 5, [6:15]. In order to find the exact locations for which we can propagate constant 0, we must compute the *intersection* of these two sets. Most analysis techniques perform, in addition to aggregation, set operations such as intersection, set difference and union.

The symbolic aggregation process models the effect of language constructs on the set of memory addresses. It is essentially an *abstract interpretation* process that produces sets of references. The remainder of this section presents the rationale

```
 1 Program Erathostenes
 2 Read n
 3 Do i = 2, n
 4   isPrime(i) = true
 5 EndDo
 6 pCounter = 0
 7 cCounter = 0
 8 Do i = 2, Sqrt(n)
 9   If (isPrime(i))
10     Call AddSolution(pSolution, pCounter, i)
11     Do j = i*i, n, i
12       isPrime(i) = false
13     EndDo
14   Else
15     Call AddSolution(cSolution, cCounter, i)
16   EndIf
17 EndDo
18 Call PrintSolution(pSolution, pCounter,
                       'Primes: ')
19 Call PrintSolution(cSolution, cCounter,
                       'Non-Primes: ')
20 End

21 Subroutine AddSolution(V, c, x)
22 c=c+1
23 V(c) = x
24 End

25 Subroutine PrintSolution(V, c, msg)
26 Print msg
27 Do i=1, c
28   Print V(c)
29 End
```

(a)



(b)

Fig. 9. Program model: (a) Sample program – Erathostenes' sieve. (b) Call Graph and Control Dependence Graphs (CD edges shown as solid lines). CD siblings are connected by dotted lines given by the postdominance relation in the original Control Flow Graph. Subprogram call relations are shown as dashed lines.

behind the design of the USR and gives a formal definition. We start by presenting the program model and then detail our *abstract interpretation* rule for each program component. The design of the USR guarantees the representation's closure over the abstract interpretation process. This guarantees the applicability of any analysis technique based on USRs and set operations to any program that fits our model.

1. Program Model

This section describes the class of programs that USR-based analysis techniques can handle.

We see the program as a collection of subprograms as in Fig. 9. Each subprogram is seen as a Control Dependence Graph (CDG). The Call Graph and all CDGs are assumed acyclic except for CDG self loops. For the simplicity of the presentation we will assume all CDGs to be trees. The same analysis techniques can be generalized to general directed acyclic CDGs. When a subprogram's CDG is not a tree, we transform it into an equivalent block structured program, for which the CDG is a tree (Section VI. A). In addition to control dependence, we preserve the postdominance relations between control dependence siblings so we can reconstruct an equivalent control flow graph. The program must be in static single assignment (SSA) form.

In our model, each terminal CDG node is either an assignment statement, an I/O statement or a subprogram call. Each internal node is either an *If-Then-Else* statement or a *Do* statement. In our model, an analysis is an abstract interpretation (as a postorder traversal) of the CDG. Within a list of siblings, they are interpreted left to right. *If-Then-Else* structures, loops and subprogram calls have specific interpretations which reflect its semantic. Additionally, analysis techniques are allowed to perform set operations: intersection, set difference and union. For example, the constant propagation process in Fig. 10(a) performs set intersections after analyzing the two loops.

2. Background: the Linear Memory Access Descriptor

In many cases, subscript functions and predicates are linear, which makes them easy to represent using a classic representation such as constraint sets [159, 160] or

triplet based LMADs [161]. For practical reasons [1] we have chosen to use the triplet based LMAD as the elementary building block for USRs. We will show how we use this representation not only when the reference pattern is completely affine, but also to describe affine subsets. The use of LMADs in our framework can be substituted with little effort by any other semantically equivalent representation such as sets of linear constraints.

[161] defined the *Linear Memory Access Descriptor (LMAD)* as a representation of the subscripting offset sequence. Consider a loop nest of depth D with indices $I_k, k = 1, D$, where $I_k = 0, U_k$. Consider a reference to memory given by $\mathrm{A}(s_1(\vec{I}), s_2(\vec{I}), \ldots, s_m(\vec{I}))$, where $\vec{I}$=$(I_1, I_2, \ldots, I_d)$. If the subscripting function can be written in a sum-of-products form with respect to the individual loop indices,

$$F_a(\mathbf{s}(\vec{I})) = f_0 + f_1(I_1) + f_2(I_2) + \cdots + f_m(I_m) \tag{3.1}$$

then, we can isolate the effect of each loop index on the subscripting offset sequence.

The isolated effect of any loop in a loop nest on a memory reference represents a *dimension* of the access. A dimension k can be characterized by its *stride* and the number of iterations in the loop. The LMAD contains a starting value, called the *base offset* and a set of dimensions. For the loop at line 2 in Fig. 1(a), the *Read* pattern on array `A` is represented by a 1-dimensional LMAD, 100+[1:n-1]. The offset is 100, the stride of the single dimension is 1 and the iteration count is n. Throughout this presentation, we will use the simple interval notation for single dimensional, unit stride LMADs: [101:100+n].

[1] Our Polaris compiler already uses LMAD representation.

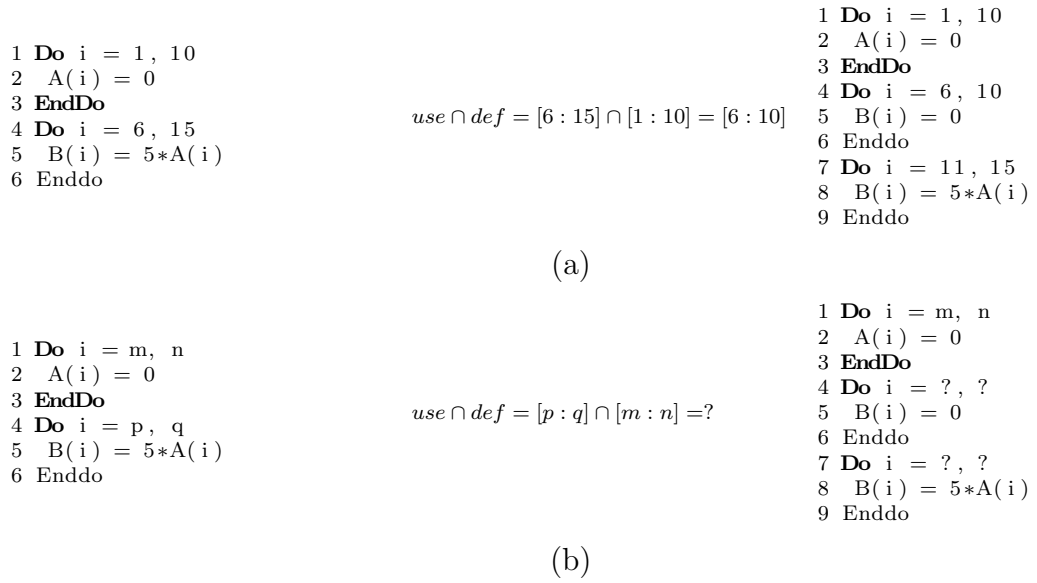


Fig. 10. USR intersection (∩): (a) when the result is an LMAD and (b) when the result cannot be represented as an LMAD.

3. Abstraction of Set Operations

a. Set Intersection

In the example in Fig. 10(a), we could express the result of the intersection $[6 : 15] \cap [1 : 10]$ as an LMAD, $[6 : 10]$. However, in the example in Fig. 10(b), due to the fact that the loop bounds are represented by symbolic names, the intersection of two LMADs cannot be represented as an LMAD. The universe of LMADs is not closed with respect to set intersection. Any analysis based solely on LMADs cannot analyze any program slice containing this code sample.

In order to represent the result of the intersection in Fig. 10(b), we have introduced a symbolic operator ∩. Rather than using a conservative LMAD value for the result, we prefer to keep it as $[p : q] \cap [m : n]$. Although the intersection cannot be performed at compile time, what is left for run time analysis is just one interval intersection operation, which is asymptotically less expensive than instrumenting every

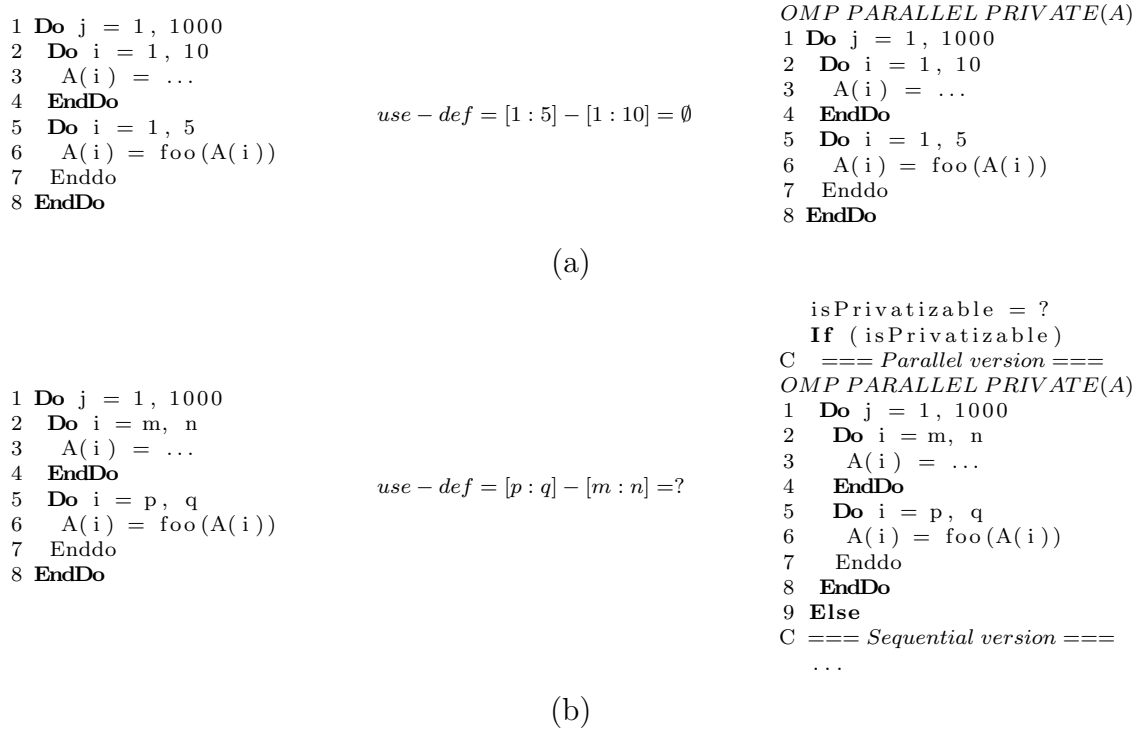


Fig. 11. USR difference (−): (a) when the result is an LMAD and (b) when the result cannot be represented as an LMAD.

reference. The advantage comes from the fact that the *use* and *def* sets are partially aggregated at compile time.

b. Set Difference

Consider the loop nest on the left hand side in the example in Fig. 11(a). Let us assume that we want to parallelize the outermost loop (line 1). Although it appears that the statement at line 6 causes a cross-iteration dependence on memory locations A(1:5), this dependence can be eliminated by privatizing array A. The equivalent parallel code is shown on the right. In order to verify the validity of the privatization transformation, the compiler must prove that the *use* at line 6 is covered by the *def* at line 3, for each iteration of the outermost loop. In other words, it must prove a

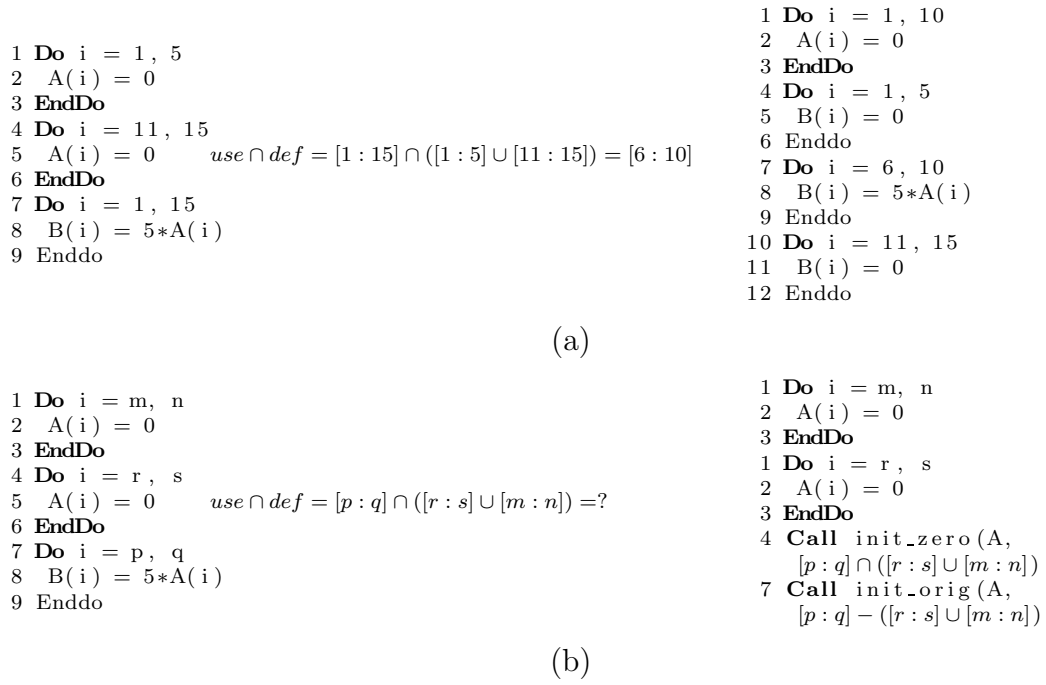


Fig. 12. USR union (∪): (a) when the result is an LMAD and (b) when the result cannot be represented as an LMAD.

set difference identity: $use - def = \emptyset$. In this case the result of the LMAD operation $[1 : 5] - [1 : 10]$ can be expressed using an LMAD. However, in the slightly more complex example in Fig. 11(b), the result of the set difference cannot be expressed as an LMAD. Similarly to the intersection operator introduced above, we use a symbolic *set difference* operator and keep the partially aggregated result $[p : q] - [m : n]$. Although this expression cannot be evaluated before run time, it costs far less to compute it than to instrument and analyze every dynamic reference to A.

c. Set Union

Consider the example in Fig. 12(a). It shows a constant propagation opportunity similar to the one in Fig. 10(a). The difference is that the def set is not contiguous, but consists of two LMADs. We use the *union* operator ∪ to describe symbolic unions

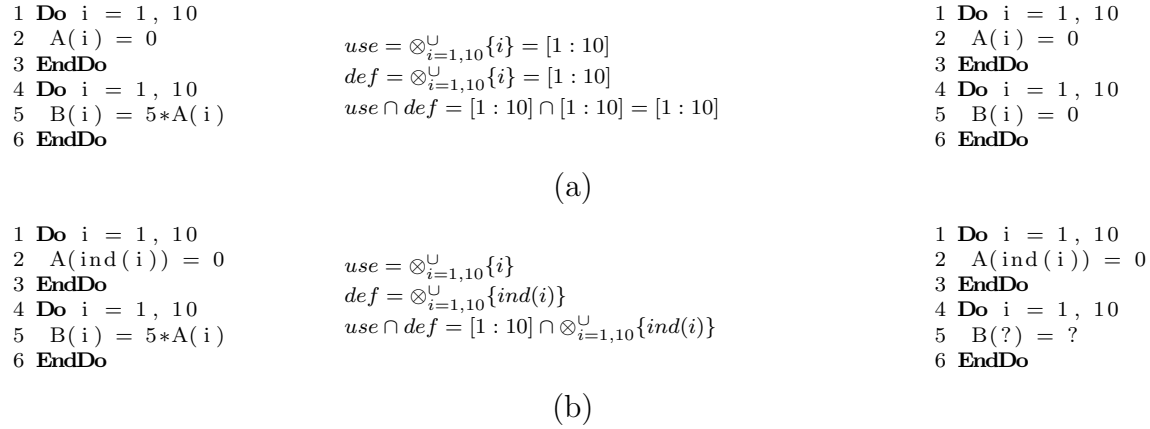


Fig. 13. USR expansion ($\otimes^{\cup}$): (a) when the result is an LMAD and (b) when the result cannot be represented as an LMAD.

between sets of references. Fig. 10(b) shows a slightly more complex case where neither the union nor the intersection could be performed symbolically. However, they are partially aggregated. At run time, the exact sets of addresses at which the constant can be propagated will be evaluated by performing the necessary set operations.

4. Abstraction of Loops

Over the previous examples we have taken for granted that the effect of a loop $Do\ i = 1, 10$ on an individual reference such as $A(i)$ is LMAD $[1 : 10]$. This is always the case when the reference pattern inside the loop is a single point which is an affine function of the loop index, as is the case in Fig. 13(a). However, when the reference inside the loop is through a nonlinear expression such as subscripted subscripts (Fig. 13(b)), we cannot aggregate the effect of the whole loop symbolically into an LMAD anymore. In order to let the abstract interpretation process aggregate the effect on memory of program slices that contains nonlinear addressing, we have introduced a new *set expansion operator* $\otimes^{\cup}_{Recurrence}$. The result of $\otimes^{\cup}_{Recurrence}$ is

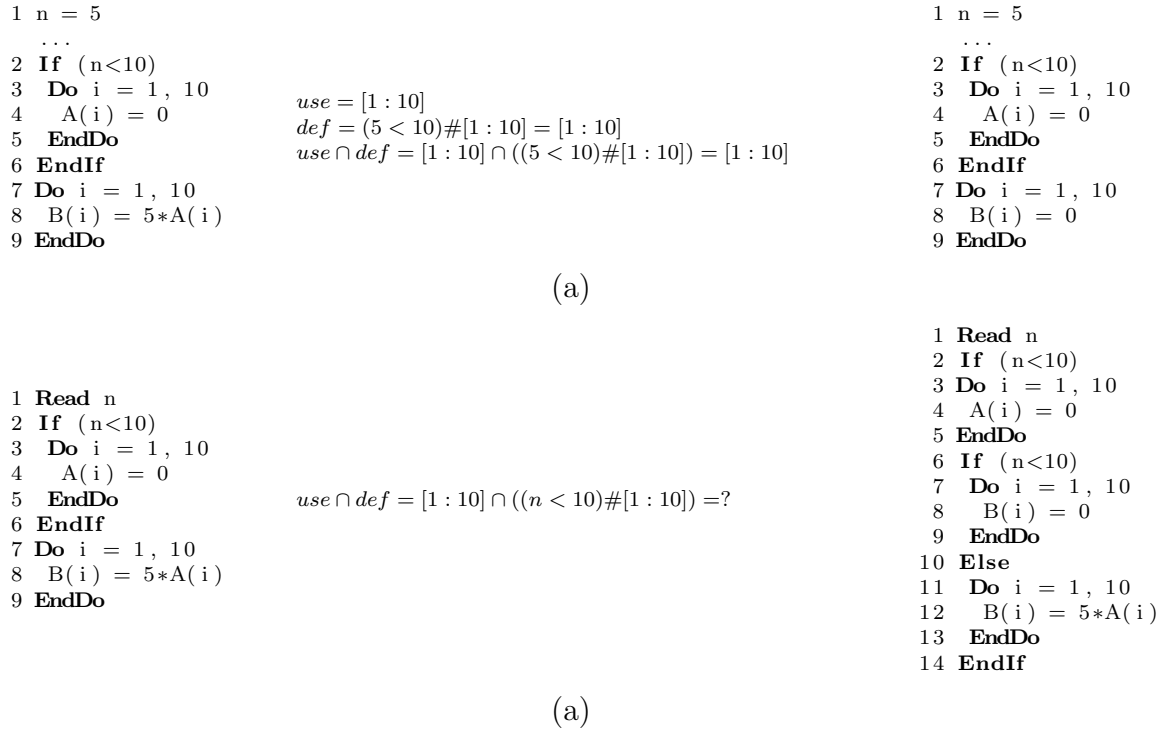


Fig. 14. USR gate (#): (a) when the result is an LMAD and (b) when the result cannot be represented as an LMAD.

the union of the effect on memory of individual iterations, over the whole iteration space. In other words, $\otimes^{\cup}_{i=1,n} set_i = \bigcup_i^n set_i$. For completeness, we also introduce the complementary operator $\otimes^{\cap}_{Recurrence}$, where $\otimes^{\cap}_{i=1,n} set_i = \bigcap_i^n set_i$. The evaluation of expansion operators can be prohibitively expensive. However, expressions involving expansion are useful when the per-iteration reference pattern is not just an individual reference, but an already aggregated LMAD. In such cases, we still benefit from an asymptotic reduction in complexity while maintaining analysis accuracy, whereas other techniques would resort to approximation and miss optimization opportunities.

5. Abstraction of Control

All nontrivial programs contain explicit control constructs such as *If-Then-Else*. In the example in Fig. 14(a), the definition at line 4 is conditioned by predicate $n < 10$. If we know that the predicate holds true at compile time, then we can infer that the definition covers the use for memory locations A(1:10) and we can propagate the constants from definition site 4 to use site 8. However, in Fig. 14(b) we do not know the value of the predicate at compile time. Then the memory reference pattern across the whole *If* block (lines 2-6) cannot be expressed as an interval. Although the LMAD extends intervals by providing a placeholder for a predicate, we chose for generality to introduce a *gate* operator #. The result of *predicate#set* is *set* when the predicate holds true and $\emptyset$ when the predicate holds false respectively. The *gate* operator is crucial to extracting control-accurate memory reference patterns, and is easy to reason with at run time. It translates into a simple logical expression evaluation.

6. Abstraction of Subprograms

Virtually all nontrivial programs consist of several subprograms. Analysis of program slices spanning multiple subprograms is crucial to the applicability and efficiency of global optimization techniques. For instance, large granularity parallelism is found usually at the outer loop level in nests spanning multiple subprograms. The simplest way to analyze subprograms is to inline them. However, for large programs inlining is impractical because it may lead to code expansion exponential in the size of the call graph. We chose not to inline subprograms. Our analysis strategy is to (1) create a parameterized view of the memory reference pattern of each subprogram and (2) to instantiate this view at each corresponding call site. In the example in Fig. 15(a),

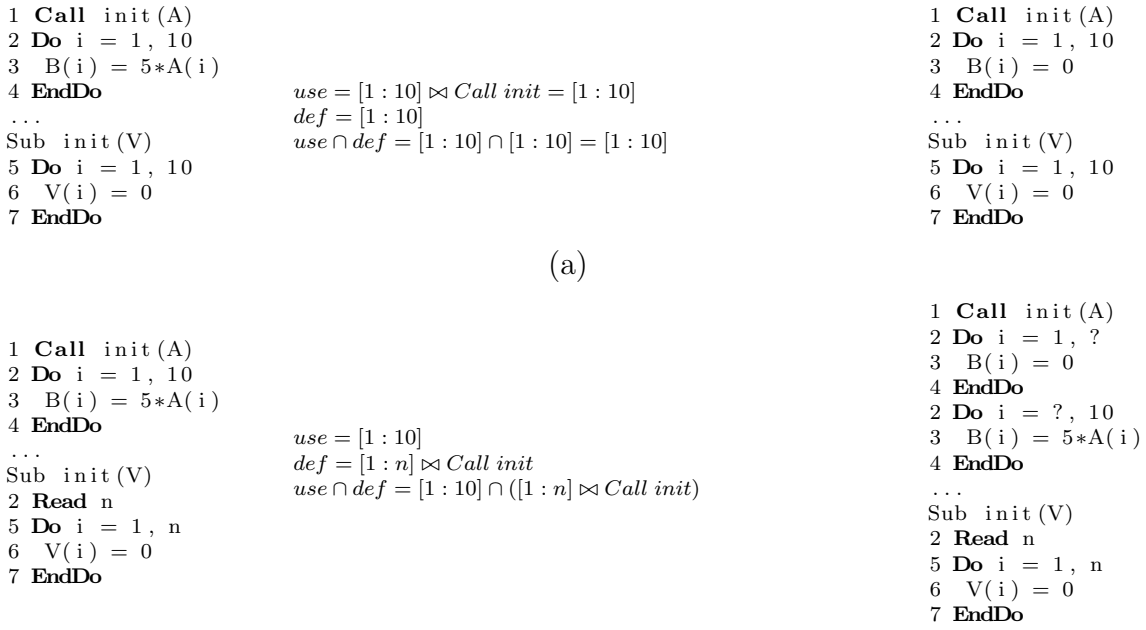


Fig. 15. USR translation ($USR \bowtie CallSite$): (a) when the result is an LMAD and (b) when the result cannot be represented as an LMAD.

the definition site within subroutine *init* references memory locations V(1:10). At the call site at line 2, this translates to definitions at locations A(1:10). The translation from the parameterized reference set V(1:10) to the actual A(1:10) can be done at compile time, thus the values can be propagated. In the example in Fig. 15(b), one of the parameters of the reference pattern, n, is defined within the called subroutine and thus cannot be translated at any call site. We introduce the translation $\bowtie$ operator to postpone the process of computing the parameterized reference set at the call site. This process can then be performed at run time, in the presence of the actual value of n, using either the inspector/executor or speculative execution run time optimization models.

$$
\begin{aligned}
\Sigma &= \{\cap, \cup, -, (,), \#, \otimes^{\cup}, \otimes^{\cap}, \bowtie, \\
&\qquad LMADs, Gate, Recurrence, CallSite\} \\
N &= \{USR\}, \;\; S = USR \\
P &= \{USR \rightarrow LMADs | (USR) \\
&\qquad USR \rightarrow USR \cap USR \\
&\qquad USR \rightarrow USR \cup USR \\
&\qquad USR \rightarrow USR - USR \\
&\qquad USR \rightarrow Gate \# USR \\
&\qquad USR \rightarrow \otimes^{\cup}_{Recurrence} USR \\
&\qquad USR \rightarrow \otimes^{\cap}_{Recurrence} USR \\
&\qquad USR \rightarrow USR \bowtie CallSite\}
\end{aligned}
$$

Fig. 16. USR formal definition. $\cap$, $\cup$, $-$ are elementary set operations: intersection, union, difference. $Gate\#USR$ represents reference set USR predicated by condition $Gate$. $\otimes^{\cup}_{i=1,n} USR(i)$ represents the union of reference sets $USR(i)$ across the iteration space $i = 1 : n$. $USR(formals) \bowtie Call\ Site$ represents the image of the generic reference set $USR(formals)$ instantiated at a particular call site.

7. Formal Definition

The USR is a symbolic representation of a program slice that computes a set of memory addresses. When the address formula is an affine function of loop indices and bounds, the USR is usually an LMAD[2]. When the analysis process combines LMADs such that the result cannot be represented as an LMAD, the USR is a symbolic expression in which the leaves are LMADs and the internal operators pinpoint the exact points and causes of static analysis failure, usually nonlinearity. They constitute an excellent starting point for extracting run time conditions from a USR equation as will be shown over the following sections.

Formally, a USR is an expression in the language presented in Fig. 16. The crucial

[2]The LMAD has some limitations to the shape of the affine polytope it can represent. We can easily adapt our system to a different primary representation by just replacing the LMAD data structure with a new data structure that preserves its semantics (such as systems of linear constraints)

feature of the USR is that it is closed with respect to all the operations required by aggregation and classification processes that make up a large class of analysis techniques. The USR can thus be used to implement a large class of optimization techniques which are guaranteed to apply to any program that fits our model.

We define the *evaluation* of a USR as the process of finding the set of addresses it represents, as integer values (i.e., not symbolic). Some USRs can be evaluated at compile-time (such as the ones that are made of an LMAD containing only known integer values). The ones that cannot be evaluated at compile-time can be embedded in the code and evaluated at run time. However, many optimization decisions do not require USRs to be evaluated at all, but rather seek answers to questions about the relations between two or more USRs.

C. Hybrid Memory Reference Analysis using USRs

Memory reference analysis usually consists of computing the relation between two or more sets of addresses, as is the case in dependence analysis or array privatization. In other cases, such as constant propagation, we are interested in computing the exact shape and size of a set of addresses, such as the ones that contain constant values.

There are two important benefits to using USRs. First, the analysis is scalable due to symbolic aggregation. Second, if a compile time decision cannot be reached, the USRs can be compared (or computed) efficiently at run time.

The following subsection presents our implementation based on USRs of the *Memory Classification Analysis* [161], a general array data flow analysis technique. We introduce then a general way to express optimization validity questions such as "is this loop parallelizable?" as USR identities. Section D shows in great detail how we can extract efficient run time tests to verify arbitrarily complex USR identities at

Table II. MCA partitions for the privatization problem on array A in Fig. 11(a).

Lines	**RO**	**WF**	**RW**
3	$\emptyset$	$\{i\}$	$\emptyset$
2-4	$\emptyset$	$[1:10]$	$\emptyset$
6	$\emptyset$	$\emptyset$	$\{i\}$
5-7	$\emptyset$	$\emptyset$	$[1:10]$
2-7	$\emptyset$	$[1:10]$	$\emptyset$

run time, thus enabling a large class of low cost dynamic optimization techniques.

1. Memory Classification Analysis

Memory Classification Analysis (MCA), presented in [161], consists of partitioning the memory locations referenced within a given program slice into ReadOnly (RO), WriteFirst (WF) and ReadWrite (RW). RO locations are read but never written, WF are first written then possibly read and/or written, and RW are first read, then written – with possibly other read and write operations in between or afterwards. This classification helps express data flow and data dependence relations across arbitrarily large program slices in a scalable way, as long as the underlying representation is scalable.

Consider the privatization problem presented in Fig. 11(a). A naive dependence analysis would report possible flow dependences at statement 6 across iterations of the outer loop. However, we can see that the *read* at line 6 is covered by the *write* at line *3* within the same iteration of the outer loop, thus it cannot possibly cause cross-iteration dependences *after* privatizing array A. However, in general it is not easy to decide whether privatization can eliminate dependences. MCA makes it easy to solve the array privatization problem. Table II presents the MCA partitions at various levels for the code in Fig. 11(a). The last row shows that, at the outer loop

```
1 A(1) = ...
2 A(2) = ...
3 A(i) = A(j)
```

(a)

Lines	RO	WF	RW
1	$\emptyset$	$\{1\}$	$\emptyset$
2	$\emptyset$	$\{2\}$	$\emptyset$
1-2	$\emptyset$	$[1:2]$	$\emptyset$
3	$\{j\} - \{i\}$	$\{i\} - \{j\}$	$\{i\} \cap \{j\}$
1-3	$\{j\} - ([1:2] \cup \{i\})$	$[1:2] \cup (\{i\} - \{j\})$	$(\{i\} \cap \{j\}) - [1:2]$

(b)

Fig. 17. Classification of references in straight line code. (a) Sample code. (b) MCA partitions.

body level, all memory locations referenced are first written to. In other words, all *reads* are contained to locations defined within the same iteration of the outer loop, thus cannot cause cross iteration dependences after privatization.

MCA partitions can be used to solve a variety of optimization problems, from dependence analysis to checkpoint size reduction. Additionally, other problems can be solved using memory reference aggregation and classification processes similar to MCA. The remainder of this section presents our implementation of MCA in which the RO, WF and RW sets are represented as USRs.

At a high level, MCA can be seen as an abstract interpretation of the program. It consists of a bottom-up traversal of the CDG of each program, and of the Call Graph at the interprocedural level. At each point, the RO, WF and RW sets are computed based on the previously computed partitions of the statements below in the CDG/Call Graph. For instance, in Table II, the classification across lines 2-4 is based on the classification at line 3.

a. Classification of References in Straight Line Code

Consider the example in Fig. 17. The effect on memory of the statement at line 1 is the partition (RO, WF, RW) shown on the second row. The RO and RW sets are empty, and the WF set consists of a single element, address 1 (addresses are

Algorithm McaSuccessiveBlocks
Input: (WF_1, RO_1, RW_1) , (WF_2, RO_2, RW_2)
Output: (WF, RO, RW)
$WF = WF_1 \cup (WF_2 - (RO_1 \cup RW_1))$
$RO = (RO_1 - (WF_2 \cup RW_2)) \cup (RO_2 - (WF_1 \cup RW_1))$
$RW = RW_1 \cup (RW_2 - WF_1) \cup (RO_1 \cap WF_2)$

Fig. 18. MCA algorithm for successive statements.

kept relative to the beginning of the array). After computing the effect of statement 1ant that of statement 2, the analysis merges them into a single (RO, WF, RW) partition. After analyzing statement 3, its effect is merged to that of the block made of statements 1 and 2.

Fig. 18 presents the formulas that merge the MCA partitions corresponding to two consecutive blocks. In order to compute the WF component across both blocks, we need to unite the WF_1 of the first block (because they will certainly be WF across both blocks) with the part of WF_2 (second block) that was not read in the first block ($RO_1 \cup RW_1$). Because the values of i and j are not known, the results cannot be expressed as LMADs. The operations that could not be performed are represented by USR operators $\cup$, $\cap$ and $-$.

Formally, a block is a contiguous sequences of Control Dependence Graph siblings. An element in a block can be a single statement such as an assignment, as well as a loop, an *If-Then-Else* structure or a subprogram call.

b. Classification of References in Conditional Blocks

Consider the example in Fig. 19. We can see that location *A(2)* gets initialized regardless of the value of *x*. Also, the use of *A(1)* at line 7 is always covered by its definition at line *2*. This kind of information can be used to validate transformations such as renaming or to verify program correctness (ensure that A(1) is not used

```
1 If (x<0)
2   A(1) = ...
3 Else
4   A(2) = ...
5 EndIf
6 If (x<0)
7   A(2) = 2*A(1)
8 EndIf
```

(a)

Lines	RO	WF	RW
2	$\emptyset$	$\{1\}$	$\emptyset$
4	$\emptyset$	$\{2\}$	$\emptyset$
1-5	$\emptyset$	$((x<0)\#\{1\}) \cup ((x \geq 0)\#\{2\})$	$\emptyset$
7	$\{1\}$	$\{2\}$	$\emptyset$
6-8	$(x<0)\#\{1\}$	$(x<0)\#\{2\}$	$\emptyset$
1-8	$\emptyset$	$((x<0)\#\{1\}) \cup \{2\}$	$\emptyset$

(b)

Fig. 19. Classification of references in conditional blocks. (a) Sample code. (b) MCA partitions.

Algorithm McaConditionalBlocks
Input: $condition$, (WF_1, RO_1, RW_1), (WF_2, RO_2, RW_2)
Output: (WF, RO, RW)
$RO = (condition\#RO_1) \cup (\neg condition\#RO_2)$
$WF = (condition\#WF_1) \cup (\neg condition\#WF_2)$
$RW = (condition\#RW_1) \cup (\neg condition\#RW_2)$

Fig. 20. MCA algorithm for mutually exclusive conditional blocks.

without being defined). Let us see how this information is extracted automatically by MCA.

In order to classify references across the *If-Then-Else* structure at lines 1-5, the analysis first classifies individual statements 2 and 4. It then applies the classification formulas shown in Fig. 20 to produce the values shown on the row labeled 1-5 in the table in Fig. 19(b). A similar process produces the classification across lines 6-8. The overall partition for lines 1-8 is produced using the algorithm presented in the previous section (Fig. 18).

c. Classification of References in Loops

The example in Fig. 21 shows the MCA process for two arrays across the iteration space of a loop. The results must be the same as if the loop were fully unrolled and

```
1  Do i=1,10
2    A(i+1) = 2*A(i)
3    B(i)   = 2*B(i+1)
4  EndDo
```

(a)

Lines for A	RO	WF	RW
2	$\{i\}$	$\{i+1\}$	$\emptyset$
1-4	$\{1\}$	$[2:11]$	$\emptyset$
Lines for B	**RO**	**WF**	**RW**
3	$\{i+1\}$	$\{i\}$	$\emptyset$
1-4	$\{11\}$	$\{1\}$	$[2:10]$

(b)

Fig. 21. Classification of references in loops. (a) Sample code. (b) MCA partitions.

Algorithm McaLoopBlock
Input: $(j=1,n)$, (WF_j, RO_j, RW_j)
Output: (WF, RO, RW)
$WF = \otimes^{\cup}_{j=1,n}\left[WF_j - \otimes^{\cup}_{k=1,j-1}(RO_k \cup RW_k)\right]$
$RO = (\otimes^{\cup}_{j=1,n} RO_j) - \left[\otimes^{\cup}_{j=1,n}(WF_j \cup RW_j)\right]$
$RW = \left[\otimes^{\cup}_{j=1,n}(RO_j \cup RW_j)\right] - (WF \cup RO)$

(a)

Algorithm FastMcaLoopBlock
Input: $(j=1,n)$, (WF_j, RO_j, RW_j)
Output: (WF, RO, RW)
$WF = \otimes^{\cup}_{j=1,n} WF_j$
$RO = \otimes^{\cup}_{j=1,n} RO_j$
$RW = \otimes^{\cup}_{j=1,n} RW_j$
$dirty = WF \cap RO$
$RW = RW \cup dirty$
$RO = RO - dirty$
$WF = WF - dirty$

(b)

Fig. 22. MCA algorithm for loops. (a) accurate but possibly slower, (b) approximative but faster.

we applied the rules for straight line code. The actual MCA partition computation shown in Fig.22 does not rely on unrolling. It is based on abstract interpretation of USRs across loops, i.e. using the $\otimes^{\cup}$ operator.

The formula for computing WF in Fig.22(a) appears to have quadratic complexity ($j = 1, n$ *and* $k = 1, j-1$). This would imply that its run time evaluation would be costly. However, in many cases run time USR evaluation is not needed, and even when it is needed it can often be done in linear time similar to a partial sum computation. The algorithm in Fig.22(b) produces simpler USRs but does not represent accurately the order between *writes* and *reads* in different iterations of the loop. The conservative

```
 1 Do i=1,1000
 2   x1 = 1
 3   Do j=1,2
       x2 = φ(x1, x5)
 4     If (x2=1)
 5       Do k=1,10
 6         A(k) = 0
 7       Enddo
 8         x3=0
 9     Else
10       Do k=1,10
11         ... = A(k)
12       Enddo
13       x4=1
14     EndIf
       x5 = φ(x3, x4)
15   EndDo
16 EndDo

Across lines 4−14:
RO_j = (x2 ≠ 1)#[1 : 10]
WF_j = (x2 = 1)#[1 : 10]
RW_j = ∅
```

(a)

$$
\begin{aligned}
WF &= \otimes^{\cup}_{j=1,2}\left[WF_j - \otimes^{\cup}_{k=1,j-1}(RO_k \cup RW_k)\right] \\
&= \left[WF_1 - \otimes^{\cup}_{k=1,0}(RO_k \cup RW_k)\right] \cup \left[WF_2 - \otimes^{\cup}_{k=1,1}(RO_k \cup RW_k)\right] \\
&= ([1:10] - \emptyset) \cup (\emptyset - \emptyset) \\
&= [1:10] \\
RO &= (\otimes^{\cup}_{j=1,2} RO_j) - \left[\otimes^{\cup}_{j=1,2}(WF_j \cup RW_j)\right] \\
&= (RO_1 \cup RO_2) - (WF_1 \cup WF_2 \cup RW_1 \cup RW_2) \\
&= (\emptyset \cup [1:10]) - ([1:10] \cup \emptyset \cup \emptyset \cup \emptyset) \\
&= \emptyset \\
RW &= \left[\otimes^{\cup}_{j=1,2}(RO_j \cup RW_j)\right] - (WF \cup RO) \\
&= \left[\otimes^{\cup}_{j=1,2}(RO_j \cup \emptyset)\right] - (WF \cup RO) \\
&= (RO_1 \cup RO_2) - (WF \cup RO) \\
&= (\emptyset \cup [1:10]) - ([1:10] \cup \emptyset) \\
&= \emptyset
\end{aligned}
$$

(b)

$$
\begin{aligned}
RW &= (\otimes^{\cup}_{j=1,2} RW_j) \cup (RO \cap WF) \\
&= \emptyset \cup \left[(\otimes^{\cup}_{j=1,2} RO_j) \cap (\otimes^{\cup}_{j=1,2} WF_j)\right] \\
&= (RO_1 \cup RO_2) \cap (WF_1 \cup WF_2) \\
&= (\emptyset \cup [1:10]) \cap ([1:10] \cup \emptyset) \\
&= [1:10] \cap [1:10] \\
&= [1:10]
\end{aligned}
$$

(c)

Fig. 23. Case study to compare the algorithms in Fig.22. (a) Sample code showing the Static Single Assignment numbers and ϕ functions for variable x. RW for the body of the outer loop using (b) the accurate algorithm and (c) the approximating algorithm.

direction is to classify read-write overlaps as *RW* although it could be possible that the actual sequence was *write*, *read*.

The difference between the accurate and approximative algorithms presented in Fig.22 is illustrated in Fig.23. The MCA partition for the loop at lines 3-15 produced by the accurate algorithm shows that the outermost loop at line 1 can be parallelized after privatizing array A. There can exist no upwards exposed *reads* since $RO = RW = \emptyset$. Using the approximating algorithm, $RW = [1:10]$, which would imply the existence of flow dependences across iterations of the outermost loop, which in turn would prevent it from being executed in parallel. We have used the accurate

```
 1 Do j=1,1000
 2   Call init(A,j)
 3   Do i=1,10
 4     R(i,j) = R(i,j) + A(i)
 5   EndDo
 6 EndDo
...
 7 Sub init(V,x)
 8 Do i=1,10
 9   V(i) = i*x
10 Enddo
```

(a)

init: *V*	**RO**	**WF**	**RW**
9	$\emptyset$	$\{i\}$	$\emptyset$
7-10	$\emptyset$	$[1:10]$	$\emptyset$
A	**RO**	**WF**	**RW**
2	$\emptyset$	$[1:10]$	$\emptyset$
4	$\{i\}$	$\emptyset$	$\emptyset$
3-5	$[1:10]$	$\emptyset$	$\emptyset$
2-5	$\emptyset$	$[1:10]$	$\emptyset$

(b)

Fig. 24. Classification of references at subprogram call sites. (a) Sample code. (b) MCA partitions.

Algorithm McaSubprogramBlock
Input: $Call\ subpgm(actuals)$, $(WF_{formals}, RO_{formals}, RW_{formals})$
Output: (WF, RO, RW)
$RO = RO_{formals} \bowtie Call\ subpgm(actuals)$
$WF = WF_{formals} \bowtie Call\ subpgm(actuals)$
$RW = RW_{formals} \bowtie Call\ subpgm(actuals)$

Fig. 25. MCA algorithm for a subprogram call site.

algorithm in all our experiments, although there were just a few cases where it was really needed. When compilation speed is very important, the simple approximating version could be used instead.

d. Interprocedural Classification of References

The classification of memory locations referenced by a called subprogram relies on the parameterized classification of the callee. We use the USR translation operation $\bowtie$ to create an actual instance of the MCA partition of the callee. The symbolic name translation is based on our interprocedural extension to Static Single Assignment. In the example in Fig. 24, $V \rightarrow A$. The translation operation (Fig.25) replaces formal arguments and global variables in the caller with their corresponding actual values at

Algorighm BuildDependenceSet
Input: (j=1,n) , (RO_j, WF_j, RW_j)
$DS = (\otimes^{\cup}_{j=1,n} WF_j) \cap (\otimes^{\cup}_{j=1,n} RO_j)$
$DS = DS \cup \left[(\otimes^{\cup}_{j=1,n} WF_j) \cap (\otimes^{\cup}_{j=1,n} RO_j)\right]$
$DS = DS \cup \left[(\otimes^{\cup}_{j=1,n} WF_j) \cap (\otimes^{\cup}_{j=1,n} RW_j)\right]$
$DS = DS \cup \left[\otimes^{\cup}_{j=1,n} RW_j \cap (\otimes^{\cup}_{k=1,j-1} RW_k)\right]$
$DS = DS \cup \left[(\otimes^{\cup}_{j=1,n} WF_j \cap (\otimes^{\cup}_{k=1,j-1} WF_k))\right]$

Fig. 26. Algorithm to compute the set of memory locations that carry cross iteration dependences (expressed as a USR).

the call site.

2. Dependence Testing as Verification of USR Identities

Data dependence is the foundation a number of important transformations necessary in order to use parallel hardware efficiently, such as thread level parallelization, SIMD-ization, vectorization or instruction level parallelization.

Most classic data dependence analysis techniques consider each pair of statements that may access the same array and prove that different iterations of a loop will access different array elements respectively. This is sufficient to prove that there is no flow of information among iterations, thus they can be executed in parallel without any synchronization.

Rather than looking at pairs of statements, we look at memory locations that correspond to data dependences. We compute the set of all the memory locations referenced by two statements executed in different iterations of a loop, and in which at least one is *write*. Fig. 26 shows how we compute this *Dependence Set*, based on the results of MCA across the loop body. DS is expressed as a USR.

Proving that there are no cross-iteration dependences reduces to proving that $DS = \emptyset$. We have reduced the data dependence problem to proving that a USR is

empty. The following section presents in detail how verifying USR identities can be done efficiently in a hybrid (static and dynamic) way.

D. Hybrid Dependence Analysis

This section presents a technique to prove USR identities such as $Dependence\ Set = \emptyset$ at compile time and run time. Although proving such identities applies to several analysis techniques, we will restrict the presentation to data dependence analysis.

Hybrid Dependence Analysis represents the process of solving dependence equations efficiently by using a mix of compile time and run time techniques. We represent dependence equations as $DS = \emptyset$, where DS is the set of all memory locations that carry dependences. The compile time part of HDA starts by trying to prove statically that the dependence set is empty. If it succeeds, the corresponding loop will be run in parallel without using any run time tests. If it proves statically that the dependence set is not empty, the loop will be run sequentially. When a decision cannot be made at compile time, HDA extracts statically simple independence conditions that are (1) sufficient to prove the loop parallel and (2) easy to evaluate at run time. Its run time analysis consists of evaluating the independence condition and selecting the parallel code version when it holds true.

In Fig. 8 we presented this process intuitively for a simple case. In this section we will describe how we automate the process of extracting simple independence conditions from general dependence equations. We will first present formally the data structures involved in this transformation and then follow with a detailed presentation of the algorithms. We use an existing USR representation [3] for sets of references, which has a tree structure as shown in Fig. 8(b). We will introduce a representation named PDAG to represent independence conditions. They can be seen as symbolic

$$
\begin{aligned}
\Sigma = \{&\wedge, \vee, \neg, (,), \otimes^{\wedge}, \otimes^{\vee}, \bowtie, LogicalExpression, Recurrence,\\
&Call\ Site, Library\ routine, Reference\ based\ test\}\\
N = \{&PDAG\}, \ \ S = PDAG\\
P = \{&PDAG \rightarrow LogicalExpression | (PDAG)\\
&PDAG \rightarrow PDAG \wedge PDAG\\
&PDAG \rightarrow PDAG \vee PDAG\\
&PDAG \rightarrow \neg PDAG\\
&PDAG \rightarrow \otimes^{\wedge}_{Recurrence} PDAG\\
&PDAG \rightarrow \otimes^{\vee}_{Recurrence} PDAG\\
&PDAG \rightarrow PDAG \bowtie Call\ Site\\
&PDAG \rightarrow Library\ routine\\
&PDAG \rightarrow Reference\ based\ test\}
\end{aligned}
$$

Fig. 27. PDAG formal definition. $\wedge$, $\vee$, $\neg$ are the elementary logical operators *and*, *or*, *not*. $\otimes^{\wedge}_{i=1,n} PDAG(i)$ holds true if and only if each of $PDAG(i)$ holds true, $i = 1, n$. $PDAG(formals) \bowtie Call\ Site$ represents the instantiation of a generic $PDAG$ at a particular call site. A specialized *library routine* may be employed to produce the value of the predicate. If a test based on simple comparisons and logical operations cannot be found, we fall back to a *reference based test*.

expressions that will produce at run time the boolean value of a dependence test.

1. Symbolic Representation: the PDAG

The Predicate Directed Acyclic Graph (PDAG) is an analytical, symbolic representation of a boolean expression. PDAGs are extracted automatically from dependence equations that cannot be solved statically $DS = \emptyset$, where DS is represented as a USR. *PDAGs are the boundary between the compile time and run time analysis.* They are the final result of static analysis, conditions used to predicate the validity of dynamic optimizations. They are inserted in the generated code and evaluated at run time. Their dynamic values are used to choose between sequential and parallel code versions. In its simplest form, the PDAG is a logical expression such as $x < 0$ in Fig. 1(c). At the other extreme, it can be an arbitrary program slice that produces

a boolean value. PDAGs are represented as trees having logical expressions as leaves and operators as internal nodes as in Fig. 8(b-e). The PDAG tree structure generally mirrors the tree structure of the dependence set as a USR, which in turn generally mirrors the block structure of the program. This makes PDAGs relatively easy to associate with sections in the original program, which makes it easier for compiler writers to program and understand the analysis process.

PDAGs are expressive enough to represent any possible dependence question, and simple enough to be quickly evaluated dynamically. The grammar in Fig. 27 defines PDAGs formally. They rely mostly on simple, logical operations and have a direct mapping to executable code. In addition to classic $\wedge$, $\vee$, and $\neg$ operators, PDAGs can also express conjunction ($\otimes^{\wedge}$) and disjunction ($\otimes^{\vee}$) of predicates over iteration spaces. Library routines such as monotonicity checks may be employed to express particular problems more efficiently, and reference based tests represent the fallback when cheaper conditions cannot be extracted.

2. Symbolic Analysis Algorithms

a. Syntax Directed Predicate Extraction

After resolving all statically analyzable dependence questions we are left with a Dependence Set (DS), represented as a USR, for which we could not give a definitive answer. For the resolution of this problem we have formulated the algorithm *Solve* shown in Fig. 28. This algorithm extracts a set of conditions, represented as a PDAG, which, when evaluated dynamically, returns *true* if and only if the dependence set is empty.

Algorithm *Solve* extracts the PDAG from the dependence set by recursively descending its USR tree and decomposing the nodes using elementary set algebra iden-

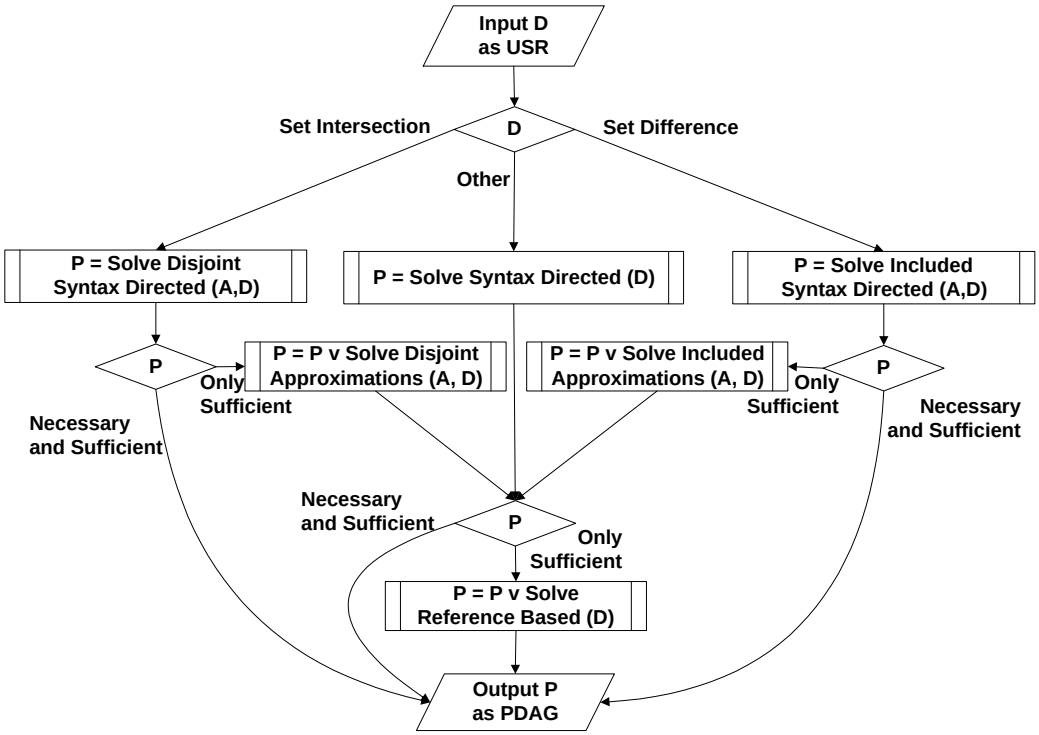


Fig. 28. **Algorithm Solve**: Extraction of a sufficient run time test as a PDAG from a dependence equation $D = \emptyset$. Details on the implementation of the subalgorithms are presented in the Appendix. We accumulate PDAGs in increasing order of complexity when the partial solutions are sufficient but not necessary, using the *logical or* operator $\vee$.

tity transformations (Fig. 29). For instance, in order to prove a union of two terms empty, it is necessary and sufficient to prove both terms empty. In other words, $A \cup B = \emptyset \Leftrightarrow A = \emptyset \wedge B = \emptyset$.

Our current implementation is optimistic, i.e., it extracts sufficient independence conditions. A similar approach can be used to extract pessimistic dependence conditions. Inexpensive pessimistic conditions could be used at run time to flag the sequential loops quickly and thus avoid the overhead of more expensive dependence tests. The algorithm maintains throughout the recursive descent process information

Algorithm Solve Syntax Directed
  Input: D as USR
  Output: P as PDAG s.t. $P \Rightarrow A = \emptyset$
Case D **of**:
  $LMADs$: $P = HasEmptyDimension(LMADs)$
  $A \cup B$: $P = Solve(A = \emptyset) \wedge Solve(B = \emptyset)$
  $q\#A$: $P = \overline{q} \vee Solve(A = \emptyset)$
  $\otimes^{\cup}_{i=1,n}(A_i)$: $P = \otimes^{\wedge}_{i=1,n} Solve(A_i)$
  $A \bowtie Call\ Site$: $P = Solve(A = \emptyset) \bowtie Call\ Site$

Algorithm Solve Disjoint Syntax Directed
  Input: A, D as USRs
  Output: P as PDAG s.t. $P \Rightarrow (A \cap D = \emptyset)$
Case D **of**:
  $B \cup C$: $P = Solve(A \cap B = \emptyset) \wedge Solve(A \cap C = \emptyset)$
  $q\#B$: $P = \overline{q} \vee Solve(A \cap B = \emptyset)$
  $\otimes^{\cup}_{i=1,n}(B_i)$: $P = \otimes^{\wedge}_{i=1,n} Solve(A \cap B_i)$
Case A **of**:
  $B \cup C$: $P = Solve(B \cap D = \emptyset) \wedge Solve(C \cap D = \emptyset)$
  $q\#B$: $P = \overline{q} \vee Solve(B \cap D = \emptyset)$
  $\otimes^{\cup}_{i=1,n}(B_i)$: $P = \otimes^{\wedge}_{i=1,n} Solve(B_i \cap D)$

Algorithm Solve Included Syntax Directed
  Input: A, D as USRs
  Output: P as PDAG s.t. $P \Rightarrow (A - D = \emptyset)$
Case D **of**:
  $B \cup C$: $P = Solve(A - B = \emptyset) \vee Solve(A - C = \emptyset)$
  $B \cap C$: $P = Solve(A - B = \emptyset) \wedge Solve(A - C = \emptyset)$
  $B - C$: $P = Solve(A - B = \emptyset) \wedge Solve(A \cap C = \emptyset)$
  $q\#B$: $P = (q, \text{true}) \wedge Solve(A - B = \emptyset)$
Case A **of**:
  $B \cup C$: $P = Solve(B - D = \emptyset) \wedge Solve(C - D = \emptyset)$
  $B \cap C$: $P = Solve(B - D = \emptyset) \vee Solve(C - D = \emptyset)$
  $B - C$: $P = Solve(B - D = \emptyset)$
  $q\#B$: $P = (\overline{q}, \text{false}) \vee Solve(B - D = \emptyset)$

Fig. 29. Algorithms to extract a PDAG from a USR identity based on USR syntax.

on whether the current solution is equivalent to the original independence problem. When the solution obtained by the recursive descent approach is sufficient but not necessary, more specialized and expensive reference based tests [129, 3] can be generated, thus avoiding a conservative decision (i.e., not parallel). The dynamic evaluation of these tests will then ensure an exact answer but will cost a higher run-time overhead, proportional to the dynamic reference count of the Dependence Set we started form. Fig. 30 presents such a case where a simple independence condition cannot be extracted.

```
1 Read *, (p(j), j=1,100),
          (q(j), j=1,100)
2 Do j = 1, 100
3     A(p(j)) =  A(q(j))
4 EndDo
```

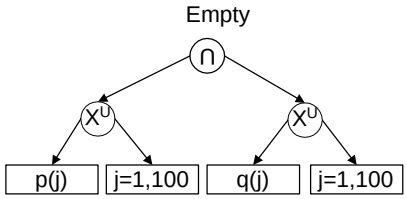


Fig. 30. A Hybrid Analysis extreme: in general, no test can solve this problem faster than the reference-by-reference LRPD test.

Unfortunately, the recursive descent approach does not work for set intersections and differences as well as for unions. An intersection could be empty even if none of its terms are (e.g., a set of odd numbers vs. a set of even ones). Algorithms *Solve Disjoint Syntax Directed* and *Solve Included Syntax Directed* continue the recursive descent according to the syntax of the terms of intersections and differences. They rely on dividing more complex equations such as $A \cap (B \cup C) = \emptyset$ into simpler equations such as $A \cap B = \emptyset$ and $A \cap C = \emptyset$, based on elementary set identities. However, there are USR configurations that cannot be broken up, such as $A \cap B \cap C = \emptyset$.

When the recursive descent described in algorithm *Solve* reaches such a point, it resorts to approximation to extract conditions that, in most cases, are sufficient but not necessary to prove independence.

b. Extracting PDAGs from USR Approximations

In the example in Fig. 31, array W could be proved privatizable by showing that the *read* at line 9 is covered by the *write* at line 5. However, the shape of the USR that describes the *write* pattern is outside any of the cases in the *Solve* algorithms presented above. We will show that even when two USRs cannot be compared directly, a meaningful PDAG can often be extracted based on comparisons between predicated approximations of the USRs.

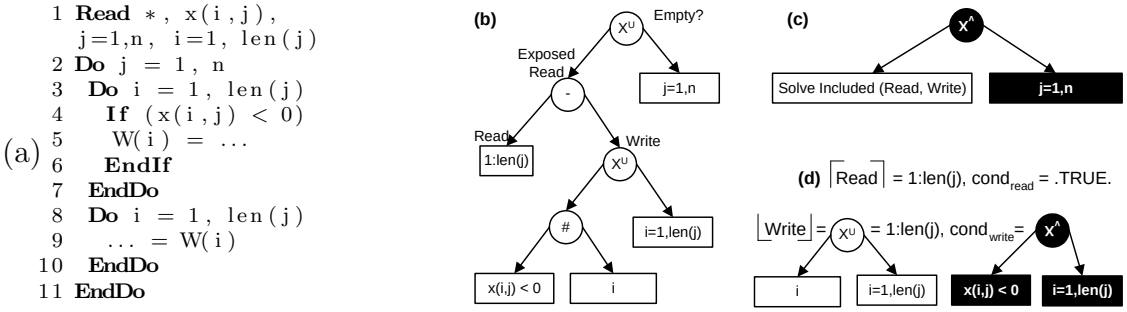


Fig. 31. Extraction of an independence predicate using approximation.

Several memory reference analysis techniques have proposed the use of approximations of reference sets in the presence of subscript arrays or arrays of conditionals [162, 161, 3]. These techniques generally approximate a memory reference set P that does not fit a particular model with a pair $(\lfloor P \rfloor, \lceil P \rceil)$ such that $\lfloor P \rfloor \subseteq P \subseteq \lceil P \rceil$ and $\lfloor P \rfloor$ and $\lceil P \rceil$ fit their model. We apply this to our framework by approximating complex USRs with predicated LMADs.

Returning to the example in Fig. 31, when trying to prove array W privatizable we cannot compare the USRs of the *read* and *write* descriptors directly. Instead, we compute $\lceil read \rceil$ and $\lfloor write \rfloor$ as LMADs and record the assumptions made during the approximation process. The problem reduces to proving $\lceil read \rceil \subseteq \lfloor write \rfloor$. Since $read \subseteq \lceil read \rceil$ and $\lfloor write \rfloor \subseteq write$, this condition is sufficient to prove that $read \subseteq write$. In our example, $\lceil read \rceil = [1 : len(j)]$, and $\lfloor write \rfloor = [1 : len(j)]$, when $\otimes^{\wedge}_{i=1,len(j)} x(i,j) < 0$. The approximation process is invoked by algorithms *Solve Disjoint Approximations* and *Solve Included Approximations* shown in Fig. 32.

The approximation algorithm (not shown) is based on a recursive descent on the USR structure. When looking for underestimates, the algorithm makes choices that maximize the size of the result, and when looking for overestimates, it minimizes. In the example in Fig. 31, the underestimate of *write* is maximized optimistically and ends up covering the overestimate of the *read*. In general, this aggressive approach

Algorithm Solve Disjoint Approximations
Input: A, D as USRs
Output: P as PDAG s.t. $P \Rightarrow (A \cap D = \emptyset)$
$(cond_A, \lceil A \rceil)$ = a conditional LMAD overestimate of A
$(cond_D, \lceil D \rceil)$ = a conditional LMAD overestimate of D
$P = cond_A \wedge cond_D \wedge SolveDisjointLMADs(\lceil A \rceil, \lceil D \rceil)$

Algorithm Solve Included Approximations
Input: A, D as USRs
Output: P as PDAG s.t. $P \Rightarrow (A - D = \emptyset)$
$(cond_A, \lceil A \rceil)$ = a conditional LMAD overestimate of A
$(cond_D, \lfloor D \rfloor)$ = a conditional LMAD underestimate of D
$P = P \vee (cond_A \wedge cond_D \wedge SolveIncludedLMADs(\lceil A \rceil, \lfloor D \rfloor))$

Fig. 32. Algorithms to extract a PDAG from a USR identity based on USR approximation.

increases the chances of extracting nontrivial conditions.

c. Predicate Extraction from Finite Valued USRs

USRs are symbolic sets that depend on the value of program variables. When a variable may take a known, limited number of values (possibly symbolic) we can partially evaluate the USR for all these possible values. Then the USR can be represented as a union of all its specialized versions, each guarded by its assumption. Consider USR $d = \{f(MOD(i,2))\}$. Then $d = (MOD(i,2) = 0)\#\{f(0)\} \cup (MOD(i,2) = 1)\#\{f(1)\}$. based on the fact that intrinsic $MOD(*,2)$ may take only one of two values, 0 or 1. Assuming that $f(j) = j/2$, the USR reduces to $\{0\}$ at compile-time. Similarly, a set difference $\{j\} - \{k\}$ can be expressed as $(j.NE.k)\#\{j\}$, which does not solve the problem at compile-time, but *leaves less to be done at run time.*

When the number of values taken by an input sensitivity variable is not known at compile-time, we can still enumerate a small set of important cases followed by a fallback solution.

An extreme case is when complex subscripts are created within the program, resulting in nontrivial USRs with empty input sensitivity sets. In such a case we

generate executable code from PDAGs and run it at compile-time.

d. Extracting PDAGs from LMAD Equations

When the recursive descent on USRs reaches leaves, we have to extract conditions from equations involving linear intervals. Although in general these are hard problems even for linear memory reference descriptors like the LMAD [161, 159], most practical cases are tractable. We have modified the multi-dimensional LMAD intersection and subtraction algorithms presented in [161] to return sufficient conditions under which their result is empty. For instance, the problem of proving two 1-dimensional LMADs disjoint, is equivalent to a bounds check and a GCD test.

3. Testing Monotonicity and Disjoint Intervals

```
1 Do j = 1, n
2   Do i = 1, len(j)
3     A(ptr(j)+i) = ...
4   EndDo
5 EndDo
```

Fig. 33. Example of a case where a sorting based test is more accurate than applying the *Solve* algorithm.

Consider the dependence problem on array A in the example in Fig 33. A direct application of the *Solve* algorithm would result in a test of $n*(n-1)/2$ bound checks, one for each pair ([ptr(j)+1:ptr(j)+len(j)], [ptr(k)+1:ptr(k)+len(k)]), where $j = \overline{1:n}$ and $k = \overline{1, j-1}$. However, a less expensive solution exists for this case: We can verify, dynamically, in $O(n\log(n))$ time, that the sequence $\lceil D_i \rceil = [lower_i : upper_i]$ is non-overlapping by sorting the pairs $lower_i : upper_i$ (based on $lower_i$) and verifying that $upper_i < lower_{i+1}$. A quicker ($O(n)$) and sufficient but not necessary version of the test verifies whether the intervals already form a monotonic sequence.

It is important to note that n may be much smaller than the actual number of dynamic memory references since it represents the number of partially aggregated intervals, rather than individual references. We extended the applicability of this test to multi-dimensional LMADs by defining order in multi-dimensional integer spaces.

Sorting-based tests are generated whenever the per-iteration reference set can be bounded by a symbolic interval.

4. Reference Pattern Library: Extensible Compiler

Recognizing and taking advantage of particular patterns such as sortable intervals will always provide better solutions to some classes of problems, since the programmer's level of abstraction is often above the programming language semantics. Pattern recognition presents two main challenges. First, a pattern must be general enough so that two semantically equivalent patterns will be recognized as such even when they are textually different. Second, the pattern recognition must be quick in order to be applicable (pattern recognition in programs is often associated to subgraph isomorphism).

We have created an offline XML database of memory reference patterns as USRs. These patterns can be analyzed by programmers and can be associated specialized library routines for a particular analysis, such as sorting-based checks for dependence analysis. A dependence test can then be broken up into parts for which solutions are known, and an overall solution can be composed using the general algorithm *Solve*.

By storing patterns as USRs, the possibility of finding a match in the library is greatly enhanced. Aggregation and normalization bring textually different memory reference patterns to a comparable form. We have identified several similar patterns across different subroutines within the same program and even between completely different programs. *Semantic* reference pattern matching reduces (partially) to *syn-*

```
 1 Read *, n, x,
   (p(j), l(j), (y(i,j),
    i=1,l(j)), j=1,n)
 2 Do j = 1, n
 3    Call geteu(W, j)
 4    Do i = 1, l(j)
 5       A(p(j)+i) = W(i)+
           + 1/A(p(j)+i)
 6    EndDo
 7 EndDo
...
...
```

```
 8 Subroutine geteu(W, j)
 9 If (x.EQ.0)
10   Do i = 1, l(j)
11     W(i) = ...
12   EndDo
13 Else
14   Do i = 1, l(j)
15     If (y(i,j).GT.0)
16       W(i) = ...
17     EndIf
18   EndDo
19 EndIf
```

Fig. 34. Example extracted from DYFESM, loop SOLVH_do20. The loop at line 2 can be executed in parallel if and only if there are no cross-iteration dependences on arrays W and A.

tactic pattern matching on the USR grammar, which can be implemented efficiently.

5. Fallback: Reference-based Dependence Tests

Extraction of equivalent simple conditions from dependence equations is not always possible, for instance in the example in Fig. 30. We have two generally applicable solutions that can solve arbitrarily complex dependence equations. In case the aggregation process was partially successful, we can embed the USRs in the generated code and evaluate them at run time [3]. The run time dependence test will consist of checking whether the result is empty. When (partial) aggregation and predicate extraction is not possible we fall back to the LRPD test [129], which has a complexity proportional to the dynamic memory reference count, but scales well with the number of processors.

6. Case Study

The HDA process for the loop at line 2 in Fig. 34 is shown in Fig. 35. Block (a) shows the dependence set resulting from the invocation of algorithm *Build Dependence*

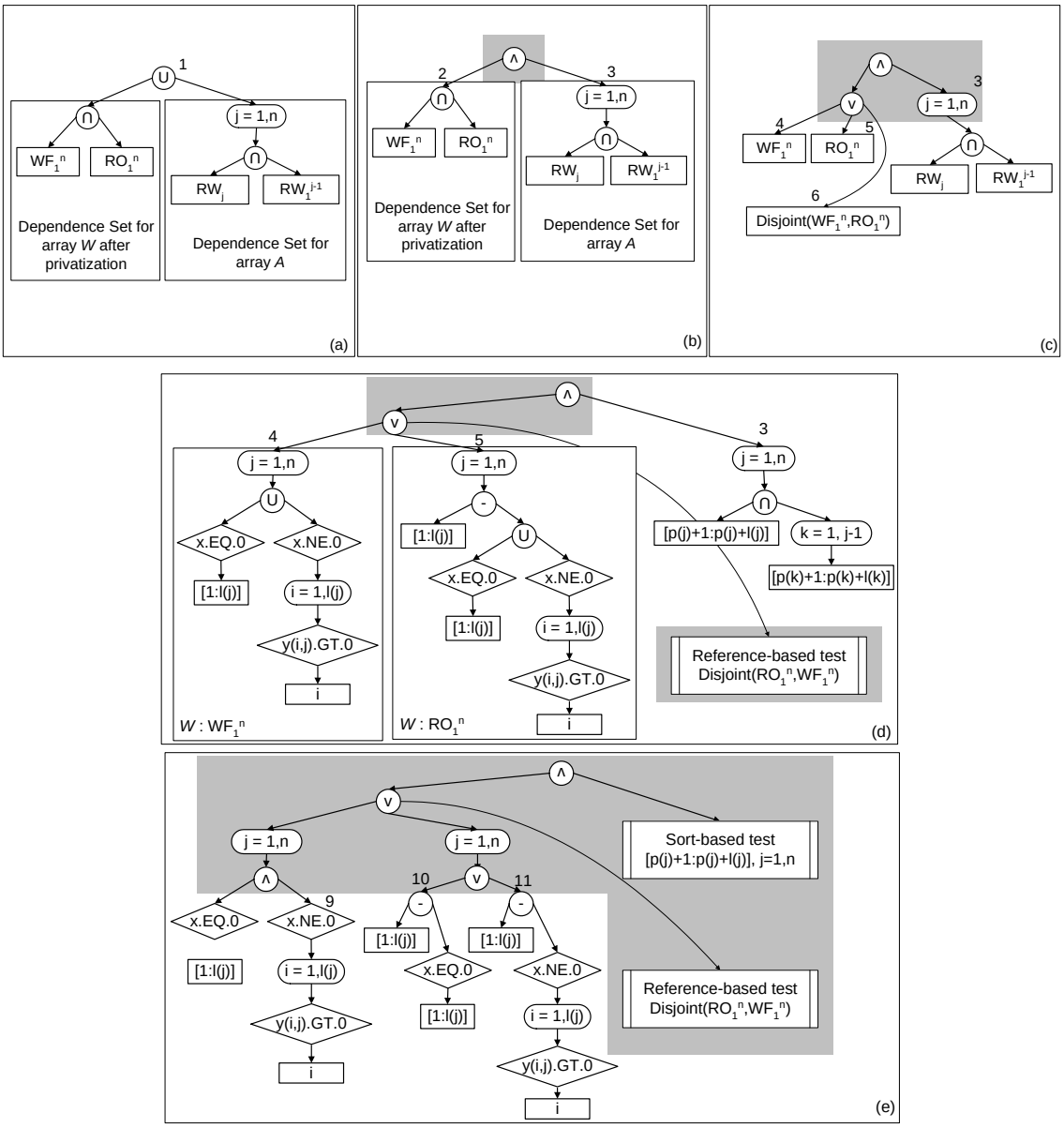


Fig. 35. PDAG extraction from the parallelization problem for the loop at line 2 in Fig. 34 (partial PDAGs are shaded). The numeric labels represent dependence equations $DS = \emptyset$, where DS is the corresponding node. For instance, equation 2 in block (b) has as DS the dependence set for variable W.

Set for variables W and A (empty descriptors such as RW for W are not shown). (b) Problem 1 is divided into subproblems 2 and 3 by applying algorithm *Solve*. (c) Problem 3 is divided into subproblems 4 and 5 by applying algorithm *SolveDisjoint*. Because problems 4 and 5 are sufficient but not equivalent to problem 3, algorithm *SolveDisjoint* will add the fallback solution, a reference-based test on array W. (d) Problems 3, 4 and 5 are detailed by showing the exact shape of their USRs. (e) Algorithm *Solve* transforms problem 4 into the conjunction of problems 8 and 9 over the iteration space of the loop. Algorithms *Solve* and *SolveIncluded* transform problem 5 into a disjunction of problems 10 and 11. Problem 3 is recognized as a pattern by algorithm *SolveDisjoint* and is assigned a library routine solution.

In Fig 36 (f) problems 8 and 9 are transformed in simple equivalent conditions by applying algorithm *Solve* recursively. Problem 10 is transformed in condition $(x.EQ.0)$ and problem 12. Problem 12 will then reduce to *true* at compile-time after it undergoes the approximation phase in algorithm *SolveIncluded* and is applied algorithm *SolveIncludedLMADs*. Similarly, problem 11 reduces to problem 13, which is also solved by the approximation phase in *SolveIncluded* followed by *SolveIncludedLMADs*. (g) The final result is shown after symbolic simplification and hoisting of loop invariants.

It is important to note that a fully automated analysis technique of USRs and PDAGs produced run-time tests that have a clear meaning to a programmer. The independence conditions on W shown in the boxes in block (g) can be identified as (1) the absolute lack of write references and (2) the absolute lack of exposed reads. The second one is the actual behavior of the application and a common pattern observed in several applications. This makes us believe that HA operates at the right level of abstraction and thus manages to extract high semantics such as data dependence from low-level program representation, similar to the way a programmer would do.

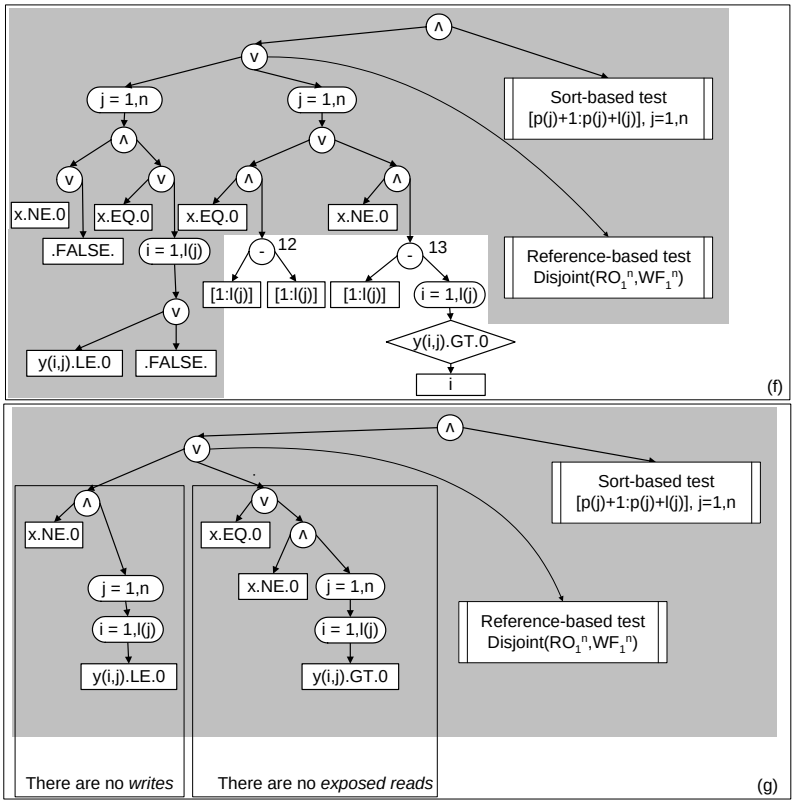
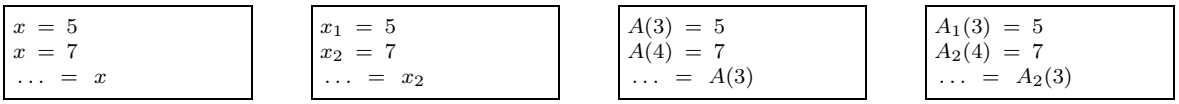


Fig. 36. PDAG extraction from the parallelization problem for the loop at line 2 in Fig. 34 (continued from Fig. 35).

E. Other Applications of Memory Reference Analysis

1. Array Data Flow Analysis

A large class of optimization decisions depend on the compiler's ability to determine with accuracy the characteristics of the flow of values in the program. Specifically, for a given *use* of a value (a variable name at a point in the program text), it is crucial to know its exact *definition* site (the point in the program text where the value was defined).



(a) (b) (c) (d)

Fig. 37. (a) Scalar code, (b) scalar SSA form, (c) array code and (d) improper use of scalar SSA form for arrays.

a. Region Array SSA

Static Single Assignment (SSA) is a program representation that presents the flow of values explicitly. In Fig. 37(a), the compiler must perform control flow analysis to find out which of the two values, 5 or 7, will be used in the last statement. By numbering each static definition and matching them with the corresponding uses, the use-def chains become explicit. In Fig. 37(b) it is clear that the value used is x_2 (7) and not x_1 (5).

Unfortunately, such a simple construction cannot be built for arrays the same way as for scalars. Fig. 37(d) shows a failed attempt to apply the same reasoning to the code in Fig. 37(c). Based on SSA numbers, we would draw the conclusion that the value used in the last statement is that defined by A_2, which would be wrong. The fundamental reason why we cannot extend scalar SSA form to arrays directly is that an array definition generally does not kill all previous definitions to the same array variable, unlike in the case of scalar variables. In Fig. 37(c), the second definition does not kill the first one. In order to represent the flow of values stored in arrays, the SSA representation must account for individual array elements rather than treating the whole array as a scalar.

Element-wise Array SSA was proposed as a solution by [163]. Essentially, for every array there is corresponding *@ array*, which stores, at every program point and for every array element, the location of the corresponding reaching definition

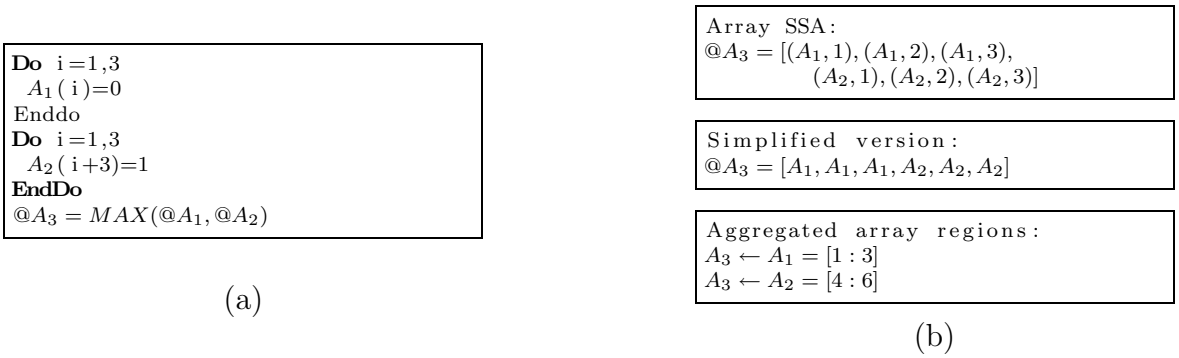


Fig. 38. (a) Sample code in Array SSA form (not all gates shown for simplicity). (b) Array SSA forms: (top) as proposed by *citeknobe.popl.98*, (center) with reduced accuracy and (bottom) using aggregated array regions.

under the form of an iteration vector. The computation of *@ arrays* consists of lexicographic *MAX* operations on iteration vectors. Although there are methods to reduce the number of *MAX* operation for certain cases, in general they cannot be eliminated. This led to limited applicability for compile-time analysis and potentially high overhead for derived run-time analysis, because the *MAX* operation must be performed for each element.

We propose a new Region Array SSA representation. Rather than storing the exact iteration vector of the reaching definition for each array location, we just store the SSA name of the reaching definition. Although our representation is not as precise as [163], that did not affect the success of our associated optimization techniques. This simplification allowed us to employ a different representation of *@ arrays* as aggregated array regions. Fig. 38 depicts the relation between element-wise Array SSA and our Region Array SSA. Rather than storing for each array element its reaching definition, we store, for each use-def relation such as $A_3 \leftarrow A_1$, the whole array region on which values defined at A_1 reach A_3.

We use the USR [3] representation for array regions, which can represent uniformly arbitrarily complex regions. Moreover, when an analysis based on USRs can-

not reach a static decision, the analysis can be continued at run time with minimal necessary overhead. For instance, in the example in Fig. 38, let us assume that the loop bounds were not known at compile time. In that case the *MAX* operation could not be performed statically. Its run time as proposed by [163] would require $O(n)$ time, where n is the dimension of the array. Using Region Array SSA, the region corresponding to $A_3 \leftarrow A_1$ can be computed at run time in $O(1)$ time, thus independent of the array size. Our resulting Region Array SSA representation has two main advantages over [20]:

• We can analyze many complex patterns at compile time using symbolic array region analysis (essentially symbolic set operations), whereas the previous Array SSA representation often fails to compute element-wise MAX operations symbolically (for the complex cases).

• When a static optimization decision cannot be reached, we can extract significantly less expensive run time tests based on partial aggregation of array regions.

b. Transformations Based on Data Flow Analysis

A number of important classic scalar transformations can be extended to arrays using MCA partitions directly, or via Region Array SSA: constant propagation, global common subexpression elimination or live variable analysis.

Additionally, array data flow information expressed using USRs can be used to implement hybrid array privatization and to reduce the amount of memory that needs to be copied during program checkpoints.

2. Efficient Recompilation

To be profitable, the overhead of dynamic compilation must be smaller than the benefits it brings. This simple rule has been enforced traditionally by detecting *hot*

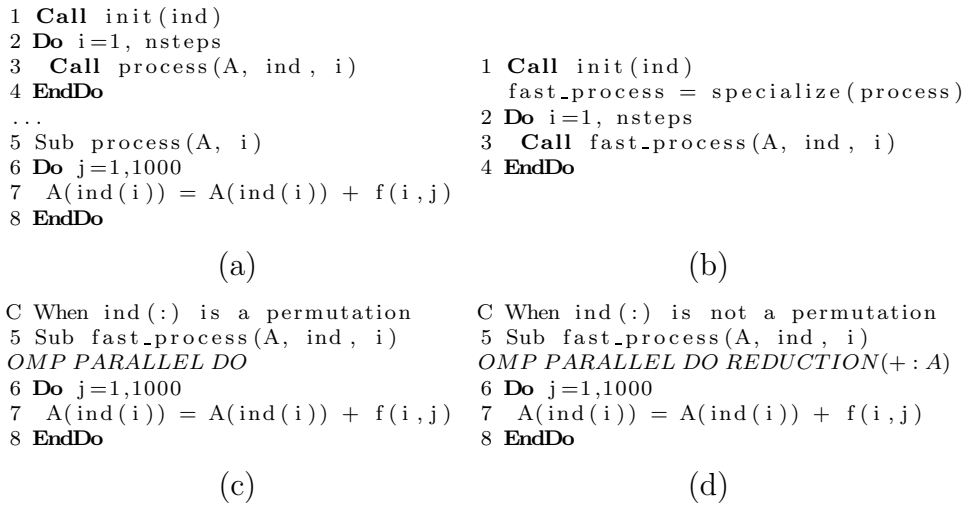


Fig. 39. Dynamic optimization through recompilation after specialization. (a) Sequential code. (b) Dynamic compilation through specialization, (c) when *ind(:)* is found to be a permutation, and (d) when *ind(:)* is found to contain repeating values.

spots, small program slices that are executed often. Hot spots are inexpensive to optimize because they are relatively small and their optimization has a great impact because they represent an important part of the total execution time.

However, hot spot based dynamic compilation was successful mostly for just-in-time compilation of otherwise interpreted programs. Since the compiled version is always faster than the interpreted one, the success is almost guaranteed. However, we consider here the problem of recompiling for optimization through *specialization* even when the original version was also compiled. In this case, hot spot based compilation cannot guarantee success anymore because the program slices selected as hot spots may or may not benefit at all from recompilation. Specialization is beneficial only when the slice input set has a positive impact on the optimization opportunities.

In addition to recognizing hot spots, it is crucial to recognize what program slices can be optimized and how often they need to be recompiled. We have shown how to

express several optimization problems as USR identities. We have also shown how to extract PDAGs from these identities. The PDAGs represent the *input sensitivity* of the optimization. They give the exact liveness range of a specialization. Consider the example in Fig. 39. If we knew that subscript array *ind* contained non-repeating values, we could classify the loop at lines 6-8 as independent and it could be executed in parallel. Otherwise, it would have to be executed as a parallel reduction, which is not as fast as a fully parallel loop. We can extract a PDAG that verifies the necessary properties of array *ind* exactly after its definition point, i.e. subroutine *init*. PDAG evaluation happens as soon as their input values are available. This way, we recompile only once, before the loop at line 2, rather than for every iteration of the loop at line 2. This particular example could be solved efficiently without recompilation by creating at compile time the two optimized versions and just selecting the right one at run time. However, the number of necessary versions is in general exponential in the number of dynamic decisions, which could make this approach impractical.

3. Program Verification and Symbolic Debugging

USRs provide a high level view of the memory reference pattern of a program. This property can be exploited by a high level debugger such as a data race checker for multithreaded program. For instance, a programmer may parallelize a loop using an OpenMP assertion, but may be unsure of the lack of data dependences, or whether an array should/could be privatized. All necessary race violations could be computed at compile- and run time and presented graphically using USRs.

The USRs could also be used to compute and display other memory related verification formulas such as uses of undefined array sections, at both compile- and run time.

F. Related Work

Data Dependence Analysis. Most of the previous data dependence work was based on the representation of memory reference sets using linear constraints. Dependence questions were reduced to proving that a system of linear constraints had no integer-valued solution [43, 49, 164, 45, 50, 165, 159, 15]. In all these systems, the symbolic expressions must be linear, although some particular extensions can handle certain classes of nonlinear references. They cannot generally be used to analyze (1) memory references through index arrays, (2) memory references controlled by arrays of conditionals and (3) memory references indexed or controlled by data values computed within the code section under analysis.

Pattern recognition and index property analysis were proposed as solutions for nonlinear reference patterns [74]. Its applicability is limited to the cases studied. Symbolic value range [54] and monotonicity analysis [39, 91, 74, 166, 57, 167] also targeted some classes of nonlinear reference patterns. They are generally not integrated well with other techniques and thus lack generality. For instance, the *Range Test* [54] compares the value ranges of two reference sets, but does not deal with strided patterns directly. We use value ranges and monotonicity information [54, 167] in a more general way, not only to compare offsets, but also strides and spans, and to prove predicate implication, redundancy or contradiction.

Run time data dependence tests were proposed to solve dependence problems that did not have compile-time solutions [141, 168, 143, 129, 169]. Their overhead may sometimes void the optimization benefits they bring. Our approach reduces the overhead by performing much of the analysis symbolically, at compile-time.

Hybrid Dependence Analysis and Parallelization. One of the first forms of hybrid analysis was conditional vectorization [170]. It is an effective technique, but

limited in scope to small loops. [148] presents a powerful interprocedural partial redundancy elimination analysis and its application to the detection of array data flow relations on particular control flow paths, which in turn leads to aggressively optimized placement of communication primitives, which is similar conceptually to hoisting USR computation to the data flow locations where their input variables become available [3]. HDA pushes symbolic analysis further and extracts PDAGs as cascades of conditions that are later hoisted in a similar fashion, which leads to even lighter run time tests. We cannot make a quantitative comparison with [148] because we targeted different classes of programs.

[31, 171, 32] synthesize simple conditions from data dependence and data flow equations on arrays. Their applicability is limited to checks on scalars such as loop bounds or scalar control flow values so they cannot extract predicates for general reference patterns through indirection arrays or arrays of conditionals. Their approach could be applied to solve cases such as the one in Fig. 1, but would fail to extract run time tests for cases such as the ones in Figs. 34 and 33. In such cases they choose to take conservative decisions. A similar approach of comparable symbolic power is presented by [161]. Safety guards are inserted to predicate optimistic results of statically undecidable LMAD operations. [139] showed how sufficient predicates can be extracted by simplifying Presburger formulas with uninterpreted function symbols. Although our implementations are different, they are fundamentally very similar. Unfortunately they did not apply it to real applications so we cannot compare the quality of the generated run time tests, which is what makes the difference in dynamic optimization methods. [3] uses USRs to express dependence tests but does not provide a way to extract simple run time tests. They propose the evaluation of USRs at run time followed by comparison to the empty set. However, in general a simple *Yes*/*No* answer is sufficient. The evaluation of USRs is generally not needed and it often results

in unnecessary run time overhead. For instance, the best speedup presented by [3] on MDG on 4 processors is 2.1, while our best is 3.7, albeit on different machines.

[172] focuses on reducing the overhead of reference-by-reference run time tests by grouping together reference sets that have the same dependence patterns. Only one representative test is performed, resulting in lower overhead. However, only accesses that have identical control and very similar indexing (e.g., differ by constant offset) are recognized as similar. The PDAGs can express much more complex relations between reference patterns and eliminate more classes of redundant checks. A decisive role is played by the USRs unification of apparently different patterns which would otherwise appear to be unrelated. Their best speedup on MDG on 4 processors is 1.7, while ours is 3.7 (though on different machines).

G. Conclusions

The advantage of Hybrid Analysis over traditional methods comes from its ability to use partial symbolic results. These results are often not sufficient to make a decision at compile time. On the other hand, they are ignored by run time methods, which redo the entire analysis process for each dynamic instance resulting in high overhead. Hybrid analysis extracts conditions from partially aggregated information which leads to run time tests of reduced complexity.

We implemented a full working Hybrid Optimization framework in the *Polaris* research compiler. Its backbone consists of an analytical representation for memory reference sets across arbitrarily large program contexts and of a predicate extraction technique that can extract sufficient conditions from identities involving sets of memory references. The entire analysis process is interprocedural and control-flow sensitive.

The following chapter presents a symbolic value comparison and logic reasoning module that is used to simplify USRs and PDAGs. Chapter VII presents a thorough validation of the impact of Hybrid Analysis to efficient automatic parallelization.

CHAPTER IV

SYMBOLIC VALUE ANALYSIS

A. Motivation

The efficiency of Hybrid Analysis depends, to a large extent, on the relevance of run time tests. A test that will always fail to prove anything will still use up time unnecessarily even if it is inexpensive. In order to extract relevant run time tests, we need to perform an as accurate as possible symbolic analysis at compile time. In addition to improving the quality of run time tests, this will also lead to more problems being solved completely at compile time.

All programs compute values which are described symbolically as variables. The values that some variables may take during execution are hard to predict; in general this problem is at least as hard as the halting problem. However, we are interested primarily in values that affect the memory reference pattern, i.e., play a role in computing the address of a memory reference. Such values are often computed in a simple way.

There are two types of symbolic values that may influence memory reference analysis, those that are used to compute indices and those that are used to predicate statements that compute addresses or statements that use the indices to reference memory. In the example in Fig. 40, the values in array W can be propagated from definition site 3 to use site 8 and thus eliminate costly multiplication operations. However, the compiler must be able to reason about the *range* of addresses referenced at the two statements. It has to prove that $m \leq 2 * m$. Although this is not generally true (e.g., $m = -5$), it is so when $m \geq 0$, which is the only interesting case anyway since for negative values the code in the loops does not get executed. Additionally,

```
 1 If (x<MAX(v(1),v(2)))
 2   Do i=1,2*m
 3     W(i) = 0
 4   EndDo
 5 EndIf
 6 Do i=1,m
 7   If (x<v(1+MOD(i)))
 8     ... = W(i)*3
 9   EndIf
10 EndDo
```

Fig. 40. Symbolic value analysis for comparison of addresses.

the compiler must prove that each of the conditions in the statement at line 7 implies the condition at line 1.

This type of symbolic reasoning about memory reference addresses and control predicates has two components. First, there has to be a mechanism to formulate optimization questions. We have already shown that the most important parallelization questions can be formulated as USR identities. Second, there has to be a mechanism to answer these questions. This section presents a symbolic mechanism to compare symbolic expressions that make up the address part of USRs and to prove relations between the predicates that represent the control.

When index functions are relatively simple expressions of the loop induction variables and the array references are not masked by a complex control flow, then the analysis is relatively straight forward. For example, if in a loop an array is indexed through an affine function of the loop induction variable and the references are control flow insensitive then the data dependence analysis can be performed accurately and, if possible and profitable, the loop can be parallelized.

Unfortunately, arrays are not always referenced in such a simple manner. Sometimes the values of the addresses used are not known during compilation, e.g., when the values of the addresses are read from an input file or computed within the program

(use of indirection arrays). In other situations although the addresses are expressed as a simple function of the loop induction variable, the control flow that masks the actual references makes it impossible to compute a closed form of the index variable and thus very difficult to perform any meaningful analysis.

However, memory reference analysis and subsequent loop parallelization, cannot be performed with sufficient accuracy when arrays are indexed by subscripts that cannot be expressed as a closed form of the loop induction variable. Arrays cannot be proved independent because their indices cannot be analyzed with classical data dependence techniques and indices of arrays (addresses) cannot be computed independently by each iteration (or processor). We propose the *Value Evolution Graph (VEG)* as a novel representation for the value flow of induction variables that cannot be expressed as a simple algebraic function of their loop index. We show how this technique can improve the accuracy of data dependence analysis, privatization and the recognition of certain classes of memory reference patterns, such as *push-back* sequences. We show how these improved techniques can lead to the automatic parallelization of a larger number of codes.

1. Background and a Motivating Example

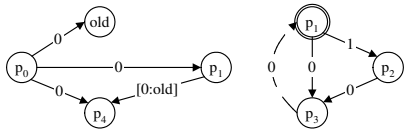
Recurrences with closed forms are those in which the i-th term can be written as an algebraic formula of i. In recurrences with closed forms most relations between values are proved using symbolic calculus. For example, references to arrays using recurrences with closed forms, can be meaningfully expressed using systems of linear constraints [173, 159, 160] or triplet-based notations [161, 174] containing the closed form terms and other symbolic values such as loop bounds. We will not address such recurrences in this chapter. When a recurrence with no closed form is used to index an array, the corresponding memory reference set cannot be summarized using an

```
 1 old = p
 2 q = 0
 3 DO i = 1, old
 4    q = q+1
 5    B(q) = 1
 6    IF (A(q).GT.0)
 7       p = p+1
 8       A(p) = 0
 9    ENDIF

10 ENDDO

11 sum = 0
12 DO i = old+1, p
13    sum = sum+A(i)
           +B(i-old)
14 ENDDO
```

(a)

```
 1 old = p0

 3 DO i = 1, old
      p1 = μ(p0 , p3)
 5    B(i) = 1
 6    IF (A(i).GT.0)
 7       p2 = p1+1
 8       A(p2) = 0
 9    ENDIF
      p3 = γ(p1 , p2)
10 ENDDO
   p4 = η(p0 , p1)
11 sum = 0
12 DO i = old+1, p4
13    sum = sum+A(i)
           +B(i-old)
14 ENDDO
```

(b)



(c)

Fig. 41. (a) Code sample, (b) in GSA after closed form substitution, (c) Value Evolution Graphs.

algebraic formula. For example, the algebraic expressions for the index of array A at line 8 in Fig. 41(a) for iterations k and *k+1* are identical, p, but their values always differ. Hence, we need to develop alternative analysis techniques that can deal with such cases. There are various uses for information about recurrence values. In the example in Fig. 41, we can find array A independent in the loop at line 3 if we show that $q < p$ and that the values of p are different in any two iterations that write to A. We can propagate the values stored in array A in the loop at line 3 to where they are used at line 13 if we know that the set of the definition indices covers the set of use indices. We can propagate the values in B if we know that $p \leq 2 * old$ (the value of p at statement 12).

2. Our Solution: The Value Evolution Graph

To solve the problems presented above we propose to model the value flow of the recurrences without closed form with the *Value Evolution Graph* (VEG) and use it

to obtain sufficient information to allow parallelization.

The reference pattern on array B in Fig. 41 (a) uses a recurrence with closed form $q(i) = i$, which was substituted in Fig. 41(b). It is easy to prove that there are no loop carried dependences on B because the index of B is expressed as an analytical function of the loop index. However, there is no such formula for the index of A because it is indexed by p, which is defined by a recurrence without a closed form (due to conditional incrementation). Fortunately, data dependence analysis does not require us to have closed form solutions, but rather to prove relations between the index sets corresponding to different iterations. In order to prove array A independent, we first need to show that statements 6 and 8 are independent. Note that at statement 6 we read from A at offsets between [1:old], and at 8 we write based on all the values of the recurrence on p. We can do it by finding all the values of the recurrence – its *image* – and prove that they do not intersect [1:old]. We also need to prove that statement 8 does not cause cross-iteration dependences by itself. We can do it by proving that the value of the index at line 8 always takes a positive *step*.

The *Value Evolution Graph* shown at the bottom of Fig. 41(c) translate the problems of computing the *step*, *image*, and *last value* of the recurrence within the first loop into graph problems. Although the idea of value flow in the program is not new [175, 88, 68], the VEG offers unique features and functionality needed by various analyses (Sec. B). We have integrated the VEG into our USR-based generic memory reference analysis framework that can thus solve multiple classes of optimization problems in the presence of recurrences without closed forms.

In this section, we will make two important points. First, we define the *Value Evolution Graph* that can represent the data flow in recurrences used as array indices which have no closed form solutions. The graphs are pruned based on control dependence predicates and produce tighter value ranges than abstract interpretation

methods. Second, unlike previous efforts of looking for patterns in the code text, we can analyze partially aggregated and classified memory descriptors. This single generic approach both extends and unifies in a single framework most cases which were previously solved using various, different, pattern matching techniques. It allows for the parallelization of important classes of memory reference patterns, e.g., pushbacks.

In the following section we will formally introduce the Value Evolution Graph (VEG), and present its use in Memory Classification Analysis. Then we will show how we have used it to perform more accurate dependence analysis, privatization and finally parallelization.

B. The Value Evolution Graph (VEG)

Finite recurrences are usually described by an initial value, a function to compute an element based on the previous one[1] (an *evolution* function), and a limiting condition. Depending on the evolution function's formula, in certain cases we can evaluate important characteristics even for recurrences without closed forms: the *distance* between two consecutive elements, the *image* of the recurrence, i.e. the set of all values it may take, and the *last element* in the sequence.

We introduce the *Value Evolution Graph (VEG)*, a compiler representation for the flow of values across arbitrarily large and complex program sections, including, but not limited to, recurrences without closed forms. Consider the loop at line 3 in Fig. 41. It performs a repeated conditional push to a stack array A. The stack pointer is stored in variable p. Due to the fact that p is incremented conditionally, there is no closed form for the recurrence that defines its value. We represent values

[1]We only address first order recurrences here.

as *Gated Static Single Assignment (GSA)* [10] names. In GSA, there are three types of ϕ-nodes. γ nodes merge two values on different forward control flow paths. μ nodes merge a loop back value with a loop incoming value. η nodes merge the outcome of a loop with the value before the loop. While this helps to discern between the values of p on the left and right hand side of the assignment at line 7 respectively, it does not differentiate between the value of p at line 8 in successive iterations. However, it makes it easy to determine that the stack array is written only at position p_2, and that p_2 is always the result of an addition of 1 to p_1. The subgraph consisting of $\{p_1, p_2, p_3\}$ (in Fig. 41(c)) represents the value flow between different GSA names for p in a single iteration of the loop. Each edge label represents the value added to its source to obtain its destination. The dashed edge carries values across iterations, but is not part of the VEG as it does not contribute to the flow of values within an iteration. We can employ well-known graph algorithms to prove that the distance between two consecutive values of p_2 is always 1, which makes the write to $A(p_2)$ be a stack push operation.

We will show how we construct the VEG in general, and how we run queries on it to compute recurrence characteristics over complex program constructs, such as loop nests, complex control flow, and subprogram calls.

1. Formal Definition

We define a *value scope* to be either a loop body (without inner loops), or a whole subprogram (without any loops). Immediately inner loops and call sites are seen as simple statements. We treat arrays as scalars and assume that programs have been restructured such that control dependence graph contains no cycles other than self-loops at loop headers. We have implemented such a restructuring pass in our research compiler.

Given a value scope, the Value Evolution Graph is defined as a directed acyclic graph in which the nodes are all the GSA names defined in the value scope and the edges represent the flow of values between the nodes.

In addition to the nodes defined in the value scope, we add, for every immediately inner loop, the set of GSA names that carry values outside the inner loop. An example is p_1 in Fig. 41.

Such nodes appear both in their current value scope graph as well as in the immediately outside value scope graph. They are called μ nodes in the context of the graph corresponding to the inner value scope and are displayed as double circles. Nodes representing variables assigned values defined outside their scope are called *input* nodes and are labeled with the assigned value (they are displayed as rectangles). The μ and *input* nodes are the only places where values can flow into a VEG. Values can flow out of the VEG through μ nodes only.

An edge between two variables p and q represents the *evolution* from p to q, defined as the function f, where $q = f(p)$. The evolution belongs to a scope if p and q are defined within the scope, and all symbolic terms in f are defined outside it. We represent four types of evolutions, additive and multiplicative for integer values and *or* and *and* for logical values. We represent an evolution by its type and the value of the free term. Certain evolutions can be composed along a path symbolically. For instance, the evolution along path $p_1 \rightarrow p_2 \rightarrow p_3$ is an additive evolution with value $1 + 0 = 1$. Instead of keeping a single value for an evolution, we store a range of possible values. This allows us to define an aggregated evolution from a node p to a node q as the union of the evolutions along all paths from p to q. For example, the aggregated evolution from p_1 to p_3 is [0:1], which represents the union of the evolution [0:0] along path $p_1 \rightarrow p_3$ and the evolution [1:1] along path $p_1 \rightarrow p_2 \rightarrow p_3$.

VEGs are as scalable as the GSA representation of the program since the number

Table III. Extracting evolutions from the program.

Statement	**Edge**	**Ev. Type**	**Label**
b_1 = a + exp	$a \rightarrow b_1$	+	exp
b_1 = a .OR. exp	$a \rightarrow b_1$	$\vee$	exp
b_1 = a * exp	$a \rightarrow b_1$	*	exp
b_1 = a .AND. exp	$a \rightarrow b_1$	$\wedge$	exp
b_1 = a	$a \rightarrow b_1$	Default	Identity
b_1 = exp	no edge, mark *input* node		
$b_2 = \gamma(b_0, b_1)$	$b_1 \rightarrow b_2$	Default	Identity
	$b_0 \rightarrow b_2$	Default	Identity
$b_2 = \mu(b_0, b_1)$	no edge, mark μ node		
$b_2 = \eta(b_0, b_1)$	$b_1 \rightarrow b_2$	Default	Loop effect
	$b_0 \rightarrow b_2$	Default	Identity
CALL sub($b_1 \rightarrow b_2$)	$b_1 \rightarrow b_2$	Default	*sub* effect

of nodes in all VEGs is at most twice the number of GSA names in the program and every node corresponding to a ϕ definition has the same number of incoming edges as the number of ϕ arguments. All other nodes have at most one incoming edge.

2. Value Evolution Graph Construction

Table III shows how we create edges from their corresponding statements. For now, we support only one evolution type per VEG. This evolution type is given by the first evolution we encounter, and is called the default type of the graph. If a value is computed in a way different from the ones shown in the table, we conservatively transform it into an *input* node and label it with $[-\infty : +\infty]$ (or [.FALSE.:.TRUE.]).

If it is computed in an assignment statement, then we try to find a closer range for the right hand side of the statement. We compute the aggregated evolution of an entire recurrence as the aggregated evolution, over all iterations, from the μ node to all nodes that may carry evolutions to the next iteration. We draw an edge from the value of the μ node to the corresponding value on the left hand side of the corresponding η definition, and we label it with the aggregated evolution of the inner recurrence. Fig. 41(c) shows such an edge between p_1 (a μ node in the inner recurrence $\{p_1, p_2, p_3\}$) and p_4. The range [0:old] is a result of multiplying the range of the aggregated evolution from p_1 to p_3, [0:1], with the iteration count of the loop, *old*. When values are obtained as a result of a subprogram call, we add edges to represent the aggregated value evolutions of the OUT actual arguments (and global variables) as functions of IN actual arguments (and global variables). In the last line in Table III, b_2 and b_1 are the OUT and IN arguments respectively.

The VEGs are built in a single bottom-up traversal of the whole program. The call graphs and the loop nest graphs of each program are traversed in reverse topological order. Within each scope we identify all definitions, build edges and associate input values. We use aggregated information from inner loops and called subprograms as shown in Table III. We compute the aggregated value evolution for all the recurrences associated with the loops using shortest/longest path algorithms that are linear in the size of the graph (number of edges + number of nodes). We compute the shortest and longest paths between every μ and *input* node and every other node. If every node is reachable from exactly one μ node and there are no *input* nodes, the complexity of the algorithm is linear in the number of GSA names + the number of arguments in all the ϕ nodes in the program. If more than one μ node can reach one same other node (coupled recurrences), the complexity may increase by a factor of at most the number of coupled recurrences.

3. Queries on Value Evolution Graphs

We obtain needed information about the values taken by induction variables by querying the VEG. All the queries we support are implemented using shortest path algorithms. Since all the VEGs are acyclic, these algorithms have linear complexity.

Given two GSA variables (possibly identical) and a loop, we can compute the range of possible values for the difference between the value of the second variable in some iteration $i+1$, and the value of the first variable in iteration i. For recurrences without closed forms, this computes the *distance* between two consecutive elements. In the example in Fig. 41, the distance between p_2 in iteration i and p_2 in iteration $i+1$ is exactly 1. This information can be used to prove that the write pattern on array A at statement 8 cannot cause any cross-iteration dependences. The value of the distance between a source node and a destination node across two consecutive iterations of a loop can be used for comparisons only if the destination node is not reachable from an *input* node.

Given a GSA variable and a loop, we can compute the range of values that the variable may take over the iteration space of the loop. For recurrences without closed forms, this computes their *image* and can be used to evaluate the *last element*. In the example in Fig. 41, the range for variable p_2 over the loop is [p_0+1:p_0+old]. This information is crucial for proving that the write pattern on array A at statement 8 cannot have cross-iteration dependences with the read pattern at statement 6 (they are contained in disjoint ranges [p_0+1:p_0+old] and [1:p_0] respectively). This information is computed in $O(d)$ time, where d is the depth of the loop nest between the given loop and the definition site of the given variable.

Given two GSA variables in the same subprogram, we can compare their values even if they are not in the same value scope, by comparing their ranges in a larger

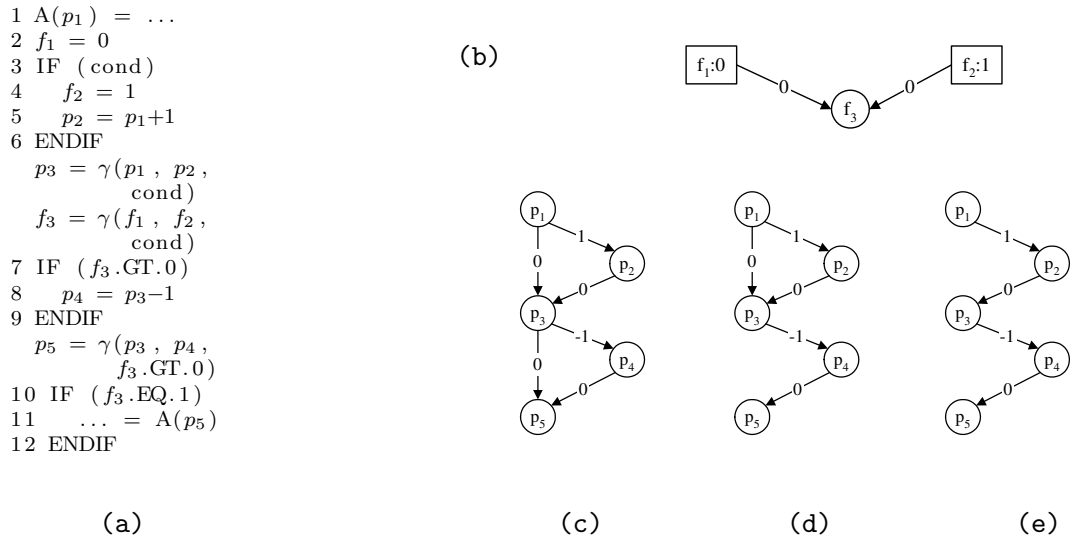


Fig. 42. (a) Sample code in GSA, (b) VEG for f_1, f_2, f_3; VEG for p_1, p_2, p_3, p_4, p_5 – (c) before pruning, (d) after pruning based on GSA Paths, and (e) based on range tracing.

common scope. This information can be used to prove either an order between their values or their equality and which in turn can be used in many compiler analyses.

4. VEG Conditional Pruning

We can prune a VEG by removing certain edges that cannot be taken when based on the truth value of a condition. The shortest path algorithms used to compute aggregated evolutions will then produce tighter ranges. Consider the code shown in Fig. 42 (a). Because we do not know anything about the value of *cond*, we cannot compare the values of p_1 and p_5, information that is needed to determine if the memory read at offset p_5 in array A is always covered by the write at offset p_1. Based on its corresponding VEG (Fig. 42 (c)), we can only infer that $p_5 \in [p_1\text{-}1{:}p_1{+}1]$.

The GSA path technique [93] describes how control dependence relations can be used to disambiguate the flow of values at γ gates. The GSA path technique can infer

that at line 11 condition $f_3.EQ.1$ holds true, which implies also $f_3.GT.0$ holds true. To the VEG, this means that value p_5 comes from p_4 and not directly from p_3. With the VEG pruned using this information (Fig. 42 (d)), we have $p_5 \in [p_1\text{-}1{:}p_1]$.

We have improved on [93] by using the VEG to trace back ranges extracted from given control dependence predicates. The read from array A at line 11 is guarded by condition $f_3.EQ.1$. This implies $f_3.EQ.1$ holds true. From this predicate, we extract the range [1:1] for f_3. In Fig. 42 (b), we trace this range for f_3 backward to see where it could have come from. Since the initial value for *input* node f_1 is 0, and the edge $f_1 \rightarrow f_3$ has weight 0, the only range that can be produced on the path $f_1 \rightarrow f_3$ is 0+0=0. The GSA gate f_3=γ(f_1,f_2,cond), associates the pair (f_1, f_3) with condition *.NOT.cond*. Since f_3 cannot come from f_1, *.NOT.cond* must be false, thus *cond* must be true. The same predicate, *cond*, controls the other gate, p_3=γ(p_1,p_2,*cond*). Since *cond* holds true, p_3 must have come from p_2, and not from p_1. So the edge $p_1 \rightarrow p_3$ cannot be taken. This leads to the graph in Fig. 42 (e). On the pruned graph in Fig. 42 (e), $p_5 = p_1$+1+0-1+0 = p_1, which proves the read at line 11 covered by the write at line 1.

This method improves on [93], leads to more accurate ranges than the abstract interpretation method used in [176], and can solve classes of problems that [88] cannot. One use of VEG conditional pruning is presented in Sec. D.

C. VEG-based Memory Reference Analysis

Fig. 43 shows the bottom-up analysis of the program context across lines 3-11 for a code snippet extracted from benchmark application TRACK. Note that the leaf node 1 represents a successfully aggregated memory access pattern (statements 4-6). The subtree rooted by node 3 represents the reference pattern across statements 3-

```
   Program main
1  Do k= 1, 100
     q1 = μ(q0 , q2)
2    old = q1
3    Call build(q1→q2)
4    ... = A(old:q2−1)
5  EndDo
...

   Function ten(b, e)
1  Do j = b, e−1
2    If (A(j)...)
3      Return .F.
4    EndIf
5  EndDo
6  Return .T.
7  End
```

```
   Sub build(p0→p5)
1  old = p0
2  Do i = 1,100
     p1 = μ(p0 , p4)
3    If (ten(old , p1))
4      Do j = p1 , p1+9
5        A(j) = ...
6      EndDo
7      p2 = p1+10
8    Else
9      A(p1) = ...
10     p3 = p1+1
11   EndIf
     p4 = γ(p2 , p3)
12 EndDo
   p5 = η(p0 , p1)
13 End
```

Context	LMAD	Node	U-estimate	O-estimate
5	j	–	j	j
4-6	$[p_1{:}p_1{+}9]$	1	$[p_1{:}p_1{+}9]$	$[p_1{:}p_1{+}9]$
3-7	–	3	$\emptyset$	$[p_1{:}p_1{+}9]$
9	p_1	2	p_1	p_1
8-11	–	4	$\emptyset$	p_1
3-11	–	5	p_1	$[p_1{:}p_1{+}9]$

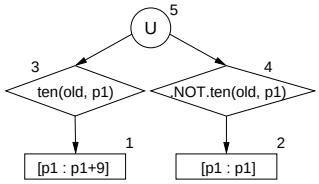


Fig. 43. Aggregation of WF across lines 3-11 for this code snippet extracted from PERFECT/TRACK/EXTEND_do400. The **LMAD** column shows the cases in which the descriptor can be represented as an LMAD. The **Node** column lists the label of the node that roots the corresponding USR in the figure on the right. The last columns show LMAD-based under- and over-estimates (as sets) for each USR.

7. The internal node 3 shows that the value of the conditional *ten(old, p_1)* – which controls the memory reference – is unknown. Throughout the aggregation process, every USR node is associated with a lowerbound and an upperbound (in the sense of set inclusion) using lists of LMADs. The *underestimate* is a list of LMADs that is completely contained in the USR. The *overestimate* is a list of LMADs that completely contains the USR.

1. Using the VEG in Memory Classification Analysis

We have shown (Section III.C.1) how MCA classifies all memory locations accessed within the context in *Read Only (RO)*, *Write First (WF)* and *Read Write (RW)*.

The most important memory classification process takes place at loop level (Fig. 45). For instance, in the example in Fig. 41, the WF pattern for array B within an iteration of the loop at line 3 is $\{i\}$. Across the entire loop, it is $\otimes_{i=1,old}^{\cup}\{i\}$ = [1:old]. When the recurrence has no closed form, these operations cannot be performed symbolically. However, we can use the VEG to detect *contiguous sequences* of memory locations indexed by recurrences without closed forms. These sequences, found by algorithm *ContiguousWrite* are used to adjusts the results of *McaLoopBlock*.

Consider the example in Fig. 43. Conceptually, the loop in program *main* performs a repeated pushback on array A, based on index q. The stack array A is also read at line 4 in program *main* and at line 2 in function *ten*. Both *reads* are to elements that have been pushed within the same iteration of the loop in program *main*, thus they are covered by *writes*. Consequently, array A is privatizable.

Traditional analysis fails because of the conditional incrementation of the index p by either 10 or 1. Recent work [74, 68, 166] focused on statement-level pattern matching of recurrence expressions. These approaches fail to relate the write to *A(j)* at line 5 in subroutine *build* to p. Also, they cannot handle the presence of *read* memory references and recurrences over multiple variables (q, *old*, p).

Our approach is to aggregate memory references symbolically using VEG information. Our addition to MCA does not require new data structures, as the new information is used to refine the existent *RO*, *WF*, and *RW* descriptors. We aggregate the reference pattern over the loop at line 4 in subroutine *build* into $[p_1{:}p_1{+}9]$.

We cannot aggregate the reference pattern over the outer loop in subroutine *build* because the recurrence on p_1 has no closed form. Regardless of the value returned by function *ten*, we can see that the write pattern is *contiguous*, i.e. it has no gaps between any two successive iterations. The write access pattern can be aggregated across the whole loop as $[p_0{:}p_5\text{-}1]$. At the beginning of any iteration i, the extent of the contiguously written section in previous iterations is $[p_0{:}p_1\text{-}1]$. The read from A at line 2 in function *ten* is always within $[\text{old}{:}p_1\text{-}1]$. We can prove it is covered by previous writes, since $\text{old} = p_0$. Also, we can find that the extent of the contiguous write for the whole loop is $[p_0{:}p_5\text{-}1]$. At the call site in program *main*, this translates into $[q_1{:}q_2\text{-}1]$, which covers the successive reads completely within every iteration. *We solved this MCA problem not based on the closed form of the index but rather on the information about the recurrence exposed by the VEG.*

In order to parallelize the loop in program *main*, we still have to prove that there are no cross-iteration output dependencies. We do it by proving that the per-iteration descriptor, $[q_1{:}q_2\text{-}1]$, is *increasing*, i.e. it has no overlaps. A VEG query is used to evaluate the step from $q_2\text{-}1$ to q_1 across two successive iterations and to prove it is positive.

Fig. 44 shows how the relations between USR and VEG operations.

2. Memory Reference Sequence Classification

A memory reference sequence is *increasing in a loop* if every access index in iteration $i+1$ is strictly larger than any index in iterations 1 to i (Fig. 46(a)). It is *contiguous in a loop* if it is contiguous within every iteration and, for any iteration i, its image over all iterations up to i is contiguous (Fig. 46(b)). It is *consecutive in a loop* if it is both contiguous and increasing in the loop (Fig. 46(c)). These definitions can be extended to strided memory access. These properties have to be proved true

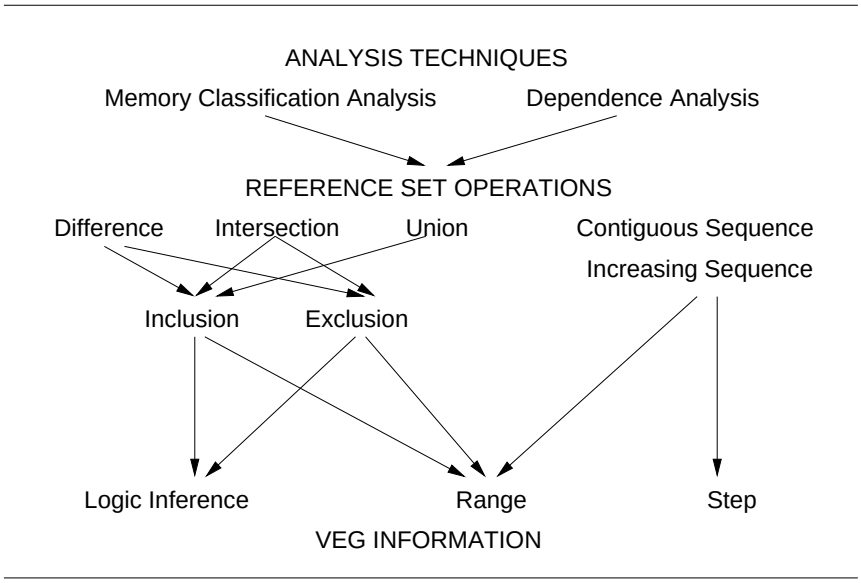


Fig. 44. Details on how the VEG information is used.

Algorithm McaLoopBlockCw()
Input: $(j = 1, n)$, (WF_j, RO_j, RW_j)
Output: (WF, RO, RW)
Call ContiguousWrite$(WF_i, RO_i, RW_i) \rightarrow CW$
Call McaLoopBlock$(WF_i, RO_i, RW_i) \rightarrow WF, RO, RW$
CALL UpdateCw$(WF, RO, RW, CW) \rightarrow WF, RO, RW$

Algorithm Update
Input: WF, RO, RW, CW
Output: WF, RO, RW
$RO = RO - CW$
$RW = RW - CW$
$WF = WF \cup CW$
END

Fig. 45. The integration of the search for contiguous write-first sequences in the Memory Classification Algorithm. We have modified the abstract intepretation phase at loop header level. The *McaLoopBlock* algorithmn is presented in Fig. 22.

across all control paths.

We use VEG information to measure and compare the extent of memory reference sets and recurrence steps. This analysis is control-flow sensitive. In order to prove a sequence contiguous, we show that on all paths, and under the same or implied conditions the step of induction variable (obtained from the VEG) is smaller or equal to the span of the memory reference, at the loop level.

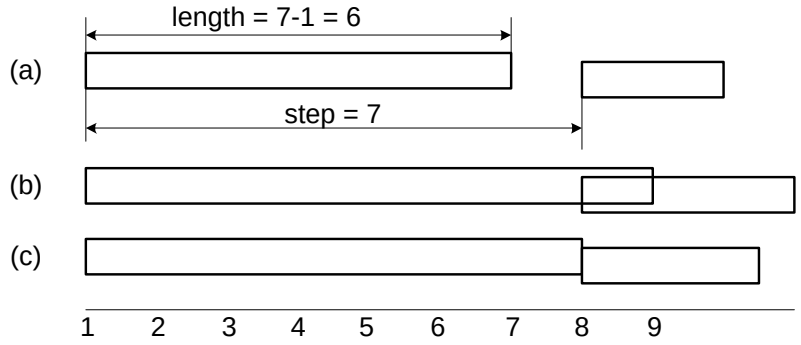


Fig. 46. Increasing (a), contiguous (b), and consecutive (c) reference patterns in a loop.

Table IV. Uses of memory reference sequence classification for the parallelization of the outer loop of a doubly nested loop.

	Sequence Class	**Context**	**Benefit**
1	Contiguous	*Inner*	Privatization
2	Increasing	*Outer*	Independence
3	Contigous	*Outer*	Efficient parallel code

3. VEG Applications to Classic Compiler Optimizations

Let us assume that we want to parallelize the outer loop of the nested loops *Outer* and *Inner*. Table IV presents the overall use of memory reference sequence classification in privatization and data dependence analysis.

a. Dataflow Analysis

We can use the WF, RO, and RW sets to prove general dataflow relations. For instance, a WF followed by a RO represents a def-use edge with weight $WF \cap RO$. This information can be used in transformations such as constant propagation. In the example in Fig. 41, we can prove that there is a def-use edge between lines 8 and

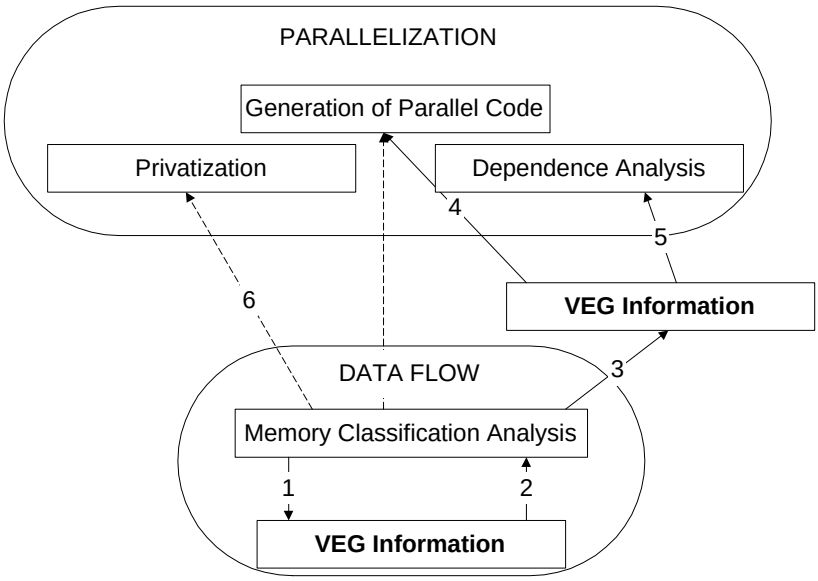


Fig. 47. Integration in the Hybrid Analysis framework.

13 on array A, with weight [old+1:p_4]. We can thus propagate all constant array values at offsets within this range.

b. Privatization

The privatization transformation benefits from memory reference sequence classification indirectly. The refined WF, RO, and RW sets for *Inner* will result in refined RO_i, WF_i, and RW_i sets for *Outer*. leading to more opportunities for privatization. This corresponds to edge 6 in Fig. 47, and to row 1 in Table IV.

c. Dependence Analysis

Let us assume that we have the descriptors RO_i, WF_i, and RW_i for *Outer*. If we can find a memory reference sequence d that includes them and is increasing in *Outer*, then there can exist no cross iteration data dependences. This corresponds to edges 3 and 5 in Fig. 47, and to row 2 in Table IV.

4. Recognition of Pushbacks and Other Parallelizable Prefix Computations

Many programs access arrays in loops according to patterns that are determined by loop induction variables. Even though induction variables are computed by recurrences, there are many important cases in which such loops can be executed in parallel. First, necessary conditions are that (i) there should be no data dependences between iterations of the loop except those involving the induction variable, (ii) there is no dependence cycle between the induction variable used as an address and the data computation, and (iii) it must be possible to compute the values taken on by the induction variable in parallel. Two cases in which the induction values can be computed in parallel are when the induction recurrence has a closed form solution or when it is associative; in the former case parallelization is trivial and in the latter case it can be done using a parallel prefix type computation [177]. Parallel prefix typically consists of three stages: (i) compute the local prefix sums of the associative induction variable, (ii) compute the prefix sums of the induction variable across processors, (iii) use the results of the cross-processor phase to compute the corresponding global values of the local indices, and then copy out the contents of the local arrays to their corresponding offset in the global array.

[178] addresses loops that contain the pattern *p = p+1; A(p) = ...* and where *p* does not appear anywhere else in the loop body, and parallelize them using a technique named "array-splitting," which is essentially a prefix computation.

In this work, we use the VEG to extend the applicability of the parallel prefix parallelization to more general types of loops that cannot be analyzed using pattern matching techniques alone. In particular, for loops with induction variables with no closed form solution, we impose the condition that the induction variable can only be

used as an address into an array, i.e., it does not contribute to the global data and/or control flow of the loop. In other words, if the induction variable is assigned to a shared variable or controls the execution of the program (e.g., used as an absolute inner loop bound) we will take the conservative approach and not parallelize it. An exception is made for the case when the value of the recurrence is used to test loop termination.

a. Pushback Sequences

We first consider loops which compute so-called *pushback sequences* that are generally defined as a sequence of consecutive write-first (WF) reference sets. In the following, we describe how we have used information provided by the VEG to extend the applicability of parallel prefix parallelization to pushback sequences. For illustration we use the code example in Fig. 43 which effectively performs a pushback on array A.

References to the pushback array have to be WF only. This implies that read accesses, to the array covered by the WF are allowed in any order. The WF set is computed accurately by the VEG improved MCA and thus qualifies more loops for parallelization. In the example in Fig. 43 we can see that the read at line 2 in function *ten* is always covered by a write in a previous iteration of the loop at line 2 in subroutine *build*, but within the same iteration of the loop in program *main*.

Most previous techniques analyze the patterns in which the induction variable appears, and from that try to infer which array addresses are used; this only works if there is a very simple (e.g., identity) relation between between induction values and array indices. We can qualify more loops as pushbacks since we can use VEG enhanced MCA to analyze more complex functions of the induction variable and

determine if the resulting index sequence satisfies the $step \leq length^2$ condition.

Information provided by the VEG can help us identify cases in which the induction variable does not contribute to the control flow, even if it would appear that it does using pattern matching techniques. For instance, in Fig. 43 the reference to A at line 2 in function *ten* is through j, which is the index of the loop at line 1 This loop has recurrence values as bounds. The looping statement can be normalized as *DO k = 1, e-b*. We use the VEG and evaluate the *(e-b)* loop bound and find that it does not depend on the induction variable of the outer loop, i.e., that we do not use the value of the induction variables of the loop we are considering (we use a local value).

We can parallelize loops where the recurrence value is also used as an early termination condition. Such cases are common for error checks such as stack overflow which usually result in premature loop exits. We execute the parallel prefix speculatively [179], compute the final value of the recurrence variable (before the termination) and then use it to copy out only the section of the private arrays that fits in the correct bounds.

b. Other Parallelizable Sequences

Using VEG enhanced MCA we can parallelize additional sequences with parallel prefix. Here are some interesting sequences we can recognize:

A sequence whose index is generated by a simple associative recurrence with *any* positive step such that $step > length$. In this case, the copy out phase will require that the computation of the indices into the global array be done in a more complex manner than for a pushback. Instead of using ranges of global addresses we have to

[2]The *step* of the recurrence versus the *length* of the memory access.

compute them individually.

Sequences whose index is generated by a more complex associative induction of some form $v = f(v, k)$ where f is an associative operator. In this case, VEG enhanced MCA can be used to guide the application of the set operations. (Although it is true the set operations themselves will be more complex, that is a symbolic manipulation problem that is beyond the scope of this paper.)

It is interesting to remark that when we do not deal with a simple pushback sequence, the parallel prefix computation of the recurrence value and the actual computation of the loop must be done, conceptually, in separate stages. Sometimes it is beneficial to perform in the local stage only the computation of the recurrence values and leave the remainder of the loop computation for the third phase of the parallel prefix. Other times, when the distribution of the recurrence computation implies a large amount of work duplication, it is beneficial to compute everything in the first phase in private storage and leave the actual address computation and copy out for the third phase. The compiler can use a simple work evaluation model to decide between the two alternatives.

D. Case Studies

Hybrid Analysis [3] integrates compile-time and run-time analysis of memory reference patterns. Its static part consists mainly of a framework for aggregation of memory references using the compact USR memory location set representation. This framework is used to perform Memory Classification Analysis which is used for automatic parallelization. We have integrated the information produced by VEGs in this framework – Fig. 47. Fig. 44 shows the relation between three levels of abstraction in the analysis process. High-level routines such as dependence analysis relies

Table V. Loops parallelized. CP = Conditional Pushback, SL(U) = Stack Lookup (and Update), P-CW = Privatization based on Contiguous Writes, P-VEG = Privatization using the VEG directly.

Program	**Loop**	**Seq. %**	**Description**
TRACK	EXTEND_do400	15-65	CP-SLU, P-CW
	FPTRAK_do300	4-50	CP-SL
	GETDAT_do300	1-5	CP-SLU, P-CW
P3M	PP_do100	52	P-CW, P-VEG
	SUBPP_do140	9	P-CW
BDNA	ACTFOR_do240	29	P-VEG
MDLJDP2	JLOOPB_do20	12	CP
ADM	DKZMH_do60	6	P-CW
QCD	QQQLPS_do21	< 1	CP
DYFESM	SETCOL_do1	< 1	CP
HYDRO2D	WNFLE_do10	< 1	CP

on memory reference set operations (such as intersection) and on the recognition of increasing memory reference sequences. These operations rely heavily on VEG information, such as step, range, or logical inferences. We implemented the VEG and integrated it with the Hybrid Analysis pass in Polaris.

Table V presents our results over codes *TRACK*, *BDNA*, *QCD*, *ADM* and *DYFESM* from the *PERFECT* benchmark suite, *P3M* from the *NCSA* suite, and *HYDRO2D* and *MDLJDP2* are from *SPEC92*. The third column shows the percentage of the total sequential execution of the program spent in the loop. The parallelization of these loops is crucial to the overall performance improvement in *TRACK*, *BDNA*, *ADM*,

P3M, and *MDLJDP2*. Although our new techniques can parallelize a larger number of loops, we only display results in addition to the ones obtained using traditional analysis techniques.

Seven out of the eleven parallelized loops were *conditional pushbacks*. The cases in *TRACK* are the most difficult as the arrays are not used as a stack at statement level, but only at the whole loop body level. We are not aware of any other static analysis that can parallelize any of these three loops. Six out of eleven loops required privatization analysis based on either *contiguous writes* or VEG information directly (value ranges).

Loop *BDNA/ACTFOR_do240* contains an inner loop that fills an index array *ind* with values within range [1:i], where *i* is the index of the outer loop. These values are then used to index a read operation on an array *xdt*. Since array *xdt* is first written in every iteration of the outer loop from *1* to *i*, this write covers all successive reads from *xdt(ind(:))*. The read pattern *ind(:)* is found to be completely contained in [1:i] based on the VEG range approximation for *ind*, which proves *xdt* privatizable. This pattern also appears on some arrays in *P3M/PP_do100*.

We also ran the analysis on the Barnes-Hut code TREE from the University of Hawaii in order to compare our results to previous work reported in [74]. This is an interesting case of an array that is used as a stack (push and pop operations) within an iteration of a loop, and is thus privatizable. However, the loop cannot contain cross-iteration dependences because the stack array is a local variable in a subroutine *treewalk* which is called from within the loop. Even if the code were inlined, our VEG-enhanced MCA would find the array privatizable in the outer loop.

```
1 DO i = 1, ny
2   DO j = 1, nx
3     DO k = 1, nz
4       p = k
5       CALL sr(A(p), inc)
6       IF (A(p).GT.0) GOTO 8
7     ENDDO
8     p = p + inc
9     DO k = p, nz
10      A(k) = ...
11    ENDDO
12    DO k = 1, nz
13      ... = A(k)
14    ENDDO
15  ENDDO
16 ENDDO
```

```
3 DO k = 1, nz
4   p1 = k
5   CALL sr(A(p1),
              inc3)
6   IF (A(p1).GT.0) GOTO 8
7 ENDDO
  p2 = η(p0, p1)
8 p3 = p2 + inc3
9 DO k = p3, nz
10    A(k) = ...
11 ENDDO
...
19 SUB sr(a, inc)
20 inc1 = 0
21 a = ...
22 IF (...) inc2 = 1
  inc3 = γ(inc1, inc2)
23 END
```

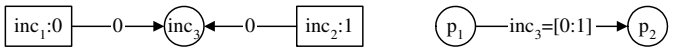


Fig. 48. Code extracted from *DKZMH_do60*.

1. ADM/DKZMH_do60

The loops at line 1 and 2 in Fig. 48 can be parallelized if we can show that array A is privatizable. We show that A has no exposed reads for the context between lines 3-14.

At lines 3-11, $WF = [1{:}p_2] \cup [p_3{:}\text{nz}]$. The distance between p_2 and p_3 is the value range for variable inc_3. This range was found by the VEG for subroutine *sr* to be [0:1]. This implies $p_2{+}1 \geq p_3$, so $WF = [1{:}\text{nz}]$. At lines 3-14, $RO = RO$ -WF $= [1{:}\text{nz}]$ - $[1{:}\text{nz}] = \emptyset$. $WF = [1{:}\text{nz}]$.

2. TRACK/EXTEND_do400

This loop is our most complex case, and was presented in detail as the loop in program *main* in Fig. 43. It consists of pushbacks performed in an inner loop. Another loop, inner to both of them, reads backwards the elements that were pushed within the same iteration of the outermost loop and, based on some condition, may modify some of these locations. It is crucial to prove that the access within an iteration of

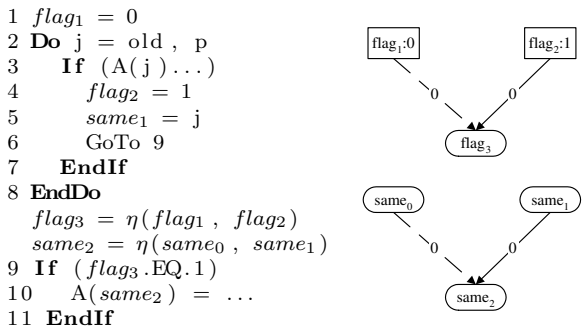


Fig. 49. Code extracted from *EXTEND_do400*.

the outer loop is confined to locations that were pushed within the same iteration. In addition to the problems discussed with relation to Fig. 43, this loop presents another problem that can only be solved using a conditionally pruned VEG.

This innermost loop is shown in Fig. 49. The range of the elements that have been pushed back within the current iteration of the outermost loop is [old:p]. The write reference at line 10 is at offset $same_2$. We must prove that $same_2$ is within [old:p]. A simple query on the range of values for $same_2$ on the VEG returns a range of [1:p] because it has to take into account the possibility that $same_2$ was not defined in the loop at line 2. The value could have come on edge $same_0 \rightarrow same_2$. Since $same_0$ could have carried a value from a previous iteration of the outermost loop, $same_2$ might not be confined to what was pushed in the current iteration. However, this edge is removed during the pruning of the VEG based on condition $flag_3$.EQ.1. Since a value for $flag_3$ could have come only on edge $flag_2 \rightarrow flag_3$, and since the pair ($flag_2$, $flag_3$) corresponds to the same control flow edge as ($same_1$, $same_2$) (based on GSA γ predicates), we can remove edge $same_0 \rightarrow same_2$. The VEG now evaluates the range of $same_2$ to [old:p] for the use at the statement at line 10.

Table VI. Comparison to recent work on memory referenced through recurrences without closed forms.

		Gupta et al [91]	**Lin, Padua [74]**	**Wu, Padua [68, 166]**	**Our Framework**
1	**Problems Solved**	Privatization, Data Dep.	Privatization, Data Dep.	Data Dep.	Privatization, Data Dep., Dataflow
2	**Method**	Memory Reference Analysis	Algorithm recognition	Monotonic evol.	Memory Reference Sequence Classif.
3	**Recurrence Model**	Implicit	Implicit: DDG	Explicit: evol.	Explicit: VEG
4	**Multi-variable**	Not specified	No	No	Yes
5	**Distance Ranges**	Yes	No	Yes	Yes
6	**Conditional Ranges**	Range extraction	No	No	Range extraction and tracing
7	**Mem. Ref. Type**	Generic	Single indexed	Not defined	Generic
8	**Interprocedural**	Yes	No	No	Yes
9	**Pushback Seq. Par.**	No	Yes (restrictive)	No	Yes (more general)

E. Related Work

1. Recurrence Recognition, Classification, and Parallelization

[87, 180, 88, 89] present the automatic recognition and classification of general recurrences. The idea of a value graph was introduced in [175]. Although similar to the SSA graph [88], the VEG adds more power and functionality to the representation: closure for the meet-over-paths operator using ranges and accuracy by pruning based on conditionals. [181] discusses the parallelization of linear recurrences. [182] and [96] present the recognition and parallelization of certain classes of recurrences but do not address cases when memory is referenced using the values of the recurrence. We express recurrence functions as paths in VEGs. Although in theory these techniques could cover more cases, in practice they are limited to recurrence formulae consisting of linear algebraic expressions coupled with conditionals. Since some of the edges in the VEG represent algebraic relations and others represent conditional execution, we

can also represent this mix of linear functions and conditionals.

2. Analysis of Memory Referenced by Recurrences without Closed Forms

[39] and [183] found more closed forms for classes of recurrences that had not been commonly recognized/substituted by compilers. [184] presents the parallelization of loop nests that may contain recurrences by flattening the nests into single loops and pre-computing the recurrences in inspector loops. This method may not be feasible when the recurrence depends on computation within the loop itself. [185] presents the use of monotonicity in reducing the number of bound checks for arrays referenced using a recurrence without closed form.

Let us follow (by row in Table VI) a comparison between our framework and the most recent work on the parallelization of loops that reference memory through recurrences without closed forms [91, 74, 68, 166].

In rows 1 and 2, [74] presents three algorithm recognition techniques that can be used for privatization and dependence analysis. [68, 166] used the concept of monotonic evolutions for data dependence tests. We introduce a single technique that covers all the problems solved by [74, 68, 166], has wider applicability, and, additionally, builds generic array dataflow information that can be used by other transformations (such as constant propagation). [91] uses monotonic information to improve memory reference set operation accuracy in a generic way, but does not recognize contiguous sequences. [68, 166] do not address privatization and [74] does it only based on specific algorithm recognition. Our analysis is generic and was used uniformly to the parallelization of loops *EXTEND_do400*, *FPTRAK_do300*, *GETDAT_do300* from *TRACK*, *DKZMH_do60*, *PP_do100*, *SUBPP_do140* and *ACTFOR_do240*. [74] cannot solve the privatization problems of the first four, and it solves the last three using two algorithm recognition methods.

As shown in rows 3, 4, 5 and 6, [68, 166] introduced the idea of evolution and a recurrence model that produces distance ranges. [91] extracts ranges from array indices as well as from predicates based on affine expressions. We believe that the VEG graph representation makes it easier to express aggregated evolutions by associating them with graph paths. These paths contain explicit evolution and control information (by using GSA). The VEG can model recurrences defined using multiple variables, unlike previous representations that rely on the statement-level pattern $i = i + exp$. The VEGs are pruned based on ranges extracted from conditional values, which leads to closer value ranges. The static parallelization of loop *EXTEND_do400* can only be decided on this pruned graph, and has not been reported before.

Rows 7 and 8 compare the genericity of the memory reference type handled by each analysis technique. [74, 68, 166] require that arrays be unidimensional and that the index expression consist of exactly the recurrence variable. The recurrence variable cannot appear in the loop text except for the recurrence statements and as an array index. Our framework is more flexible: we analyze partially aggregated generic memory descriptors that represent the reference pattern in a single statement, an inner loops or a whole subprogram uniformly. Loop *DKZMH_do60*, and loops *EXTEND_do400* and *FPTRAK_do300* reference memory in inner loops and via subroutines; some arrays are two-dimensional; in one case array elements are seen as scalars inside a called subprogram; in a few cases, the recurrence variable appears in the bounds of an inner loop, while the actual array index expression is the loop index.

Row 9 shows that our parallelization of *pushback sequences* is more general than the one presented in [74] where the important loops *EXTEND_do400*, *FPTRAK_do300* and *GETDAT_do300* from *TRACK* could not be parallelized.

Our work also led to improvements to the range techniques presented by [54, 176],

and to the GSA path technique presented by [93].

F. Conclusions and Future Work

The symbolic value information offered by the VEG is crucial to the efficiency of USR and PDAG-based analysis techniques. They solve more problems at compile-time, thus avoiding unnecessary run time tests. They also result in more meaningful and lighter run time tests, which leads to significant reduction in overhead, and thus overall performance increase.

For now, we treat arrays as scalars. We are planning to investigate the use of array dataflow information produced by MCA to create more expressive value evolution graphs.

We are also looking into further applications of value evolution graphs to the GSA path technique. Preliminary results show that, with minor improvement, we could solve more complex problems such as the compile time parallelization of loop *INTERF_do1000* in code *MDG* from the *PERFECT* suite.

CHAPTER V

ENGINEERING A HYBRID AUTOMATIC PARALLELIZER

A. Automatic Parallelizer Overview

We have implemented the automatic parallelization tool in the Polaris research compiler framework based on Hybrid Analysis. Our parallelization tool takes as input sequential Fortran 77 code and produces Fortran code with OpenMP parallelization directives.

In the example in Fig. 50, the sequential loop (a) was parallelized by inserting OpenMP directives before and after the loop. OpenMP is a source-level directive based parallelization language. Using OpenMP the programmer, and in our case the parallelizing compiler, can specify what loops are to be run in parallel, which variables must be privatized, the variables that participate in a reduction operation and other issues related to multithreaded execution.

We chose OpenMP due its wide acceptance by compiler and library providers. However, the Polaris internal representation is not hardwired to OpenMP directives. It is rather made of abstractions, represented as annotations, that may or may not correspond directly to OpenMP directives. There are thus two steps that are taken in the parallelization process. First, the parallelizer analyzes the program and builds the parallelization annotations. These annotations are then translated into OpenMP directives and, in some cases, modifications to the code.

The following sections present our run time parallelization design based on Hybrid Analysis. The last section presents in detail the parallelization abstractions and the way they are translated into OpenMP directives.

```
1 Do i=1, 100
2   tmp = f(i)
3   A(i) = A(i+100)+tmp
4   s = s+A(i)
5 EndDo
```

(a)

```
$OMP PARALLEL
$OMP+PRIVATE(TMP)
...
$OMP DO
$OMP+REDUCTION(+:S)
 1 Do i=1, 100
 2   tmp = f(i)
 3   A(i) = A(i+100)+tmp
 4   s = s+A(i)
 5 EndDo
$OMP END DO
...
$OMP END
```

(b)

Fig. 50. Loop parallelization. (a) Original sequential loop. (b) After parallelization using OpenMP directives.

Table VII. Comparison of parallel code generation strategies.

Strategy	**Advantage**	**Disadvantage**
Multiple versions	Fastest	Large code size
Dynamic generation	Small codes size	Recompilation penalty
Only parallel	Small codes size, fast parallel	Slower sequentially

B. Static vs. Dynamic Parallelization

When the loop is found parallelizable statically, no run time dependence tests are needed. Otherwise, we need to predicate the parallel execution to the run time value of the PDAG that represents the independence condition.

In the example in Fig. 51, the loop can be run in parallel only if the two arrays are respectively independent. There are three approaches (Table VII) to generating code that can be run either in parallel or sequentially depending on a dynamic value.

```
1 Do i=1,100
2   A(p(i)) = A(i)
3   B(q(i)) = B(i)
4 EndDo
```

(a)

```
  isLoopParallel = isParallel_A AND isParallel_B
  mustRestoreTc = false
  If (NOT isLoopParallel)
   mustRestoreTc = true
   tc = omp_get_num_threads()
   Call omp_set_num_threads(1)
  EndIf
$OMP PARALLEL
$OMP + PRIVATE(TMP)
$OMP DO
  1 Do i=1,100
  2   A(p(i)) = A(i)
  3   B(q(i)) = B(i)
  4 EndDo
$OMP END DO
$OMP END
  If (mustRestoreTc)
   Call omp_set_num_threads(tc)
  EndIf
```

(b)

Fig. 51. Run time loop parallelization. (a) Original sequential loop. (b) After parallelization using OpenMP directives.

First, we could generate two versions (sequential and parallel) at compile time. The right version will then be dispatched dynamically using a conditional jump. This approach would be fastest but it could incur exponential code increase when loop nests are parallelized at multiple levels. There are also other factors discussed over the following sections that could increase the number of versions further.

Another approach would be to generate parallel code at run time using a dynamic compiler. We have not explored this possibility due to the lack of dynamic compilation capabilities in our compiler framework. Although the overhead of recompilation could be large, this approach could work in cases where the code cannot be parallelized well otherwise. The compiler could be run concurrently with the application and the parallel code versions could be used as they become available. This approach could also be attractive when other issues such as run time privatization and reduction parallelization (see following sections) are factored in.

Rather than generating multiple (parallel/sequential) versions, we chose to just generate a parallel version. When the dependence test fails, the number of threads is set to 1 so the loop gets executed sequentially. The number of threads gets then restored to its original value, which is usually set by the user via a shell environment variable.

C. Dynamic Optimization Strategy

Our run time optimization (parallelization) model takes a code slice (loop) and produces an optimized (parallelized) version. The choice between the optimized (parallelized) and original version is made at run time based on the value of a run time test expressed as a PDAG.

1. Inspector/Executor

In most cases the value of the run time test can be computed before the loop is executed. For those cases we implemented an inspector/executor model conceptually similar to [141]. Rather than computing communication schedules we just produce a boolean value: parallel or sequential.

The inspector/executor model is relatively easy to implement. We find the point in the program where all the necessary values to compute the PDAG become available. If this point dominates the entry to the parallelized loop, then we can apply the inspector/executor strategy. We generate code to evaluate the PDAG and save its result in a boolean variable. The value of this boolean variable is then used in the run time test as shown in Fig.51(b).

If the data needed to compute the PDAG are not available before entering the loop, we extract a slice of the loop that computes just the necessary values. Unfor-

tunately, in some cases the computation in the slice can be so expensive that it may reduce significantly or even cancel the profitability of the optimization (parallelization).

For performance reasons, it is crucial that the inspector phase be parallelizable. Otherwise, the parallelization will not scale with the number of available processing units. Although all the operations required to evaluate PDAGs are parallel, it is possible that the slice that precomputes needed values cannot be parallelized (it could be a linked list traversal).

2. Speculation

When the values needed to compute the PDAG are not available before the loop we could extract a slice to precompute them. However, this can be very expensive. In the worst case, the whole loop could appear to be a strongly connected data flow graph that must be executed sequentially, and which produces a needed value at the end of the last iteration. In such a case the inspector is the whole loop, so no optimization is possible.

Fortunately there is an alternative that does not require precomputation of values needed by the PDAG. Instead, we can execute the loop *speculatively* in parallel and evaluate the PDAG along. The values required by the PDAG will always be available before the end of the loop, so we will know whether the loop was indeed parallelizable at the latest upon exit from the parallel section.

a. Checkpointing

In order to account for the case when the speculation failed, we must checkpoint the state of the program before entering the speculative section. In case of failure, the program state is restored to that before the speculative section, and the code is

reexecuted using the unoptimized (sequential) version.

We implemented an efficient checkpointing scheme based on array dataflow analysis such that only objects that may be modified within a loop, and which are live upon exiting the loop, are saved before executing the loop in parallel speculatively. We used the dataflow information produced by MCA.

3. Inspector/Executor vs. Speculative Execution

The choice between inspector/executor and speculative execution is either dictated by the data dependence relations or by a performance model. For many classes of access patterns there are parallel inspectors. Any access based only on precomputed subscript arrays, induction variables and per-iteration temporary variables leads to parallel inspectors. When an array `A` is written based on an index or conditional that contains references to `A`, there may exist a cycle between the computation and address. For arrays this situation cannot always be proven at compile time (though a linked list traversal can be proven). Then we have the choice to either distribute the loop and isolate the statements that are in the cycle or to use the speculative parallelization strategy [155, 1]. If we believe that the statements that potentially form a data dependence cycle are indeed sequential (e.g., linked list traversal) then speculative execution will fail and loop distribution is the better choice. The loop containing the cycle will be executed serially and its results will be used by the second, possibly parallel loop. When USRs are computed, they store references to the statements they were extracted from. These references are kept throughout the aggregation process. In case there is a dependence involving arrays, found as an overlap of two USRs, the statements referenced by the two USRs give us a superset of the dependence cycle.

When dependence cycles are not an issue, then the decision is based on the ratio between the execution time of an inspector loop and that of the entire loop. Small

```
 1 totalError = 0
 2 ptr = 0
 3 Do i=1,1000
 4  Do j=1,10
 5   W(j) = ...
 6  EndDo
 7  sum = 0
 8  Do j=1,20
 9   sum = sum+W(j)
10  EndDo
11  error = f(sum)
12  If (error<tolerance)
13   ptr = ptr+1
14   Result(ptr) = sum
15   totalError = totalError + error
16  EndIf
17 EndDo
```

Fig. 52. Example of a loop that can be run in parallel after removing dependences through privatization and after reduction and pushback parallelization.

inspectors seem to perform well. A more detailed discussion about these choices can be found in [130].

Regardless of the chosen strategy, the run-time overhead for dependence testing is reduced by the level of aggregation that our **HA** framework achieves.

D. Transformations to Remove Dependences

Consider the example in Fig. 52. Each iteration computes some values that are stored in array *W*. Their sum is then computed and stored in scalar *sum*, and used to compute an error measure, *error*. If the error is within some tolerance threshold, the sum is pushed to a result stack, *Result*, and the error is added to a total.

Based solely on data dependence analysis, the loop at line 3 should be declared sequential. There are possible cross iteration dependences on arrays *W* and *Result*, as well as on scalars *j*, *sum*, *error*, *ptr* and *totalError*.

Let us notice that variables *W*, *j*, *sum* and *error* are defined before being used in each iteration of the outermost loop. Our analysis also finds that their WF_i sets are

loop invariant. Essentially, that means that there are output dependences between each iteration, but these dependences can be eliminated by privatizing them, i.e., creating a private copy for each iteration. In practice, we create private copies for each execution thread rather than for each iteration. Our analysis also finds that their RO_i and RW_i sets are empty, except for W, where $RO_i = [11 : 20]$. This means that privatization removes all other cross iteration dependences. Since their liveness ranges are within a single iteration, there can possibly be no value flow across different iterations.

When privatizing array W, the parallel version will allocate additional storage per processor. Each thread will reference its private version of W wherever the sequential version would reference W. This could create two problems. First, the loop at line 8 uses elements of W that are initialized before the loop. Although those *reads* do not cause cross iteration dependences, they will use uninitialized data when reading from the private version of W. We must initialize the private versions by copying in from the original W at locations [11:20]. The second problem could happen if W is used after the loop. In that case, we must copy out the values at locations [1:10] from the private version of the thread that executed the last iteration to the original version.

Variable *totalError* has a nonempty RW descriptor, which means it has a cross iteration flow dependence. This dependence cannot be removed by privatization since there could be a flow of values across iterations in case the condition at line 12 is true for at least two iterations. However, the specific pattern of the computation of *totalError* allows us to make an algorithm substitution, widely known and accepted as *reduction parallelization*. A detailed discution of reduction parallelization can be found in [186].

Variables *ptr* and *Result* are part of another pattern that we classified using the Value Evolution Graph as a conditional pushback sequence. This pattern also has an

```
1 Do i=1,1000
2   Do j=begin(i),end(i)
3     W(j) = ...
4   EndDo
...
5 EndDo
```

Fig. 53. In order to parallelize the outer loop, privatization of array *W* may or may not be needed depending on the values of subscript arrays *begin* and *end*.

equivalent parallel counterpart that we substitute in automatically.

These transformations have been studied extensively and are well documented in literature. However, there are a few aspects that become interesting when they are performed in the context of dynamic parallelization. The remainder of this section presents in detail some of the challenges (and opportunities) of these transformations presented by the need (and capability) of making decisions at run time.

1. Hybrid Privatization

In the example in Fig. 52 array *W* is written at locations [1:10] in every iteration of the outermost loop. It is thus clear at compile time that it must be privatized to eliminate output dependences.

In other cases, it cannot be determined at compile time whether privatization is needed. Consider the case in Fig. 53. If $begin(i) = 10 * i + 1$ and $end(i) = 10 * i + 10$, statement 3 does not cause any dependences in the outer loop. However, if $begin(i) = 1$ and $end(i) = 10$, there are dependence between any two different iterations of the outer loop.

The conservative decision would be to always privatize when not sure. However, this could be highly inefficient. It may be that array *W* is initialized in independent blocks and its values will be used after the loop nest. The array may have a very

Table VIII. Comparison of run time privatization strategies.

Strategy	**Advantage**	**Disadvantage**
Multiple versions	Fastest	Large code size
Predicated	Smaller codes size	Predication penalty
Pointer-based	Smallest codes size	Pointer penalty
Dynamic compilation	Small codes size	Recompilation penalty

large size, so privatization might use a large amount of unnecessary storage and will perform unnecessary copy-in and copy-out. It would be better to actually analyze the subscript arrays before the loop and find out whether privatization is needed or not.

We express the privatization problem as a USR identity, which translates into a PDAG based on interval trees.

$$\otimes^{\cup}_{i=1,n}\left[WF_i \cap \left(\otimes^{\cup}_{k=1,i-1} WF_k\right)\right] = \emptyset$$

Unfortunately knowing at run time whether the variable is privatizable is too late. The code has already been generated at compile time. It is thus necessary to generate code that can switch at run time between using the original variable and using the private variable.

There are at least four possible designs (Table VIII). First, we can create two versions of the whole loop. This approach achieves the lowest run time overhead but may lead to increase in code size exponential in the number of run time decisions. Second, we could create predicated multiple versions just for the individual statements that may reference the privatized variable. This is our current implementation. This leads to a lower increase in code size but has possibly higher overhead. Third, we can replace all uses of the variable in the loop with a pointer which is set at run time before the loop based on the privatization decision. This could be the best approach

```
1 W(1) = ...
2 W(n) = ...
3 Do i=1,1000
4   Do j=2,n-1
5     W(j) = ...
6   EndDo
7     ... = W(1)
...
8 Enddo
```

Fig. 54. Privatization with copy-in. Only the ends of the array must be copied.

but it requires pointer manipulation which is unavailable in our Fortran 77 compiler framework. Finally, we could rely on dynamic code generation through recompilation. Although the overhead of recompilation is generally high, the quality of the generated code might make up for it for important loops that run for a long time and do not need to be recompiled often.

a. Hybrid Copy In

Copy-in and *copy-out* are operations required to maintain the private and shared (original) versions of a variable consistent. The conservative direction is to perform both copy-in and copy-out when unsure. However, being overly conservative is often suboptimal. Using USRs, we can express the exact array sections that must be copied in. In the example in Fig. 54, $Copy\ in = \{1, n\}$. In general,

$$Copy\ in = \left(\otimes^{\cup}_{i=1,n} RO_i\right) - \left[\left(\otimes^{\cup}_{i=1,n} WF_i\right) \cup \left(\otimes^{\cup}_{i=1,n} RW_i\right)\right]$$

b. Hybrid Copy Out and Last Value Assignment

Conceptually, *Copy-out* is the complementary operation of *Copy-in*. When performing privatization, the original shared variable must be updated after the loop using the values stored in the private versions. The update is required only if the

variable may be used after the loop.

There are two reasons why a variable may not be used after the loop. First, it may be that the variable is a temporary that is not referenced on any control flow path after the loop. This can be checked easily by traversing the control flow graph. Second, it may be that the variable is rewritten on all control flow paths before it is used again. In this case the values defined in the loop are not needed again. When the variable is scalar, this can also be checked relatively easily. When it is array, this is a complex array dataflow problem. Fortunately, we can use the MCA information to solve it efficiently.

While *Copy-in* incurs relatively small run-time overhead (time proportional to the amount of private data), *Copy-out* can be more complex. In case the last iteration does not write a private array completely, we have to check which previous iteration wrote it last.

This problem is known as determining the *last assignment* and it can be formulated in two ways. Traditionally, it was formulated as finding, for each memory location, the iteration vector and statement that wrote it last. This statement, at this iteration vector can then write directly to the shared rather than private version. This approach is very precise but quite complex when the loop spans multiple subprograms because iterations vectors become much more complex (require call stack information). We have also given a dual formulation to this problem. For each iteration, we compute the set of memory locations that can be written out as a USR. At the end of the loop, this set is written out to the shared variable by each iteration. Since all these sets are mutually disjoint, the *copy-out* operation is fully parallel.

$$Copy\ out = WF_i - \left(\otimes^{\cup}_{k=i+1,n} WF_i\right)$$

```
1 Do i=1,1000
2   Do j=begin(i),end(i)
3     R(j) = R(j) + ...
4   EndDo
...
5 EndDo
```

Fig. 55. In order to parallelize the outer loop, reduction parallelization on array W may or may not be needed depending on the values of subscript arrays *begin* and *end*.

2. Hybrid Reduction Parallelization

In the example in Fig. 55 it cannot be known at compile time whether the reference pattern on array R is fully independent or whether there are cross-iteration dependences for the outer loop. The outer loop can be parallelized in both cases because the operation on R is recognized as a reduction. However, reduction parallelization is significantly more expensive than a fully parallel loop.

[186] presents a more general discussion on how to parallelize reductions at run time using an adaptive system based on parameters such as the degree of sparsity of the contention matrix (elements X iterations). Our contribution here is that we can find out easily a particular case, i.e., when the reduction is trivial (independent update). We express this using a USR identity from which we extract a PDAG.

$$\otimes^{\cup}_{i=1,n}\left[RW_i \cap \left(\otimes^{\cup}_{k=1,i-1} RW_k\right)\right] = \emptyset$$

The conservative direction is to always execute as a reduction. However, if we know at run time that the pattern is actually independent, we can run a more efficient code version. The implementation is analogous to the one for run time privatization discussed in Table VIII.

```
 1 Sub compute(W, R, n)
 2 Dimension R(1000), W(*)
 3 Do i=1,1000
 4   Do j=1,n
 5     W(j) =   ...
 6   EndDo
...
 7 EndDo
...
 8 Program main
 9 Dimension R(1000), W(10000)
10 Read n
11 Call compute(R, W, n)
```

Fig. 56. In order to parallelize the outer loop, reduction parallelization on array *W* may or may not be needed depending on the values of subscript arrays *begin* and *end*.

3. Pushback Sequence Parallelization

We have discussed in Section IV.C.4 *pushback sequences* that are generally defined as a sequence of consecutive write-first (WF) reference sets. The fundamental effect of the transformation is that it eliminates all dependences caused by the pushback operation on the stack arrays and stack pointers.

Our pushback recognition based on USRs is more general than previous recognition methods which can deal only with textual matches (which are the trivial particular case when using USRs).

E. Automatic Detection of Array Bounds

In the example in Fig. 56, in order to parallelize the loop at line 3, we must privatize array *W*. If we just insert an OpenMP directive *PRIVATE* for *W*, the OpenMP compiler will generate an error message: *Cannot use PRIVATE with an assumed size array*. This message is caused by the fact that the OpenMP compiler cannot figure out the size of the array, thus it does not know how much memory to allocate for the private versions (and therefore the extent of copy-in and copy-out operations).

When the array does not have a LASTPRIVATE clause it is possible to find a conservative overestimation of the array size. The simplest, though practically useless is the whole address space. We can do better by tracing the array into the calling context. In our case, it is traced to array *main::W*, which is of size 10,000. However, it may be that the dynamic value of n is 10, and we would still overshoot by a factor of 1000.

Our algorithm first checks the USRs that describe the reference pattern within the loop. If this pattern is linear, we compute the bounds directly from the USR. If the pattern is nonlinear, we compute a linear approximation. We then compare this linear approximation against the conservative measure obtained by analyzing the originating storage within calling contexts. The minimum is used as new array bounds. When there are multiple loops in the same subprogram, we can either take the maximum across all loops, or allocate private variables explicitly for each loop, bypassing the OpenMP mechanism.

When the array does have a LASTPRIVATE clause, the exact size must be known. Overestimating it may result in writing over the bounds of the original array in the copy-out phase.

F. Case Study: DYFESM/MXMULT_do10

Fig. 57 shows an important computational core in benchmark application DYFESM (42% of the total sequential execution time). The main data structure, array *MX*, is divided into logical blocks corresponding to a physical discretization of the two dimensional object it models. Array *MX* contains one block for each physical discretization block. Each block in *MX* is made of all physical elements fully contained in its corresponding physical block. Additionally, *MX* contains a block that stores

```
 1 Do iss = 1, nss
 2    iloc = pptr(iss)
 3    neqi = iblen(iss)
 4    Call blckmx(iss, mx(iloc), mx(ilocb), x, neqi)
 5 EndDo

 6 Sub blckmx(iss, mxi, mxb, x, neqi)
 7 Call zerov(mxi, neqi)
 8 Do k = 1, nepss(iss)
 9    id = idbegs(iss)+k-1
10    ... // compute array 'mxe'
11    Call assemr(id, mxe, mxi, mxb)
12 EndDo

13 Sub assemr(id, rhse, rhsi, rhsb)
14 Do in = 1, nnped
15    node = Abs(icond(in, id))
16    iblock = iwherd(node, 1)
17    irel = iwherd(node, 2)
18    If (iblock.EQ.nblock)
19      Do i = 1, 5
20        rhsb(i+irel-1) = rhsb(i+irel-1)+rhse(i, in)
21      EndDo
22    Else
23      Do i = 1, 5
24        rhsi(i+irel-1) = rhsi(i+irel-1)+rhse(i, in)
25      EndDo
26    EndIf
27 EndDo
```

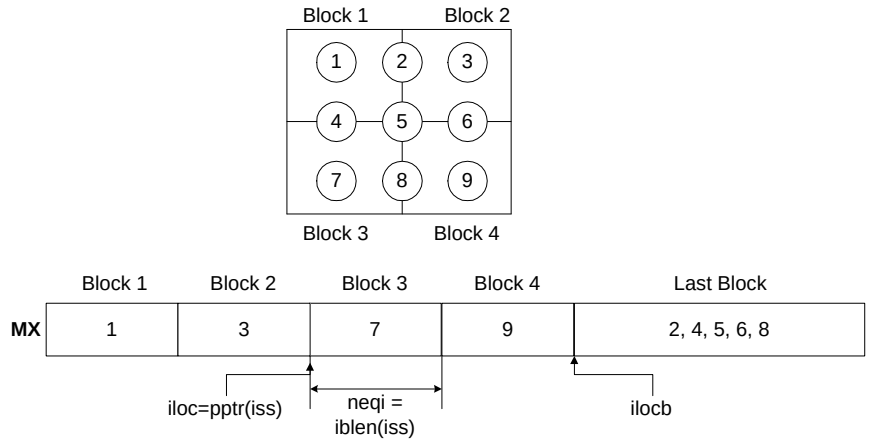


Fig. 57. Code extracted from loop MXMULT_do10 in benchmark application DYFESM (PERFECT suite) and schematic representation of the main data structure, array MX.

all the boundary nodes (shared by at least two physical blocks). Depending on block size, the size ratio of the last block over the whole array can range from 0 to 100%. In the test input size *MX* contains 4 blocks of 4 elements each, plus a last block of size 9.

The loop at line 1 iterates over the physical blocks. In each iteration *iss*, it calls subroutine *blckmx*. Within *blckmx*, arrays *MXI* and *MXB* are aliased to *MX* blocks *iss* and the last one respectively. Subroutine *blckmx* calls subroutine *assemr*, which takes an array of intermediate values *RHSE* and updates either *RHSI (MXI)* or *RHSB (MXB)* based on the values of indirection arrays *iwhered* and *icond*.

Each iteration produces two descriptors: WF_{iss}, which is exactly the *iss* block in *MX*, and RW_{iss}, which is a part of the last block. The operations at lines 20 and 24 are reductions. Moreover, in every iteration of the outermost loop the origin of WF references (line 7) dominates the RW operations at lines 20 and 24. It results that the dependences that cannot be removed by privatization and reduction parallelization consist of the intersection of WF and RW, as illustrated by Fig. 58. The USR representation in this figure differs slightly from the one used throughout this paper. It was generated automatically by Polaris using the *GraphViz dot* tool [187].

Fig. 59 shows a PDAG extracted from the dependence set in Fig. 58. This PDAG essentially tests whether the RW sections are empty for all *iss = 1, nss*. The PDAG consists only of comparisons, but it is only sufficient, and not necessary to prove independence. In this particular case, this PDAG will actually not produce meaningful information for any realistic input set. The dependence test for *MX* consists of a cascade. The PDAG made of comparisons is doubled by the evaluation of the whole USR and its comparison against the empty set.

Fig. 60 presents the USR that computes all the locations that have cross-iteration RW overlaps. It is a union across the iteration space of the intersection of the per-

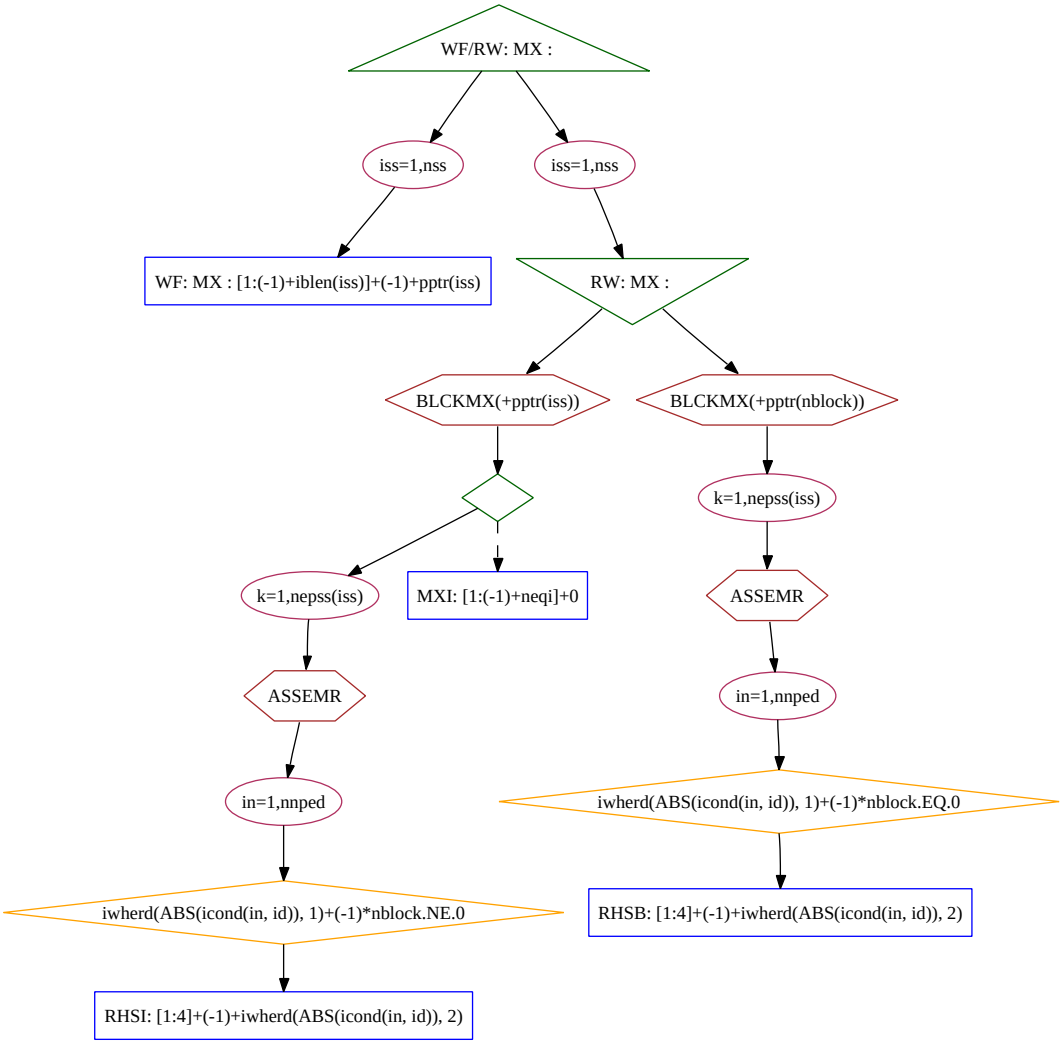


Fig. 58. Dependence set as a USR for array MX in loop MXMULT_do10. Only RW vs. WF dependences shown. Triangle = intersection, inverted triangle = union, ellipse = recurrence, empty diamond = difference (second term designated by dotted line), diamond with conditional = gate, hexagon = translation across subprogram boundary, and rectangle = list of LMADs.

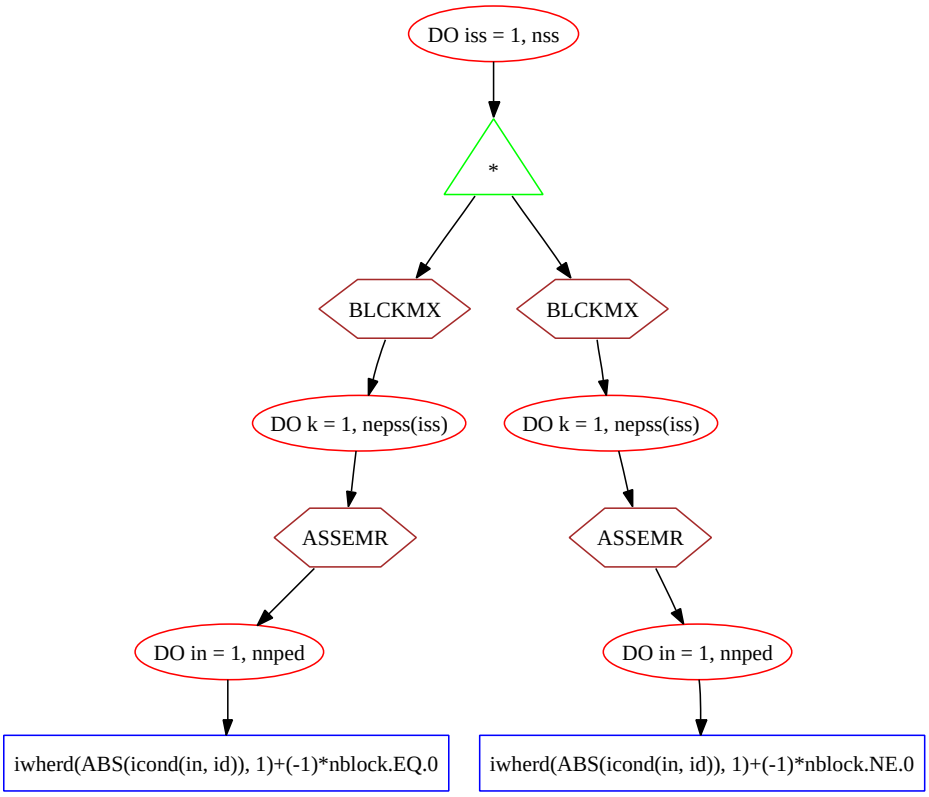


Fig. 59. Sufficient PDAG consisting of only simple expressions extracted from the dependence test on array MX in loop MXMULT_do10. Ellipse = logical AND across an iteration space, triangle = logical AND, hexagon = translation across subprogram boundaries, rectangle = conditional expression.

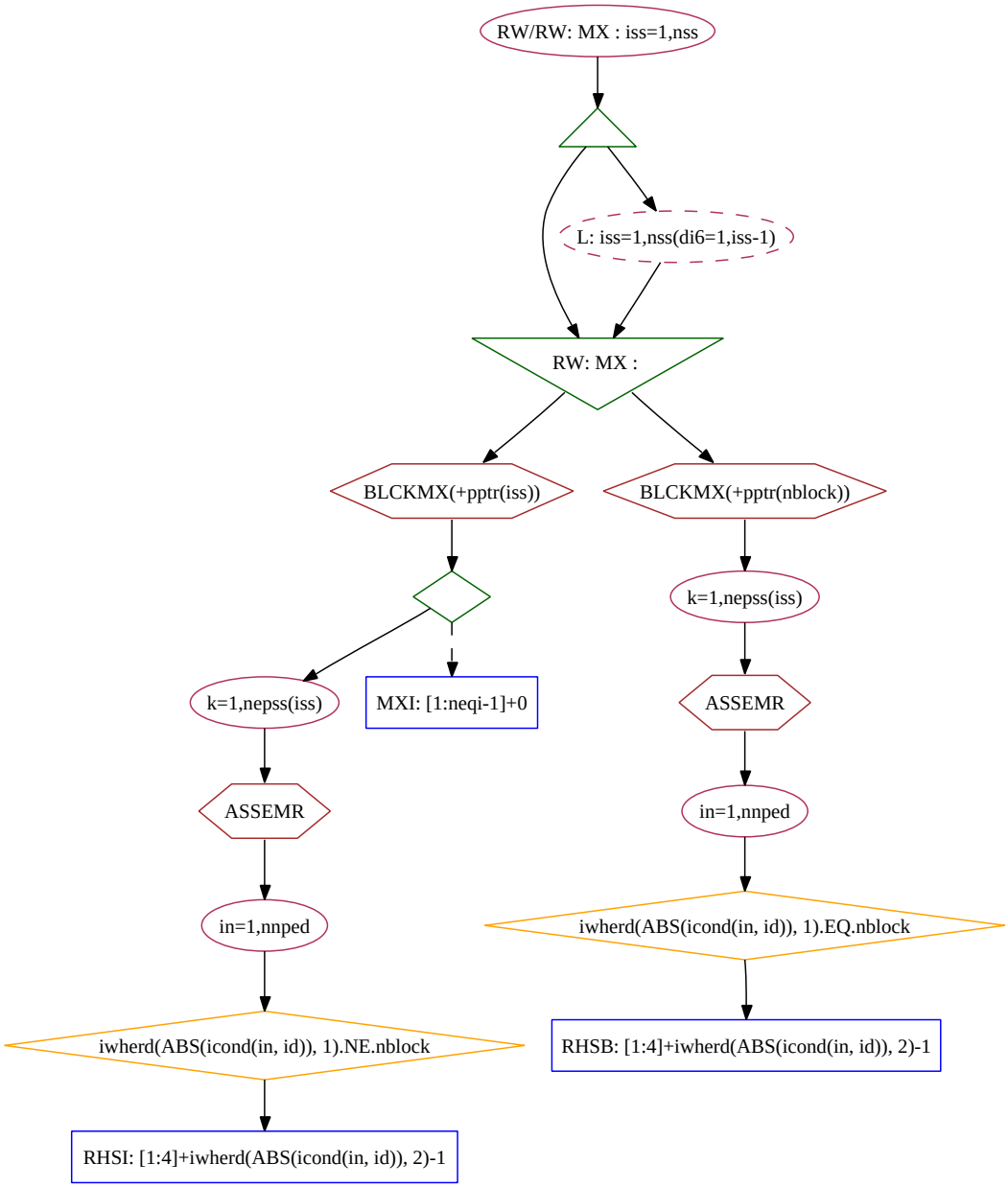
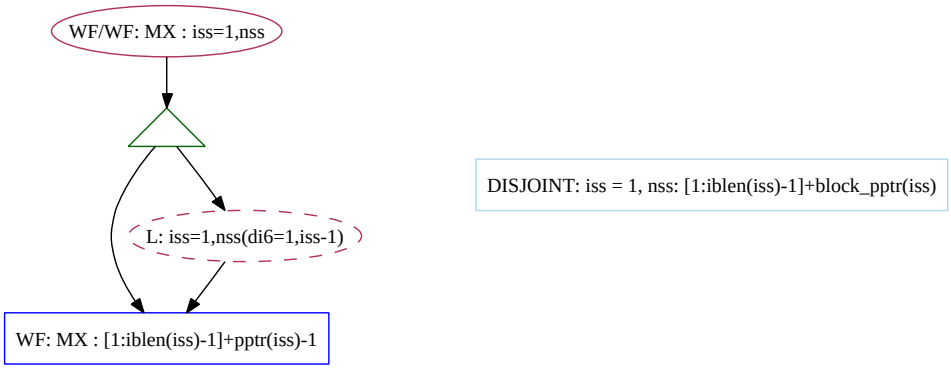


Fig. 60. Dependence set as a USR for array MX in loop MXMULT_do10. Only RW vs. RW dependences shown. They are the ones that can be removed by parallelizing the reduction operation. In addition to the symbol explanation given in Fig. 58, the dotted ellipse means a partial iteration space, in this case *1,2,...,iss-1*.

iteration RW_{iss} and the RW across all iterations before iss. The PDAG extracted by Hybrid Analysis essentially tests whether the RW sections are empty for all *iss = 1, nss*. The memoization mechanism in the PDAG extraction algorithm actually returns a reference counted clone of the PDAG shown in Fig. 59. This PDAG is doubled by a USR evaluation test.

In general, this test is crucial to the correctness of the parallelization. In particular cases (when there are no WF/WF overlaps), the loop could be executed as a full reduction using OpenMP primitives regardless whether there are or not RW/RW overhead. The work complexity of the reduction operation would be $O(n * p)$ (assuming n elements on p processors). Our solution is much better. When we knew that the operation is indeed fully independent, we avoid the final reduction operation by using shared rather than private storage for *MX* (switching at run time based on the overlap PDAG). Even though for this loop the test always fails, we can compute the *exact reduction operation footprint*, which in our case is the last block in *MX*. This reduces the work complexity from $O(n * p)$ to $O(\sqrt{n} * p)$, in the case where the number of blocks stays constant but the block size increases with n, the total number of elements.

Fig. 61 presents (a) the output dependence set and (b) its corresponding necessary and sufficient PDAG, which consists of a call to a run time library that can decide whether a set of intervals are mutually disjoint. Similarly to the dynamic reduction decision, it is important from a performance stand point to know whether there are any output dependence. The presence of dependences would trigger a possibly expensive computation of last value assignment locations for each iteration, which is similar in complexity to a reduction on operator MAX. Knowing that there are no dependences would avoid this possibly expensive operation and would eliminate the need to allocate private storage and to initialize it. In this particular case it



(a) (b)

Fig. 61. (a) Output dependence set descriptor as USR and (b) necessary and sufficient PDAG as call to a run time library.

turns out dynamically that there are no output dependences (the blocks are accessed within bounds respectively). However, here privatization still takes place because it is imposed by the reduction operation.

All the tests discussed above are generated as inspectors. They all rely on indirection arrays that are either read from an input file or computed at the beginning of the execution, before entering the main computation loop. The reuse rate is about 1000, which means each test was executed only once for 1000 dynamic instantiations of the loop. Table IX shows the dynamic characteristics of the tests. The simple PDAGs fail to prove RW and WF disjoint, but the more expensive test based on USR evaluation succeeds.

Fig. 62 presents the actual parallel (OpenMP) code generated by Polaris using Hybrid Analysis. Variables *mxmult_do10_is_indep*, *mxmult_do10_mx_nopriv*, *mxmult_do10_mx_nored* and r_43 are precomputed before the loop. The first three are the values returned by the inspector cascades that decide whether the loop can be

```
      IF (.NOT.mxmult_do10_is_indep) THEN
C       Save the current number of threads.
        mxmult_do10_np = omp_get_num_threads()
C       Run sequentially.
        CALL omp_set_num_threads(1)
      ENDIF


!$OMP PARALLEL
!$OMP+DEFAULT(SHARED)
!$OMP+PRIVATE(ISS,NEQI,PC1,PC0,MX0)
      IF (.NOT.mxmult_do10_mx_nored) THEN
        CALL usr_zero_out_r8(mx0, r_43)
      ENDIF
!$OMP DO
      DO iss = 1, nss, 1
C       Call to build COPY OUT descriptor r_57 for (mx, mx0).
        CALL rtlmadi_mxmult_do10_mx_p_red_copy_out_o_1(iss)
        neqi = iblen(iss)
        pc0 = pptr(iss)
        pc1 = pptr(nblock)
        IF (.NOT.mxmult_do10_mx_nopriv.OR..NOT.mxmult_do10_mx_nored) THEN
          CALL blckmx(iss, mx0(pc0), mx0(pc1), x, neqi, neqb)
        ELSE
          CALL blckmx(iss, mx(pc0), mx(pc1), x, neqi, neqb)
        ENDIF
    2   CONTINUE
        IF (.NOT.mxmult_do10_mx_nored.OR..NOT.mxmult_do10_mx_nopriv) THEN
          CALL usr_copy_out(mx0, mx, r_57, 8)
        ENDIF
      ENDDO
!$OMP END DO
      IF (.NOT.mxmult_do10_mx_nored) THEN
        CALL usr_reduce_add_r8(mx0, mx, r_43)
      ENDIF
!$OMP END PARALLEL


C     After per-symbol tests.
      IF (.NOT.mxmult_do10_is_indep) THEN
C       Switch back to parallel.
        CALL omp_set_num_threads(mxmult_do10_np)
      ENDIF
```

Fig. 62. Parallel code for loop MXMULT_do10. Variables *mxmult_do10_is_indep*, *mxmult_do10_mx_nopriv*, *mxmult_do10_mx_nored* and r_43 are precomputed before the loop. The call to *rtlmadi_..._copy_out_o_1* computes USR *r_57*, which is then used in the call to *usr_copy_out*.

Table IX. Run time tests actually executed to decide whether the dependence structure on array *MX* prohibits or allows parallelization. %S represents the time spent in the test as a percentage of the execution time of the loop.

Test	**Type**	**Accuracy**	**Success**	**% S**
Parallel/Sequential	Simple Expression	Sufficient	Fail	0.005
	USR Evaluation	Necessary&Sufficient	Pass	0.025
Indep. Update/Reduct.	Simple Expression	Sufficient	Fail	0.005
	USR Evaluation	Necessary&Sufficient	Fail	0.030
Indep. Write/Priv.	Interval Trees	Necessary&Sufficient	Pass	0.005

executed in parallel (indep ≡ there are no unremovable dependences), whether the reference pattern on *MX* is free of output dependences (nopriv ≡ there are no output dependences), and whether the RW reference pattern on *MX* is an independent update or a reduction (nored ≡ there are no cross-iteration RW dependences). RW dependences are treated separately only when the compile time analysis recognized the operation that produces them as a reduction.

1. Discussion

The code generation presented in this case study is not necessarily the most efficient possible. It is the result of a strategy that applies to the general case, and which takes into account a large array of possible run time scenarios.

In this particular case, the compiler could do better by identifying at compile time the originating sites of WF and RW respectively and create code versions that write directly in the shared MX assuming that the output dependence test will pass. This code version can be selected at run time after the output dependence test does pass. The advantage is that the copy-out phase is not necessary in this optimized version.

Unfortunately the number of optimized versions is an exponential function of the

number of dynamic decisions. An alternative would be to generate optimized code at run time, but that is beyond the scope of this paper.

CHAPTER VI

COMPILER DESIGN AND IMPLEMENTATION

We have implemented the Hybrid Analysis framework in Polaris [188], a source to source Fortran 77 research compiler. Our implementation consists of a generic bottom-up program traversal method that implements the abstract interpretation process to aggregate memory references. The atom of information is a triplet (RO, WF, RW), each of which is represented as a USR. When the analysis reaches a loop header, we perform dependence analysis, i.e., extract PDAGs from dependence questions, and generate parallel execution code and run time tests when necessary.

This section presents four important aspects of the implementation. First, we had to implement a set of prepasses to bring the input program to our program model. Second, we had to implement a symbolic analysis engine in order to push the static component of hybrid analysis as far as possible. Third, we will discuss the design rationale and complexity of the USR. Fourth, we will present the design and implementation of the PDAG, our boundary between static and dynamic analysis.

A. Making General Applications Fit our Program Model

The program model presented in Section III.1 helped us formalize the analysis process. However, most programs use language constructs that do not fit this model. We have implemented a series of *filters* that transform a given program such as a standard benchmark application into a program that fits our model.

First, we transform the program into an equivalent one in which the control dependence graph (CDG) is acyclic. Then we disambiguate aliased variables by using a single name for a whole alias class. Last, we perform a series of transformations that translate language constructs specific to Fortran 77 into simpler equivalent ones,

thus avoiding special cases.

1. Bringing Programs to Block Structured Form

A program in block structured form is a program that contains no jump instructions such as GOTO or RETURN. In such a program the only control structures are nested *If-Then-Else* blocks, *Do* and *While* loops. For simplification, we treat internally *While* loops as *Do* loops with an infinite number of iterations, each of which is guarded by the *While* condition. The *While* loops are restored before the code generation phase.

In all block structured form programs the CDG is a tree which mirrors the block hierarchy relations. Conversely, all programs for which the CDG is a tree can be rewritten without any jump statements by simply traversing the CDG. Jump statements can be treated as *No-ops* on the CDG because they are just control markers which on the CDG are represented explicitly by CDG edges.

When the CDG is not a tree but a DAG, it can be converted to a tree by splitting nodes with multiple parents. Although in theory this could lead to an exponential increase in the number of statements, in practice the increase was below 40% across a large class of benchmark programs. Newer, better written codes use fewer GOTOs and thus require almost no node splitting. Our proposed analysis can be performed on programs for which the CDG is a DAG, but we have preferred to simplify them to trees (thus block structured programs) for simplicity.

Trivial CDG cycles (self loops) can be tolerated by our analysis as long as they are marked as loops. This way, the abstract interpretation process will process them as loops and thus produce an accurate view of the memory reference pattern.

Nontrivial CDG cycles do not fit our representation. In more common terms they correspond to loops with premature exits. Our approach is to modify the CDG

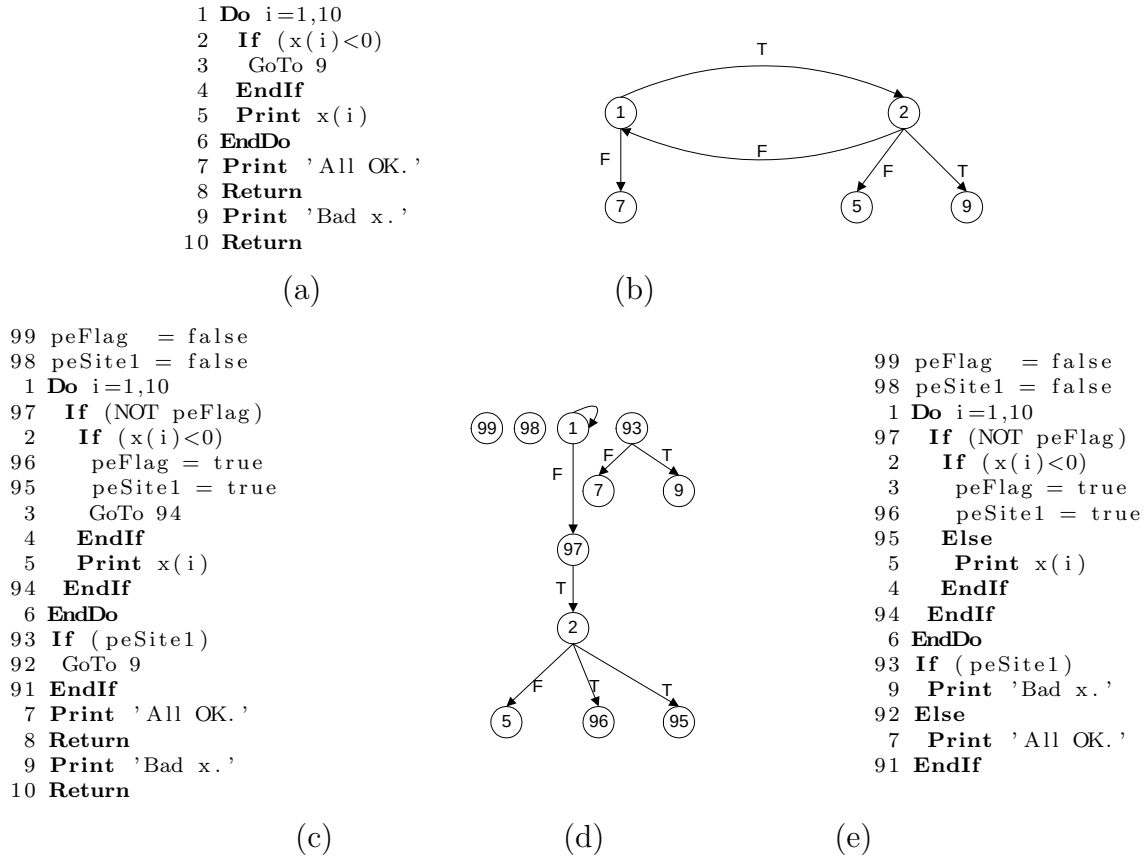


Fig. 63. Code Restructuring. (a) Original code and (b) corresponding CDG with a nontrivial cycle. (c) After insertion of control variables *peFlag* and *peSite1*. (d) Corresponding CDG with only a trivial cycle and (e) code generated from the CDG (without any jump statements such as GOTO and RETURN).

so that it becomes a DAG with self loops. This modification is illustrated in Fig. 63. The loop at line 1 in Fig. 63(a) contains a premature exit controlled by the conditional at line 2. The conditional and the loop header form a cycle in the CDG. In order to remove this cycle, we introduce a control variable *peFlag* (Fig. 63(c)), which is set to true when the premature exit would have happened. Instead of jumping outside the loop, we now jump to the end of the current iteration. The whole iteration is guarded by the control variable. Additionally, we keep another control variable *peSite1* which stores the exact location of the premature exit in case there are multiple such sites. The stub right after the loop jumps to the correct target when the loop is exited prematurely. Although this transformed program still contains GOTOs, its CDG ,Fig. 63(d), does not contain cycles anymore except for the self loop at node 1. We then rewrite this CDG as the program in Fig. 63(e).

In general, our restructuring process relies on introducing control variables for every node in nontrivial strongly connected components in the CDG, except for the node that has a parent outside the connected component. At this point, our algorithm cannot deal with strongly connected components that can be entered from more than one point.

2. Alias Disambiguation

Aliases are different names that reference the same memory location. Recognizing aliases is crucial especially for analyses where the order of references is important, such as privatization. The Fortran 77 standard forbids the programmer from creating aliases within a subprogram by associating two formal arguments or global variables, when any of the objects is written to within that subprogram (or any called subprogram). While this helps greatly with intraprocedural analysis, there are other classes of alias that are important to recognize, and where possible eliminate, when

```
1 Sub test
2 Common /c/ x(100), z(10), w(90)
3 x(11) = 1
4 Call init(a)
...
5 Sub init(v)
6 Common /c/ y(10,10), t(5)
7 If (y(1, 2)=1)
8   v = 0
9 EndIf
```

(a)

```
1 Sub test
2 Common /c/ x_y(100), z_t1(5), z_t2(5), w(90)
3 x_y(11) = 1
4 Call init(a)
...
5 Sub init(v)
6 Common /c/ x_y(10,10), z_t1(5), z_t2(5), w(90)
7 If (x_y(1, 2)=1)
8   v = 0
9 EndIf
```

(b)

Fig. 64. Unification of COMMON structures disambiguates aliases. (a) Original code. (b) After common unification.

performing interprocedural analysis.

a. Commons

In Fortran 77 global variables are managed using COMMONs. They are globally visible names that contain a list of objects (scalars and arrays) of sizes known at compile time. Unfortunately these lists do not have to match across different subprograms. Consider the example in Fig. 64(a). The same common block */c/* has different variable lists in the two subprograms. When performing interprocedural analysis it is crucial to map name *test:/c/x* into *init:/c/y*. In order to make this translation easier, we implemented a unification pass that renames the common variable lists (and their associated uses) so that they match across subroutines. There are cases where this renaming schemes cannot work, such as the case when the memory used by a floating point variable in a subprogram is used for two integers in another. In that case, our analysis considers them all aliased. We do not perform array reshaping when unifying in order to preserve as much information about array bounds as possible. This information is valuable for extracting symbolic range information because Fortran 77 forbids out of bounds array accesses.

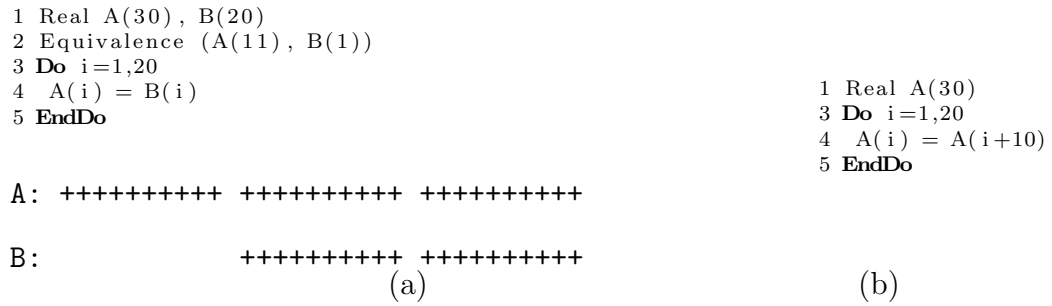


Fig. 65. Unification of EQUIVALENCE-ed names disambiguates aliases. (a) Original code. (b) After equivalence unification.

b. Equivalence

In the example in Fig. 65(a), the loop at line 3 seems parallelizable. However, it actually contains cross iteration dependences due to the fact that A and B are aliased. When the alias is disambiguated in Fig. 65(a), the dependences are easy to point out. We perform an automated renaming algorithm for all EQUIVALENCE-ed names. This routine attempts to not reshape arrays wherever possible. The names that are still aliased after this filter are marked as such and considered as the same object by all subsequent analysis passes.

c. Type Mismatches across Subprograms

In legacy codes it is relatively common to hand-optimize codes for a small memory usage. In Fortran 77, this is usually done by identifying call graph slices that do not have any data dependences. Subprograms in the same slice usually communicate with each other through commons, but some such variables are temporary with respect to the whole slice. In this case, these variables global to a slice are overlayed in memory with some other variables global to another slice. This overlay is done by giving the same name to two different commons. Our common restructuring mechanism may

```
 1 Sub compute(A, V, W)
 2 Dimension A(10,10)
 3 Dimension V(*), W(1)
 4 Do i=1,10
 5   Do j=1,10
 6     V(j) = ...
 7     W(j) = ...
 8   EndDo
 9   A(f(i),i) = g(V, W)
10 EndDo
```

Fig. 66. Array bounds issues.

fail in these cases. However, in many such cases the interprocedural analysis across the two slices is not relevant since they do not share an actual data flow.

When there is an overlap between a *Real* and an *Integer* variables in two different subprograms, due to either commons or argument matching, the Fortran 77 standard forbids any data flow to occur. This means that the data flow paths are guaranteed to be different. We have implemented a simple pass that recognizes type mismatches, traces them to the original storage, and splits the storage by renaming based on the number of actual data flows. This transformation is the reverse of the manual storage-saving optimization and can be used just for analysis purposes (the code can be left in its original hand-optimized form).

3. Language and Programming Style Issues

There are several Fortran 77 issues and related programming patterns that we want to address for clarity. They are not fundamental to the techniques proposed in this document but are important for practical reasons.

a. Array Bound Declarations

Fortran 77 forbids out of bounds array accesses. In the example in Fig. 66, this means, assuming the code follows the language standard, that the loop at line 4 cannot carry any dependences on *A* since the access to *A(f(i), i)* is always within *A(1:10,i)*.

Most compilers rely on language standards although there are programming practices that go against them. For instance, in order to parallelize the loop at line 4 in Fig. 66, the compiler has to privatize arrays *V* and *W*. Array *V* is declared with assumed size. The backend *OpenMP* compiler that we use currently reports an error if a *Private* directive is attached to an assumed size variable. We designed a pass that figures out a safe size for assumed size variables based on both the originating storage and the USRs that describe its access pattern within the loop (sometimes not the whole array must be privatized).

Unfortunately array *W* is declared as having a single element but is used as if it had 10 elements. This is a common practice against the language standard. When we generate a *Private* directive for *W*, the backend compiler privatizes only the first element, which results in erroneus results. We addressed this common mistake by conservatively replacing declarations such as *W(1)* with *W(*)*.

b. Multiple Subprogram Entries

Fortran 77 allows a subprogram to be entered at different points. While offering an incremental amount of flexibility, this causes an unnatural difficulty in representing relations between subprograms in the call graph. We decided to transform the programs into equivalent ones such that every subprogram has exactly a single entry point, situated at the first executable statement. While this does increase the code

```
1 Program main
2 Call compute(...)
...
3 Do i=1,100
4   Call compute(...)
5 EndDo
```

```
 7 Sub compute(...)
 8 Data isInitialized /false/
 9 If (NOT isInitialized)
 0   Call initialize(...)
11   isInitialized = true
12 EndIf
 ...
```

Fig. 67. Lazy initialization pattern.

size, it is sufficiently rare not to make a visible difference in performance overall.

c. Data Statements

DATA statements are the Fortran 77 equivalent of *static* local variables in C and C++. They are guaranteed to be initialized only upon the first dynamic entry to the subprogram where they are declared. Because they create a special case in the data flow structure of the program, we decided to replace them with global variables that are initialized using assignment statements (and loops) at the beginning of the main subprogram.

d. Lazy Initialization Code

Fig. 67 shows a common pattern used especially in conjunction with DATA statements. Subprogram *compute* is offered as a library function. However, the library does not offer an interface for subroutine *initialize* so this subroutine must be called on demand upon first entry to *compute*. This pattern is not particular to Fortran 77, but is used in C programs as well.

Unfortunately, while making it easy to design modular libraries, this makes it hard for the compiler to analyze loops such as the one at line 3 in Fig. 67. When subprogram *initialize* has global side effects, these effects seem to possibly happen at every iteration. These apparent side effects may prevent further optimization such as

Table X. Attribute grammar for generating Fortran code for USRs.

Syntax Production	**Attribute Grammar Production**
$D \rightarrow LMAD\ list$	`CALL set(D.val, LMAD list)`
$D \rightarrow D_1 - D_2$	`CALL subtract (D.val, D1.val, D2.val)`
$D \rightarrow D_1 \cap D_2$	`CALL intersect(D.val, D1.val, D2.val)`
$D \rightarrow D_1 \cup D_2$	`CALL unite (D.val, D1.val, D2.val)`

$D \rightarrow D_1 \# Gate$	`IF (Gate.predicate)` THEN
	`  CALL set (D.val, D1.val)`
	`ELSE`
	`  CALL set (D.val, ∅)`
	`ENDIF`

$D \rightarrow D_1 \bowtie CallSite$	`CALL shift(CallSite.offset, D.val, D1.val)`

$D \rightarrow D_1 \otimes (i = 1, N)$	`CALL set (D.val, ∅)`
	`DO i=1,N`
	`  CALL unite (D.val, D.val, D1.val)`
	`ENDDO`

parallelization.

Our solution to this common problem is to recognize these patterns by finding conditions that control assignment statements that change the value of their predicate. When such structures are invoked at several sites, all of which dominated by a single one, then the initialization phase can only occur at the common dominator site. We then split the original suprogram in two (*initialization* and *work*) and invoke them both at the common dominator. At all other sites, we invoke only the *work* component.

B. USR Design and Implementation

When a USR-based optimization decision cannot be reached at compile time, Hybrid Analysis generates run time code to make the decision at run time. In most cases, the run time test will be a PDAG. In other cases, the run time test will consist of evaluating the USR. In both cases, it is crucial that the USR be simplified as much

as possible at compile time, so that the associated run time evaluation phase has reduced overhead.

The code generation for USR evaluation is a translation to Fortran of the language presented in Fig. 16. In order to evaluate $[1 : n] \cap [101 : 100 + n]$ we first obtain the value of n (assume it is 10). Then we perform the required operation $[1 : 10] \cap [101 : 110] = \emptyset$. The translation is based on an attribute grammar [8] described in Table X. The only attribute we use is the run-time value of the USR as a list of LMADs made of known integer values. Gates translate to *If* statements, recurrences to *Do* loops. Set operations and *CallSite* nodes translate into calls to a run-time support library that operates on lists of LMADs with known integer values. The generated statements are inserted in the code at the first point where all the values they use are defined. In case the evaluation method is an inspector, we clone the program slice [189] that computes the values referenced by the USR.

1. USR Optimization

This section focuses on methods to reduce the complexity of USRs. This reduction in complexity will translate into a reduction in run time overhead for the evaluation of USRs and associated PDAGs. Optimization is performed both during compilation and when evaluating USRs at run time. A USR can be viewed at the same time as a set, as an algebraic expression or as a parse tree, which means we can apply known simplification techniques for these three types of representation.

The static optimization phase is based on symbolic analysis. It either simplifies or restructures USRs so that their predicted evaluation time decreases. Such USR transformations are based on dataflow analysis (such as loop invariant hoisting), on control dependence analysis (such as `AND`-ing mutually exclusive predicates), based on set identities (such as $(A - B) - A = \emptyset$), and based on lattice identities (such as

Table XI. USR evaluation cost model.

USR D	**Cost** $c(D)$
$LMAD\ list$	0
$D_1 - D_2$	$c(D_1) + c(D_2) + 1$
$D_1 \cap D_2$	$c(D_1) + c(D_2) + 1$
$D_1 \cup D_2$	$c(D_1) + c(D_2) + 1$
$D_1 \# Gate$	$c(D_1) + 1$
$D_1 \bowtie CallSite$	$c(D_1) + 1$
$D_1 \otimes (i = 1, N)$	$N * c(D_1) + N - 1$

$A - \top = \emptyset$). Loop invariant USR hoisting is similar to the inspector re-use technique in [141]. The remainder of this section presents the static methods in detail.

Other optimizations are performed at the run-time library level, e.g., contiguous aggregation, coalescing and interleaving. These optimizations were introduced in [13] as compile time optimizations.

a. Optimization Based on Minimal Evaluation Cost Form

Throughout this section we will assume that all data needed to evaluate the USR are already precomputed (no slicing is necessary). Let us associate a cost with every USR as shown in Table XI.

The USR can be viewed as a tree. An evaluation of the USR can be viewed as a bottom-up traversal of its tree. The cost of the evaluation can be estimated using our model.

Let us observe that the same USR can be represented in several equivalent forms. For instance: $\cup_{j=1}^{N}(A \cap B_j) = A \bigcap \cup_{j=1}^{N} B_j$. In the first case, the evaluation cost is $N*1+N-1 = 2*N-1$, while in the second it is $N-1+1 = N$. In general, an USR can be brought to equivalent forms through transformations such as distribution. We define the *minimal (evaluation) cost form* of an USR the form that has the minimal

cost across all possible equivalent forms.

Intuitively, the evaluation cost decreases when we hoist above recurrence nodes all nodes that are invariant to the respective recurrences.

b. Partial Invariants

The observation in the previous section relies on the fact that we identified A as invariant within the recurrence $j = 1, N$. In general, an operand of a recurrence may contain invariants that cannot be hoisted trivially above the recurrence operator. For instance, $\cup_{j=1}^{N}(A_j - B - C_j) \neq \cup_{j=1}^{N} A_j - B - \cup_{j=1}^{N} C_j$. The right formula is $(\cup_{j=1}^{N}(A_j - C_j)) - B$.

This section presents an algorithmic O(N) method for extracting all invariants with respect to a recurrence from an USR with N operands. We consider that we already know which operands (as LMAD lists) are invariant.

The extraction is based on the following equations:

$$\cup_{j=1}^{N}(A \cup B_j) = \quad A \bigcup \cup_{j=1}^{N} B_j \tag{6.1}$$

$$\cup_{j=1}^{N}(A \cap B_j) = \quad A \bigcap \cup_{j=1}^{N} B_j \tag{6.2}$$

$$\cup_{j=1}^{N}(A_j - B) = \quad (\cup_{j=1}^{N} A_j) - B \tag{6.3}$$

$$\cup_{j=1}^{N}(A - B_j) = \quad A - \cap_{j=1}^{N} B_j \tag{6.4}$$

In order to bring the operands of the recurrence to one of the forms in the equations above we perform transformations such as the following (similar equations exist for all combinations of set operations).

$$(A_i \cup B) \cup C_i = \qquad B \cup (A_i \cup C_i) \tag{6.5}$$

$$(A_i - B) \cap C_i = \qquad (A_i \cap C_i) - B \tag{6.6}$$

$$(A_i \cup B) - C_i = \quad (A_i - C_i) \cup (B - C_i) \tag{6.7}$$

```
1  Do j=1,n
2   Do k=1,m
3    tmp(k) = ...
4   EndDo
5   Do k=1,m
6    IF (c(k,j)<0)
7      ... = tmp(k)
8    EndIf
9   EndDo
```

Fig. 68. Access pattern that can be approximated using LMADs.

Based on our simple model, none of these transformations increases the cost. Some of them (such as 3) seem to be non-profitable since the cost stays the same. We still prefer to hoist invariants in order to take advantage of other optimization opportunities such the ones presented over the following sections.

c. Approximation with LMAD Lists

Consider the code in Fig. 68. The *read* (2) access pattern on array `tmp` has a complex shape because it is controlled by an array of conditions `c(:,:)`. However, regardless of its shape, the *read* memory accesses are completely overlapped by the previous *writes* (1). This results in the array *tmp* being privatizable in the context of parallelizing the outer loop.

We cannot make this inference using USRs directly. The details needed to precisely represent the shape of the memory access make it hard to compare the *read* and the *write* descriptors.

Our solution is to use two additional descriptors with every USR. They represent an *Overestimate* and an *Underestimate* of the USR using *lists of LMADs*.

The rules for approximating an USR with a list of LMADs are shown in Table XII. Using these rules we compute estimates for arbitrarily complex USRs. The approximations must be conservative, i.e. an overestimate must be larger than its

Table XII. USR approximation using LMAD lists.

USR *D*	***U(D)***	***O(D)***
$LMAD\ list$	$LMAD\ list$	$LMAD\ list$
$D_1 - D_2$	$U(D_1) - O(D_2)$, or $\emptyset$[1]	$O(D_1) - U(D_2)$, or $O(D_1)$
$D_1 \cap D_2$	$U(D_1) \cap U(D_2)$, or $\emptyset$	$O(D_1)$ if $size(D_1) < size(D_2)$, else $O(D_2)$
$D_1 \cup D_2$	$U(D_1) \cup U(D_2)$	$O(D_1) \cup O(D_2)$
$D_1 \# Gate$	$\emptyset$	$O(D_1)$
$D_1 \bowtie CallSite$	$U(D_1) \bowtie CallSite$	$O(D_1) \bowtie CallSite$
$D_1 \otimes (i = 1, N)$	$O(D_1) \otimes (i = 1, N)$, or $\top$	$U(D_1) \otimes (i = 1, N)$, or $D_1\|_{i=1}$, or $\emptyset$

corresponding memory reference descriptor. On the other hand, very loose estimates do not lead to any conclusions. The rules shown in Table XII reflect our effort of maintaining the estimates as close to the real descriptors as possible.

These estimations are used to simplify the USRs based on the fact that $O(D_1) \subseteq U(D_2) \Rightarrow D_1 \subseteq D_2$. This is expressed in terms of USR operations using the following equations.

$$O(D_1) \subseteq U(D_2) \Rightarrow \quad D_1 - D_2 = \emptyset \tag{6.8}$$

$$O(D_1) \subseteq U(D_2) \Rightarrow \quad D_1 \cap D_2 = D_1 \tag{6.9}$$

$$O(D_1) \subseteq U(D_2) \Rightarrow \quad D_1 \cup D_2 = D_2 \tag{6.10}$$

d. Language Specific Optimization

As described in the previous section, every USR has two estimates (O and U). In some cases they cannot be obtained or maintained and they are conservatively assumed to be $\top$ (top), respectively $\bot$ (bottom). While for $\bot$ there is a definite value ($\emptyset$), such a value does not exist for $\top$. There are two provisions in the `Fortran 77`

Standard that help obtaining values for $\top$ within certain contexts.

When the dimension of an array is fully specified (all dimension bounds are expressions different from the asterisk), $\top$ is conservatively set to the size of the array (Fortran 77 Standard, section 5.2.3, page 5-3).

When an array is passed as an actual argument to a subprogram, it is subject to the *Restrictions on Association of Entities* (Fortran 77 Standard, section 15.9.3.6, page 15-20). The standard states that if two dummy entities in the called subprogram are associated during the execution of a subprogram call, than neither dummy entity can be defined during that particular call. We use this provision of the standard to limit the $\top$ value of USRs that represent *write* memory descriptors that can be associated. Consider this statement: `CALL sub(A(1), A(10))`. We can infer that the descriptor that corresponds to the *write* pattern corresponding to the first argument is limited to the memory region `A(1:9)`. Otherwise the standard would be violated, since the part of `A` that extends past `A(9)` is associated with the second argument.

Redundancy Elimination and Trivial Case Detection

There are two sources for both redundancy and triviality. First, the predicates in USR gates may be incompatible or they may imply each other. More formally,

$$(D_1 \# G) \cap (D_2 \# \bar{G}) = \emptyset \tag{6.11}$$

$$(D_1 \# G) - (D_2 \# \bar{G}) = D_1 \tag{6.12}$$

$$(D \# G_1) \cap (D \# G_2) = D \# G_1, \forall G_1 \Rightarrow G_2 \tag{6.13}$$

$$(D \# G_1) - (D \# G_2) = \emptyset, \forall G_1 \Rightarrow G_2 \tag{6.14}$$

Second, set lattice properties like idempotency, complementariness, absorption

lead to simpler descriptors. Formally,

$$A \cap A = A \tag{6.15}$$

$$A \cap (B - A) = A \cap B \cap \bar{A} = \emptyset \tag{6.16}$$

$$(A \cup B) \cap A = A \tag{6.17}$$

The set lattice of the USRs representing the memory access for a symbol always has a bottom value ($\perp = \emptyset$). As we showed in the previous section, in certain cases a top value ($\top$) can be computed. These values can be used for simplifications as follows.

$$A - \top = \emptyset,\ A - \perp = A \tag{6.18}$$

$$A \cap \top = A,\ A \cap \perp = \perp \tag{6.19}$$

$$A \cup \top = \top,\ A \cup \perp = A \tag{6.20}$$

C. PDAG Design and Implementation

PDAGs are implemented as directed acyclic graphs that can overlap with each other in memory (using reference counted pointers). This allows for an exponential number of logical nodes to be stored in linear space. This feature is crucial for the scalability of techniques such as nested parallelism detection, where many of the analysis domains are pairwise subsets.

In general, the detection of parallelism at an outer loop is not directly related to the detection of parallelism at a contained, inner loop. However, a subset of the problems is the same – and this commonality is exploited naturally by the way PDAGs

```
1 Read n
2 Do i=1,n
3   A(i+100) = A(i)
5   B(i+200) = B(i+300)
6 EndDo
```

(a)

```
1 Read n
2 Do i=1,n
3   Do j=1,n
... // RO_{i,j}, WF_{i,j}, RW_{i,j}
8   EndDo
9 EndDo
```

(b)

Fig. 69. (a) Similar dependence tests for arrays A and B can be stored using a single PDAG. (b) The per-iteration MCA partition descriptors will appear in sub-PDAGs in dependence questions at both loop levels.

are built. We use memoization aggressively when building PDAGs. In the example in Fig. 69(a), although the two arrays are referenced at different addresses, the run time dependence test is the same, $n \leq 100$. When building the PDAG for the dependence test for A, we cache the result indexed by the input expression $n \leq 100$. The second time we want to build a PDAG, for the dependence test for B, we do not actually build it, but rather use the cached PDAG. In addition to saving memory and time during compilation, this actually reduces the run time overhead.

In the example in Fig. 69(a), let us assume that we have already performed MCA for the body of the inner loop. The same resulting USR descriptors will be used to build PDAGs for dependence tests at both loop levels. Parts of the results of their comparisons may be common between the tests at the two loop levels. The common parts are detected automatically by the memoization process during PDAG construction. It does not matter whether the loops are nested cleanly. They could actually be in different subprograms or/and separated by an arbitrary number of statements. The analysis using USRs and PDAGs is thus scalable and very robust.

D. Complexity of Hybrid Analysis

We now show that the computational effort of Hybrid Analysis is quite manageable both at compile-time as well as at run-time thus yielding a viable solution for optimization based on memory reference analysis such as thread level automatic parallelization.

In order to understand the structure of space and time complexity of Hybrid Analysis, we will separate its compile time phase into two parts. First, the Memory Classification Analysis creates USRs but does not formulate optimization questions, thus it does not extract PDAGs. Other analysis techniques based on aggregation of USRs across the program will have similar complexity. Second, we have the extraction of PDAGs from USR identities as needed by automatic parallelization.

1. Compile Time Complexity

a. Memory Classification Analysis

We will show that the memory and time used at compile-time is $O(\sum_{sym} \textit{StaticAccessCount(sym)})$ if no USR simplification is performed. Below, we give the time complexity of our analysis of a single symbol assuming that the symbolic forward propagation, range dictionary, and interprocedural SSA passes have already run. Throughout this section, we assume symbolic comparisons of algebraic expressions to take constant time.

The overall memory budget is composed of the storage needed to keep the USR internal nodes and the memory needed to store the primary representation objects (the LMADs). We allow the parse trees for different USRs to overlap in memory. This way the number of additional internal nodes needed to represent the result of any USR operation is constant. Every summary set update for successive blocks or statements

can create 15 more USR nodes, every recurrence 9 more, every conditional can create 9 nodes, and every routine call 3 more (the number of nodes can be counted as the maximum number of operators on the right hand side in Figs. 18, 20, 22, and 25. The number of USR nodes is upper bounded by $(15 * S + 9 * L + 9 * I + 3 * C)$, when there are S statements, L loops, I IF statements, C call sites that may have effect on the access pattern. Storage for the primary representation (LMADs) may increase exponentially (worst case) with the number of static memory references. We avoid this by limiting the number of LMADs that we store in an LMAD list to a constant (50, for now). In our experiments, the limit was never reached because most operations on LMADs either produce an LMAD (not increasing the size), or are not exact, in which case the result is represented as an USR. The size of an LMAD is proportional to the number of dimensions of the access pattern, which in practice is < 4. Thus, total memory usage for computing the access pattern on an array `A` is upper bounded by $(12 * S + 9 * L + 9 * I + 3 * C)$*sizeof(USR)+$S * 50 * 4$*sizeof(1D-LMAD), i.e., it is linear in the number of program statements that may have effect on the access pattern.

Some of the optimizing transformations we apply to USRs require bottom-up traversals of the associated parse trees with constant-time pattern matching performed at every node. Because the size of USRs grows linearly, and there are a linear number of aggregations, the time complexity is upper bounded by

$$O(\sum_{sym} StaticAccessCount(sym)^2)$$

The quadratic behavior is not reached in practice because the optimizing transformations performed reduce the sizes of USRs as they are aggregated.

Table XIII. Compile-time analysis statistics in seconds for both MCA and PDAG extraction for parallelization. Column 4 and 5 show the total number of USR and PDAG nodes created (operator or leaves).

Code	**Lines**	**Time**	**USRs**	**PDAGs**
ADM	5,791	455	35,249	10,456
ARC2D	3,099	102	13,178	22
BDNA	4,919	36	11,181	156
DYFESM	3,903	38	6,841	756
FLO52	2,508	120	8,371	0
MDG	1,237	15	8,085	744
OCEAN	2,738	122	14,664	208
SPEC77	4,582	303	75,032	4,733
TRACK	2,523	245	27,790	2,931
TRFD	656	120	1,684	139
APPLU	3,980	56	13,212	34
APSI	7,488	399	36,593	10,800
MGRID	489	108	2,089	0
SWIM	435	7	1,785	0
WUPWISE	2,184	45	4,710	60
HYDRO2D	4,461	33	5,911	11
MATRIX300	439	3	1,458	0
MDLJDP2	4,172	18	6,928	444
NASA7	1,204	48	8,545	547
ORA	373	7	2,562	0
SWM256	487	8	1,520	0
TOMCATV	194	5	1,056	32

b. PDAG Extraction

The complexity of the syntax-directed translation could be exponential in the worst case, due to productions such as: $A \cap (B \cup C) = \emptyset \mapsto (A \cap B = \emptyset) \wedge (A \cap C = \emptyset)$. However, this tendency is avoided through aggressive memoization of solutions to common subproblems. The extraction of approximative tests and the pattern matching algorithms have complexities at most linear with the size of the given USR.

Table XIII presents compilation statistics for a set of 22 applications from bench-

mark suites PERFECT and SPEC. The number of USR and PDAG nodes is relatively small. On the average, a USR node occupies about 3 KB, while a PDAG node occupies about 24 bytes. There is no precise correlation with the number of lines of code because applications differ greatly in the static number of memory references. In some cases the compilation times are long because of failed attempts to simplify USRs, which may result in up to quadratic complexity [3].

2. Run Time Complexity

a. USR Evaluation

The additional memory required at run-time is for the lists of LMADs used at run-time to evaluate USRs. Our USR evaluation scheme is similar to a register-based evaluation scheme for an arithmetic expression. Instead of machine registers we use lists of LMADs. In the worst case the number of our 'registers' is linear with the number of memory reference statements in the original code. Also, this number is known at compile-time and they can be statically allocated. If the access pattern is found linear at compile-time (as in direct indexing), then the size of a 'register' is input data invariant (the size of an LMAD is proportional to the number of linear dimensions in the space it represents). If the access pattern is found linear at run-time even though it did not seem linear at compile-time (as in subscripted subscripts that take linear values at run-time), then the size of the 'register' will still be constant. The size of the 'register' increases only when a recurrence has a non-linear access pattern that cannot be aggregated using LMADs even at run-time. In the worst case, the size of a 'register' can be the same as the size of the data set tested for dependences. In that case, we fall back to shadow array based analysis such as the LRPD test. Although the complexity of LRPD is asymptotically the same as the

Table XIV. Run time test dynamic overhead reduction through HDA: ratio between the number of actual memory references and the number of PDAG operations performed at run time. Only the applications with run time tests are shown.

App.	ADM	ARC2D	DYFESM	MDG	OCEAN
Ratio	$1.8 * 10^5$	$1.2 * 10^7$	$1.5 * 10^4$	$6.7 * 10^0$	$2.1 * 10^4$
App.	SPEC77	TRACK	TRFD	APSI	NASA7
Ratio	$1.0 * 10^0$	$1.0 * 10^0$	$5.6 * 10^4$	$1.6 * 10^7$	$3.0 * 10^6$

dynamic memory reference count, its individual operations are very light, so the overall overhead can be tolerated in many cases.

The *time complexity* is (worst case) that of the LRPD test, i.e., proportional to either the number of distinct memory references or number of references for dense and sparse access patterns, respectively. However, in practice, the actual complexity is orders of magnitude smaller, depending on the degree of reference aggregation that HA manages to extract. Many times we need only constant time to evaluate a small number of conditions. Even with USRs that take non-constant time to evaluate, our framework can easily take advantage of value reuse (a.k.a. schedule reuse) through aggressive hoisting.

b. PDAG Evaluation

PDAGs are almost always faster to evaluate than USRs. In general they are used when we do not need to evaluate a USR but rather answer a question about an identity involving USRs. These questions are easier than evaluating the USRs themselves and verifying the identity at run time. In some case, we do have to fall back to either evaluating USRs or performing the LRPD test. Table XIV shows the reduction in the number of run time operations as compared to the dynamic memory

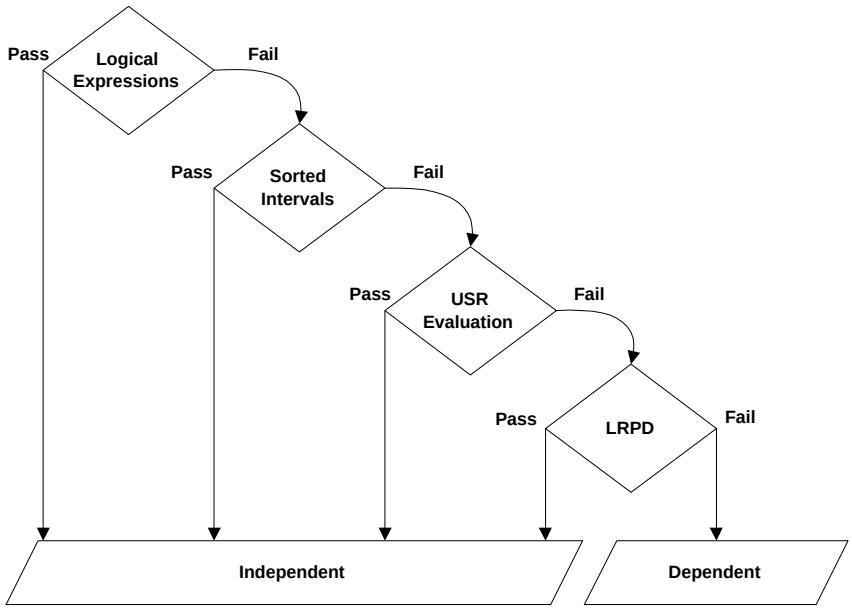


Fig. 70. Cascade of sufficient run time tests in increasing order of complexity.

reference count through USR aggregation and PDAG synthesis.

PDAGs contain four types of run time operations: (1) evaluation of elementary conditional expressions, (2) sorted checks, (3) actual evaluation of USRs and comparison to the empty set and (4) reference-by-reference LRPD. We extract, for each dependence equation, a *cascade of tests* (Fig. 70). ordered by their estimated complexity. They range from $O(1)$ tests as the one in Fig. 1 to $O(n)$ dynamic reference instrumentation as is the case in Fig. 30. Evaluating USRs at run time generally consists of fewer (but more complex) operations than the reference-by-reference LRPD [129]. In some cases they may either degenerate into inefficient enumerations or take conservative decisions that can lead to false negatives. The LRPD has overhead proportional to the dynamic reference count, but is optimal for cases where aggregation and equation inversion are not possible (Fig. 30), and is always applicable, precise, and has a more predictable complexity.

All the tests can be run in either inspector/executor mode, or during speculative parallel executions of the code. In both cases, we reuse the test results by means of inspector hoisting, PDAG and USR common subexpression recognition, and run time test result memoization. The choice between inspector/executor and speculative execution requires a complex cost model. Presently, we choose speculation over inspector/executor only if (1) a parallel inspector cannot be extracted or (2) if we cannot extract a light inspector (a slice made of only scalar definitions). The actual test code generation consists of a syntax-based translation from the PDAG grammar to Fortran.

We apply loop invariant hoisting to USRs and PDAGs by performing aggressive invariance analysis on their sets of input variables. Invariance problems on USRs resulted from subscripted subscripts are formulated as dependence problems on the subscript arrays, which *are solved by the same HDA algorithm applied to the subscript array.* This is achieved by representing the exact referenced memory regions of the subscript array as USR themselves, and thus identifying the exact subregion of the subscript array that affects the shape or size of the memory pattern on the host array. An interesting problem arises when a more expensive test such as LRPD can be hoisted out of a loop, but a simpler O(1) version is loop variant. At this time we (simplistically) hoist tests as far away as possible and build cascades from tests at the same loop nesting level respectively.

CHAPTER VII

EMPIRICAL EVALUATION

The experimental evaluation presented in the following sections will show that Hybrid Analysis (a) extracts a very high degree of parallelism and, often, all the available parallelism from a large number of applications, (b) it is applicable to a large number of applications, (c) allows the generation of minimal run time tests and (d) contributes significantly to the overall parallelization of programs, i.e., they are instrumental in obtaining the presented results.

We have focused on the detection of parallelism rather than on optimizing parallel code execution (e.g. locality enhancement, load balancing). We believe that the major challenge in front us is to detect parallel loops, a step which preconditions any further optimizations. We believe that the consistent solid performance results across a large number of standard benchmark applications proves our claims on the effectiveness of HA. Comprehensive analysis reports, performance tables and graphs can be found at

`http://parasol.tamu.edu/compilers/ha`

A. Methodology

We ran the automatic parallelizer based on HA on a set of industry standard benchmark programs. The parallel code generation is done automatically using OpenMP directives without any further optimizations. The selection of the loops for which parallel code and possibly dynamic tests were generated was based on profiling their sequential execution time. The automatic selection of parallelization candidates based on some more sophisticated performance model is beyond the scope of this paper.

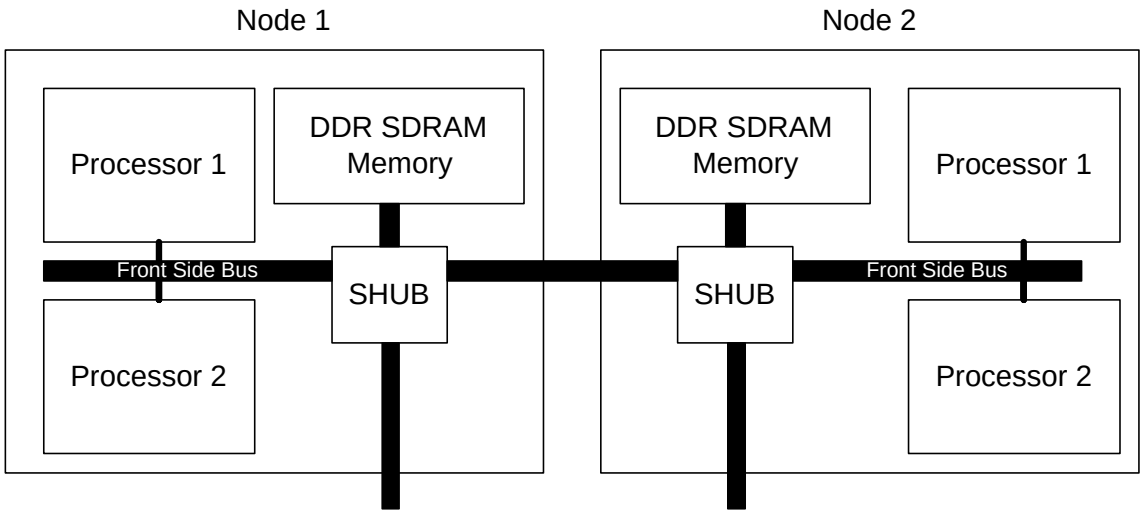


Fig. 71. SGI Altix 3700 system computational brick.

1. Hardware and Software Environment

We believe that the large number of results we obtained on a variety of systems proves the effectiveness of HA. Since Polaris produces Fortran code with OpenMP directives, we specify, for each architecture, the backend compiler used to generate the executable multithreaded code.

We compare against three configurations: First, against the Intel compiler with options *-O -parallel -par_threashold100* on the Altix and MacBook. Second, against the IBM XL compiler with automatic parallelization enabled [125] and options *-O5 -qsmp*. First, against automatic parallelization by SUIF [116]. Unfortunately most of the previous results that we compare against were obtained on different systems that we did not have access to.

Speedups were always measured relative to the sequential execution time on the same machine with the same compiler optimization level.

The SGI Altix 3700 is a cache coherent non-uniform memory access machine (CC-NUMA) at the supercomputing center at Texas A&M University. The machine

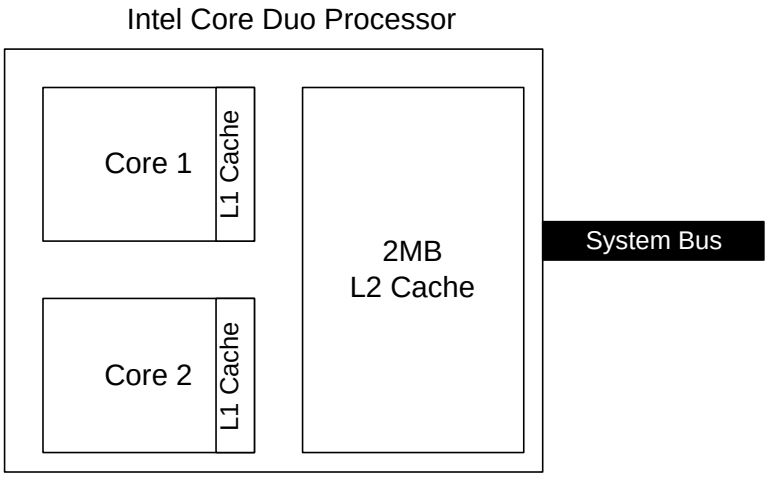


Fig. 72. Intel Core Duo processor in an Apple MacBook notebook computer.

has 128 processors organized in a fat tree. Fig. 71 presents a computational brick. Each such brick contains four processors. The system does not offer guarantees of allocating memory in the RAM of the owner processor. The processors are IA-64 Itanium2 (Madison) 1.3 GHz. The cache coherence protocol is snooping on the FSB within a node and directory based (arbitration in SHUB) across nodes. Each node has 4 GB of memory. We used as a backend compiler the native Intel version 9.0 compiler with options *-O -openmp* (except where noted otherwise).

The SGI Origin 350 we ran on is made of a single module. It has 4 R16000 MIPS processors running at 600 MHz, 8 GB of RAM and individual 4MB L2 cache per each processor. The cache coherence mechanism is unspecified. We used the native SGI compiler with options *-O -mp*.

The Apple MacBook notebook is a single processor system, dual core with shared L2 cache (Fig. 72). Sharing the on-chip L2 cache has the potential to reduce greatly the traffic on the bus and offers the opportunity to implement cheap synchronization mechanisms. The system has 512 MB of RAM. We used the native *Intel version 9.1* compiler with options *-O -openmp*.

We have not run experiments on the SGI Challenge machine. However, we compare against the results reported by [116]. The machine had 4 MIPS R4400 processors running at 200 MHz. The cache coherency mechanism is snooping on the shared bus. The backend compiler and optimization options are not specified.

We have not run experiments on this machine. However, we compare against the results reported by [125]. They show results in a two 1.1 GHz Power4 processors configuration. The backend compiler was *IBM XL Fortran*, with flags *-O5 -qsmp*.

We measured the effectiveness of automatic parallelization using Hybrid Analysis using the Polaris compiler framework. We have not used the data dependence analysis, privatization and reduction parallelization passes already implemented in Polaris because they have applicability limited to intraprocedural loops. We did, however, make full use of several Polaris infrastructure elements, such as value range dictionaries, induction variable recognition and substitution, Gated SSA and interprocedural constant propagation among others. The new parallelization methods based on HA supersede the previous methods in practically all cases.

Polaris produces Fortran code with OpenMP directives. This code was compiled using the native compiler on the target machines. On the Altix machine we used the Intel Fortran Compiler version 9.0 with options *-O -openmp* (except where noted otherwise). On the Apple machine we used the Intel Fortran Compiler version 9.1 with options *-O -openmp* (except where noted otherwise).

2. Input Data Sets

PERFECT [121] is an industry standard benchmark application suite made of floating point scientific computation such as molecular dynamics, structural mechanics or missile tracking, among others. The PERFECT codes have traditionally been harder to optimize automatically because their memory reference patterns appear

Table XV. Experiment environments.

	PERFECT	**SPEC2000**	**SPEC Other**
Polaris/HA	Altix, Macbook	O350, Macbook	Altix, Macbook
Intel	Macbook	Macbook	Macbook
IBM Toronto	-	Power4	-
SUIF	Challenge	-	-

complex and input dependent. Most compilers can extract only low granularity parallelism, which often cannot lead to performance even on a tightly couple parallel machine as the Apple MacBook. However, programmer analysis found [121] large granularity parallelism which can produce significant speedups on even the more loosely coupled parallel machines such as the SGI Altix.

SPEC2000 [190] is the most widely accepted CPU benchmark suite. We only show results on a set of five floating point applications because they were the only ones written in Fortran 77, a prerequisite to using Polaris. One other application written in Fortran 77, *sixtrack* could not be analyzed because of its size, since some components of Polaris do not scale well beyond 10,000 lines of code.

Previous SPEC. We also show results on some previous versions of the SPEC benchmarks (SPEC89 and SPEC92).

Several applications were left out either because our compiler framework could not parse or analyze them correctly, or because they did not contain sufficient parallelism to be of interest to a parallelizing compiler.

We do not have measurements for every pair (platform, data set). Table XV shows the results we obtained and the ones we are comparing against. Most of our results were collected on the Altix and MacBook systems. We report results only for SPEC2000 on the O350 because the results on the Altix were very inconsistent across runs (more than 100% variance). The machine has 128 processors, runs at over 90%

utilization and the processor affinity of data is not guaranteed. The results on the O350 were obtained in single user mode.

Some of the older PERFECT codes were compiled with *-O0* (both the sequential and the parallel versions) *only* on the Altix machine. The reason for this choice was to increase in the most uniform manner the execution times of these codes. These benchmarks and their (initially) reduced input sets have, on today's machines, a very short sequential execution time. Their parallelization, while correct, brings the time of some loops down to the execution time of barriers on a CC-NUMA machine, making it impossible to measure the effect of parallelization. We strongly believe that the structural characteristics are still quite relevant and that expanding the execution times by disabling sequential optimizations is a reasonable experiment for measuring parallelism. In fact they are harder to parallelize than newer benchmarks with larger input sets. For PERFECT codes MDG and TRACK we have larger input sets and thus they have been compiled with *-O*.

3. Performance Metrics

Speedup was computed as $T_{sequential}/T_{parallel}$. $T_{sequential}$ is the wall clock time in seconds of the whole original application run sequentially. $T_{parallel}$ is the wall clock time in seconds of the whole parallelized application, including the overhead incurred by multithreading, run time tests and speculation mechanisms (checkpoint/restore/reexecute). Both $T_{sequential}$ and $T_{parallel}$ were measured on binaries produced with the same optimization level in the backend compiler, and run on the same machine.

Normalized Execution Time is defined as $T_{parallel}/T_{sequential} * 100\%$. It shows the relative reduction in time through parallelization, and is more common when comparing architectures of the same scale, such as multicore processors.

Parallel Coverage is defined as $T_{parallelizable}/T_{sequential} * 100\%$. $T_{parallelizable}$ represents the part of the sequential execution time that is spent in loops that will be parallelized. Based on Amdahl's law, scalable parallelization requires parallel coverage close to 100%.

Granularity of parallelization is measured as $T_{sequential}/N_{global\ synchronizations}$, where $N_{global\ synchronizations}$ is the dynamic count of global synchronization points. Loosely coupled parallel machines such as the 128 processor Altix system with a fat tree interconnection network require high granularity parallelization because the cost of synchronization is high. Low granularity can be tolerated better on the MacBook, where the shared L2 cache allows for very quick synchronization.

B. Hybrid Analysis Automatic Parallelization Results

Fig. 73 presents full application speedups on all the benchmark codes. Automatic parallelization based on HA resulted in speedups of at least 3 on 4 processors for 11 out of 22 applications and of at least 2 on 4 processors on 18 out of 22 applications. The static part of HA is powerful in itself and manages to parallelize more loops than previous static analysis methods in Polaris. Its strength lies primarily in its ability to analyze large interprocedural contexts such as GLOOP_do1000 (over 1,000 lines of code), which could not be previously parallelized by Polaris. More importantly, the speedup improvement through the dynamic component of HA is significant in 8 out of the 22 applications.

OCEAN and *NASA7* (partially) suffer from lack of memory locality in their time consuming FFT loop nests. *APPLU* has outer loop flow dependences and cannot be parallelized using the *DOALL* model. Several loops in *TOMCATV* could not be parallelized at the outermost level resulting in low granularity and limited speedup

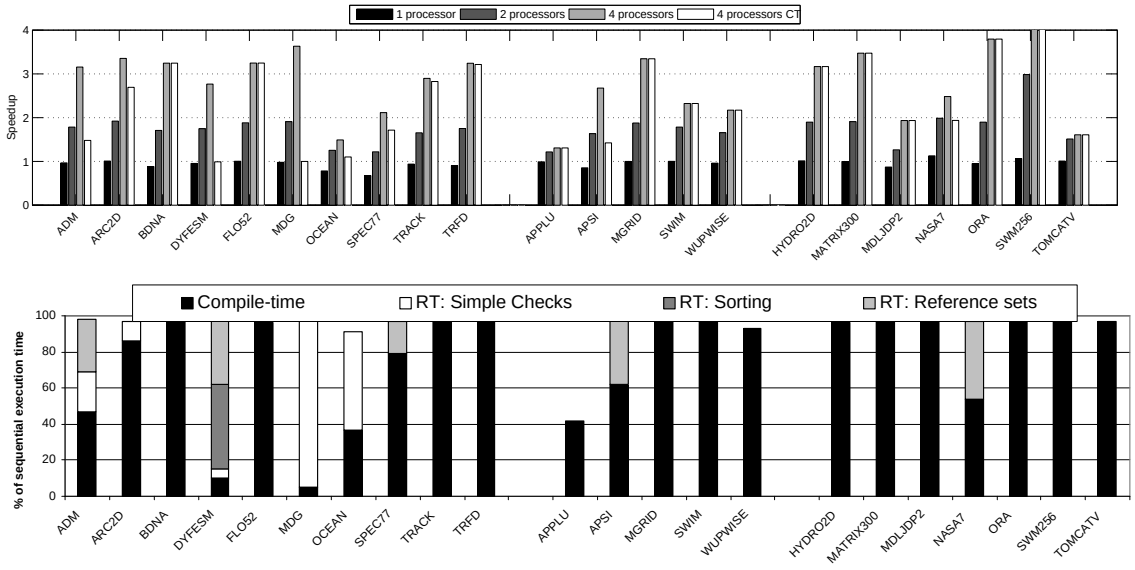


Fig. 73. Hybrid Analysis results on the Altix (PERFECT and Previous SPEC) and the O350 (SPEC2000) systems respectively. In the top graph, the white bars (4 processors CT) correspond to speedups obtained using only compile-time methods and measured on 4 processors.

despite large parallelization coverage.

The second graph in Fig. 73 shows the coverage of parallelization achieved by HA. For 21 out of 22 applications the coverage is over 90% and many are at the 99% level. The exception, *APPLU*, contains a large section with loop-carried flow dependences. The excellent coverage does not sufficiently do justice to the power of HA because it does not quantify the fact that we can detect course grain parallelism (outer loop level) as well as fine grain level (inner loops). The exception was TOMCATV, where the outer loop was found sequential and thus only inner loops were parallelized. In the near future we plan to run our experiments on a machine that supports well nested parallelism in order to better present the quality of the parallelization we obtain.

Fig. 74 shows the behavior of automatically parallelized code on the dual core,

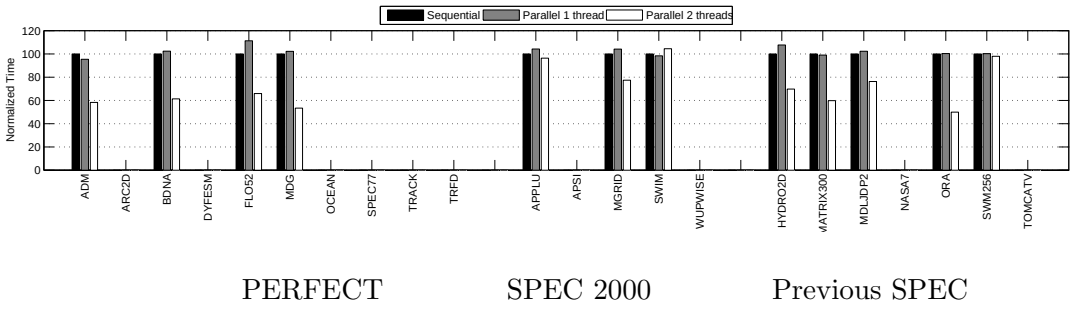


Fig. 74. Polaris/HA normalized execution times on a dual core processor. Sequential, parallel code executed on 1 and 2 threads respectively. All times are normalized to the sequential execution time. Intel Core Duo 1.83MHz, 2x256 MB RAM 5300 / Mac OSX Tiger 10.4.7, XCode 2.2.1, Intel Compiler 9.1.24 -openmp -O. ADM was compiled with *-O0* because the compilation with *-O* resulted in erroneous execution.

single processor system.

C. Evaluation of Run Time Tests

Overall, HA generated 42 tests based on evaluation of elementary conditional expressions, 30 sorted-based tests and 81 based on USR run time evaluations. The parallelization of only 4 loops required the application of the reference-by-reference LRPD test. The second graph in Fig. 73 shows the coverage (and thus importance) of the PDAG technique (evaluation of simple comparisons, sorting-based checks, USR evaluation and reference-by reference LRPD) in parallelizing the codes. Table XIII(b) presents the reduction in dynamic operations achieved by HA relative to reference-by-reference (LRPD) tests as being at least four orders of magnitude in 7 applications. The overhead of run time tests for all the applications that could not be parallelized statically proves to be negligible (less than 0.1%) in most cases. In *ADM*, the overhead of 4.67% is due to the run time evaluation of complex USRs. However, because this run time test can be reused (outer loop invariant) its overhead decreases to 0.1% in

the *APSI* version (much larger input set). The total run time overhead in *MDG* is 2.8% and is partially due to checkpointing for speculative parallelization.

Tables XVI and XVII show in detail which of the major loops required run time tests.

D. Comparison to Other Parallelizing Compilers

We cannot compare our results with any of the previous hybrid parallelization techniques [139, 31, 171, 32, 3] because they did not provide extensive results across whole benchmark suites. The techniques described by [148] are applied to other classes of programs (which involve point to point communication) and cannot be compared directly. This section compares HA against a commercial compiler (Intel) and two research parallelizers (IBM Toronto Lab [125] and SUIF [116]).

1. The Intel® Compiler

Fig. 75 presents a comparison of the performance obtained by Polaris with Hybrid Analysis against the Intel Compiler. Each pair of bars corresponds to the speedups gained by automatic parallelization using the Intel Compiler and Polaris with HA respectively. All execution times were measured on an Apple MacBook notebook with Intel Core Duo 1.83MHz processor, with 2x256 MB RAM 5300 main memory, running Mac OSX Tiger 10.4.7, and using XCode 2.2.1, Intel Compiler 9.1.24 -openmp -O.

The graph shows that in most cases Polaris performs significantly better than the Intel compiler. The difference comes from

- the increased coverage and granularity resulted from dynamic analysis (ADM, MDG, DYFESM)

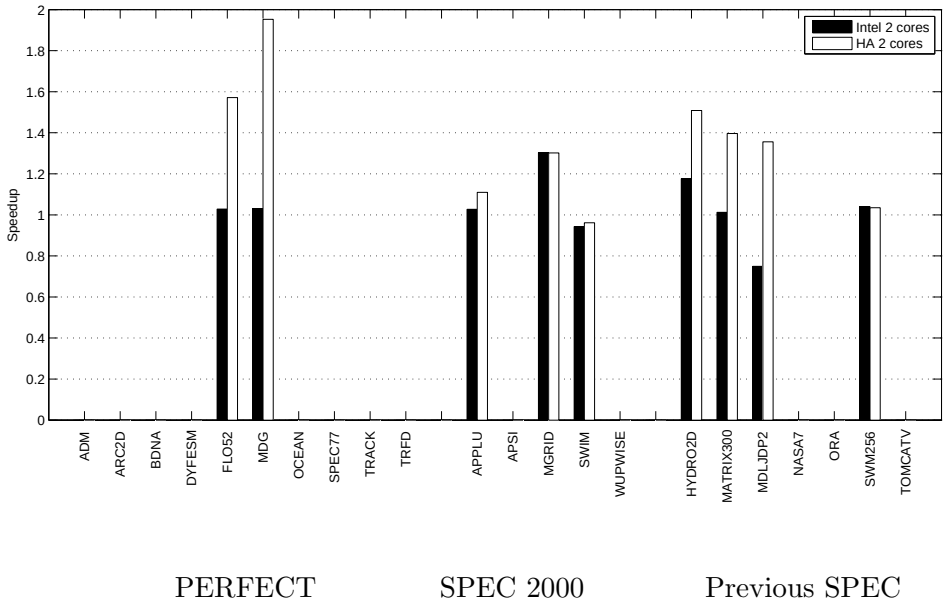


Fig. 75. Comparison of Polaris/HA vs Intel Compiler. Speedup after automatic parallelization on a dual core processor vs. a run of the original application.

- Intel Compiler's lack of reduction recognition and array privatization (ADM, BDNA)
- the more powerful interprocedural array reference analysis mechanism based on USRs and VEG in Polaris (ADM, TRACK, TRFD).

In the two cases where the Intel Compiler does better (MGRID and SWM256), it does marginally so. The same loops are parallelized statically in these two applications by both Polaris/HA and the Intel Compiler.

The last column in Tables XVI and XVII show in detail that most of the major parallelizable loops across all programs are missed by Intel Compiler's parallelization mechanism.

In conclusion, although the Intel Compiler manages to parallelize a large number of smaller loops, this does not translate in speedup even on the tightly coupled dual

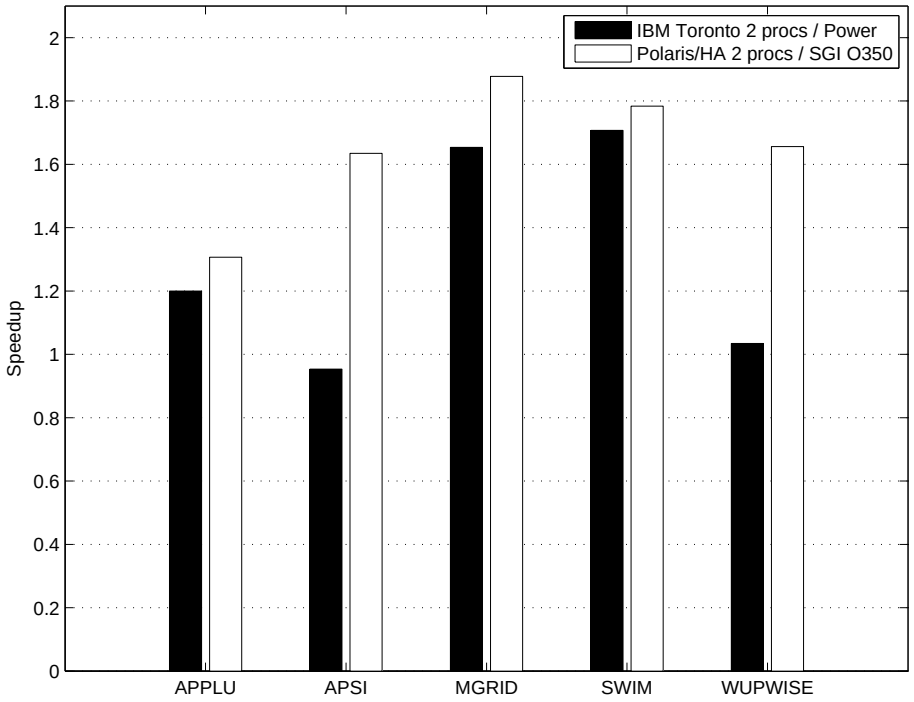


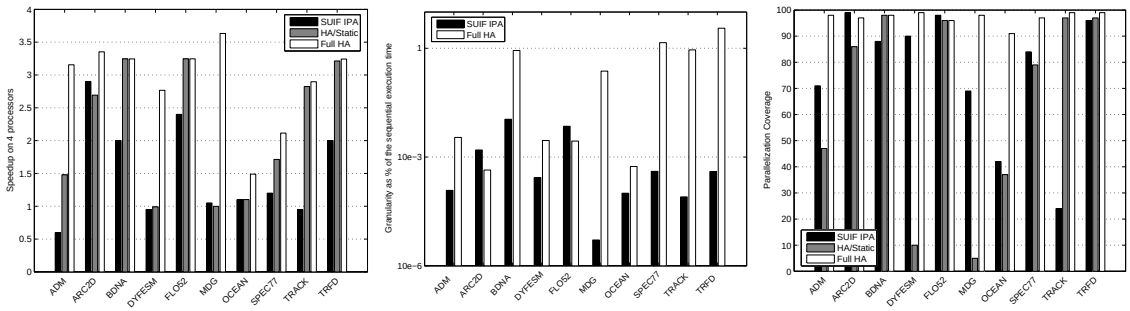
Fig. 76. Speedup comparison between 2 processor runs using Polaris/HA on the SGI O350 and the IBM Toronto Lab parallelizing compiler on a Power machine.

core MacBook machine. This proves that our proposed array analysis techniques are crucial to efficient automatic parallelization.

2. The *IBM Toronto Lab* Parallelizing Compiler

The *IBM Toronto Lab* presented their results in the automatic parallelization of the SPEC2000 benchmarks on a two processor Power4 machine [125]. Fig. 76 shows a comparison between HA and their results, for the five applications that are common to our and their benchmark set. In the cases of APPLU, MGRID and SWIM, the differences are sufficiently small to be attributed to the differences in architectures and backend compilers.

However, the 2-processor run of APSI parallelized by the IBM compiler is slower than the single processor run of the original code, whereas the HA version reduces



(a) (b) (c)

Fig. 77. Comparison of automatic parallelization using HA against SUIF Interprocedural Automatic Parallelizer: (a) Speedup on four processors, (b) Granularity as average duration of a parallelized section expressed as percentage of the execution time, and (c) Coverage as percentage of the execution time.

the running time by about 40%. Table XVII shows that several major loops in APSI require run time tests (RUN_do*).

In WUPWISE there are no run time tests created by HA, but array privatization is necessary to parallelize all major loops.

In conclusion, the IBM parallelizing compiler seems to lack some of the same essential analysis and transformation mechanisms as the Intel compiler: array privatization, reduction recognition and run time analysis techniques.

3. The SUIF Research Compiler

[116] presents the most recent (2005) comprehensive results in automatic parallelization, although based only on static analysis. Their techniques were implemented in the SUIF compiler and include interprocedural data dependence analysis and privatization. Fig. 77 presents a detailed comparison with our results in the automatic parallelization of PERFECT benchmarks. We compare SUIF/IPA with the static part of HA (HA/Static) and with the full, static and dynamic, HA (Full HA). It is not perfectly accurate because the measurements were taken on different machines.

Furthermore, SUIF uses locality enhancement transformations which we did not do.

Program ARC2D was parallelized with approximately same coverage and at the same granularity by both compilers and resulted in similar speedups, which shows that the speedup comparison, although slightly imperfect, is quite relevant. SUIF/IPA shows better parallelization coverage on ADM than the HA/Static. However, HA/Static favors parallelization at a higher level of granularity which results in positive, though modest, speedup. Given an appropriate system we could generate nested parallelism and exploit both fine and coarse grain parallelism. Full HA has better coverage and higher granularity resulting in speedup of more than 3 on 4 processors. The execution time in ADM (and more so in DYFESM and MDG) is dominated by large loops that iterate over the whole data set and which can only be parallelized at run time. SUIF/IPA manages to parallelize only inner loops and gets good coverage but cannot achieve speedups. Full HA parallelizes them at the highest level of granularity available and achieves good speedups on all three of them.

In conclusion, the SUIF compiler has powerful array privatization and reduction parallelization techniques based on interprocedural analysis. However, its performance is limited for the cases where the reference patterns are not linear or/and input dependent. Additionally, VEG-based value analysis leads to better results even when comparing statically parallelized loops in ADM, BDNA and TRACK.

Our hybrid (static and dynamic) methods often find parallelism at the outer level of large nests spanning multiple loops in different subprograms, involving indirection and complex control. This leads to better performance on both NUMA machines as well as more tightly coupled dual cores. Hybrid Analysis increases the efficiency of classic dynamic methods and produces scalable speedups close to the maximum performance level achievable through multithreading.

E. Discussion of Important Loops

Tables XVI and XVII present the most important loops from the PERFECT and SPEC benchmark suites that were parallelized by Polaris using Hybrid Analysis. The *%* column shows the relative importance of the loop as the percentage of execution time spent in the loop during a sequential run of its host application. The *DD Test* column classifies the data dependence tests needed for parallelization. The most common one is CT, which means the loop was proven independent at compile time. Run time dependence tests are classified into SE (simple expressions such as $n \leq 100$ as well as loops over simple expressions), IT (interval trees), UE (evaluation of the USR that describes the dependence set at run time, followed by comparison against the empty set), and LRPD (reference instrumentation and analysis of the resulting trace). The fifth and sixth columns (*Priv* and *Red*) show whether privatization and reduction parallelization were necessary. *CT* means that privatization was proven necessary at compile time. In some cases such as *ADM/RUN_do20*, the legality of the privatization transformation is proven at run time as part of the DD Test. *RT* marks the cases when they appeared to be necessary at compile time, but could have been proven unnecessary at run time (so unnecessary copy-in, copy-out and reduction operations would have been avoided). The *PB* column shows which sequences were recognized as pushbacks and parallelized. The *IPA* column shows which loops contained subprogram calls, thus required interprocedural memory reference analysis. However, not only the loops that contain subprogram calls required interprocedural analysis. For instance, it is possible that an inspector be hoisted interprocedurally to its definition point even though the loop it is extracted from does not contain any subprogram calls. The *Exec* column shows, for the cases when a run time dependence test was required, whether the execution strategy was inspector/executor or specula-

Table XVI. Loop parallelization in PERFECT codes. % = percentage of total application execution time. DD Test = type of data dependence test required (CT = compile time, RT = run time, SE = simple logical expressions, IT = interval trees, UE = USR evaluation, LRPD = LRPD run time test) Priv = type of privatization required (A = array privatization). Red = type of reduction required. PB = pushback required. IP = loop contains subprogram calls. EX = execution type (IE = inspector/executor, SP = speculative execution). Intel = parallelized automatically by the Intel Compiler (version 9.0, *-parallel -par_threshold100*).

Code	**Loop**	**%**	**DD Test**	**Priv**	**Red**	**PB**	**IP**	**EX**	**Intel**
ADM	RUN_do20,...,100	44	RT:SE,UE	CT,A	-	-	√	IE	-
	D*DTZ_do30	31	CT	CT,A	CT	-	√	-	-
	DKZMH_do20,50	11	CT	CT,A	-	-	√	-	-
	WCONT_do40	5	CT	CT,A	CT	-	√	-	-
ARC2D	STEPF*_do*	29	CT	CT	-	-	-	-	√
	*PENT*_do*	14	CT	CT	-	-	-	-	√
	FILERX_do15	14	RT:SE,UE	CT,A	-	-	-	IE	-
	RHS*_do*	10	CT	CT	-	-	-	-	√
	TK*_do1	8	CT	CT	-	-	-	-	-
BDNA	ACTFOR_do240,500	89	CT	CT,A	CT	-	-	-	-
DYFESM	MXMULT_do10	73	RT:IT,UE	RT:IT,A	RT:IT,UE	-	√	IE	-
	SOLVH_do20	9	RT:SE	RT:IT,A	-	-	√	IE	-
	FORMR0_do20	7	RT:IT,UE	RT:IT,A	RT:IT,UE	-	√	IE	-
	SOLXDD_do4,10,30,50	9	RT:IT	RT:IT,A	RT:IT	-	√	IE	-
FLO52	*FLUX*_do*	55	CT	CT	-	-	-	-	√
	PSMOO_do40,80	21	CT	CT	-	-	-	-	-
	EULER_do*	15	CT	CT	CT	-	-	-	√
MDG	INTERF_do1000	93	RT:SE	CT,A	CT	-	√	SP	-
	POTENG_do2000	6	CT	CT,A	CT	-	√	-	-
OCEAN	FTRVMT_do109	41	RT:SE	CT	-	-	-	IE	-
	IN_do10	15	CT	-	-	-	-	-	-
	OUT_do10	15	CT	-	-	-	-	-	-
	CSR,RCS_do20	7	CT	CT	-	-	-	-	-
	ACAC,SCSC_do30,40	6	CT	CT,A	-	-	-	-	-
SPEC77	GLOOP_do1000	48	CT	CT,A	CT	-	√	-	-
	GWATER_do1000	24	RT:LRPD	CT,A	CT	-	√	SP	-
	SICDKD_do1000	4	CT	CT,A	-	-	√	-	-
TRACK	EXTEND_do400	50	CT	CT,A	-	√	√	-	-
	FPTRAK_do300	46	CT	CT,A	-	√	√	-	-
	NLFILT_do300	2	RT:LRPD	CT,A	-	-	√	SP	-
TRFD	OLDA_do100	67	CT	CT,A	-	-	-	-	-
	OLDA_do300	28	CT	CT,A	-	-	-	-	-
	INTGRL_do140	3	RT:IT	RT:IT,A	-	-	-	IE	-

Table XVII. Loop parallelization in SPEC codes. (Legend in Table XVI.)

Code	Loop	%	DD Test	Priv	Red	PB	IP	EX	Intel
APPLU	JACL*_do#1	34	CT	CT	-	-	-	-	-
	RHS_do#1,2,3,4	20	CT	CT	-	-	-	-	√
APSI	RUN_do*	25	RT:SE,UE	CT,A	-	-	√	IE	-
	D*DTZ_do40	40	CT	CT,A	CT	-	√	-	-
	DKZMH_do30,60	12	CT	CT,A	-	-	√	-	-
	WCONT_do40	6	CT	CT,A	CT	-	√	-	-
	HYD_do20	5	CT	CT	CT	-	-	-	-
MGRID	RESID_do600	52	CT	CT	-	-	-	-	√
	PSINV_do600	27	CT	CT	-	-	-	-	√
	RPRJ3_do100	7	CT	CT	-	-	-	-	√
	INTERP_do400,800	8	CT	CT	-	-	-	-	√
	COMM3_do100,200,300	5	CT	CT	-	-	-	-	-
SWIM	SHALLOW_do3500	48	CT	CT	CT	-	-	-	-
	CALC1_do100	14	CT	CT	-	-	-	-	√
	CALC2_do200	17	CT	CT	-	-	-	-	√
	CALC3_do300	19	CT	CT	-	-	-	-	√
WUPWISE	MULDEO_do100,200	47	CT	CT,A	-	-	√	-	-
	MULDOE_do100,200	46	CT	CT,A	-	-	√	-	-
HYDRO2D	FILTER_do*	42	CT	CT	-	-	-	-	√
	FCT_do*	18	CT	CT	-	-	-	-	√
	ARTDIF_do*	14	CT	CT	-	-	-	-	√
	TRANS*_do*	12	CT	CT	-	-	-	-	√
	TISTEP_do*	6	CT	CT	-	-	-	-	√
	S1,S2_do100	4	CT	CT	-	-	-	-	-
MATRIX300	LBMK14_do20	13	CT	CT	-	-	-	-	-
	SGEMM_do*	86	CT	-	-	-	√	-	-
MDLJDP2	FRCUSE_do20	76	CT	CT	CT	-	√	-	-
	FRCBLD_do20	11	CT	CT	CT	√	√	-	-
	POSTFR_do*	8	CT	CT	CT	-	-	-	-
	PREFOR_do*	5	CT	CT	-	-	-	-	-
NASA7	VPETST_do110	26	CT	CT	-	-	√	-	-
	GMTTST_do120	24	RT:UE	CT	-	-	√	IE	-
	CFFT2D*_do130,150	17	RT:LRPD	CT	-	-	-	SP	-
	BTRTST_do120	10	CT	CT	-	-	√	-	-
	CHOTST_do120	9	CT	CT	-	-	√	-	-
	EMIT_do5	6	CT	RT:IT,A	-	-	-	IE	-
ORA	MAIN_do9999	99	CT	CT	CT	-	√	-	-
SWM256	CALC1_do100	31	CT	CT	-	-	-	-	√
	CALC2_do200	38	CT	CT	-	-	-	-	√
	CALC3_do300	30	CT	CT	-	-	-	-	√
TOMCATV	MAIN_do100/2,120/2,60,...	96	CT	CT	CT	-	-	-	√

tive execution. The *Intel* column shows whether the *Intel® Compiler* parallelized the corresponding loop. We did not have access to loop level analysis results from *SUIF* and the *IBM Toronto Lab* parallelizing compiler.

The tables show that the methods presented in this dissertation are not only applicable, but instrumental to the efficient automatic parallelization of real programs. The remainder of this section matches the techniques presented in this dissertation to the corresponding parallelized loops.

1. Value Evolution Graph

The information produced by VEGs was used throughout the whole USR based memory reference analysis. It led to more parallelization problems being solved at compile time, such as the array privatization problems in loops *ADM/DKZMH_do50*, *APSI/DKZMH_do60*, *BDNA/ACTFOR_do240* and *TRACK/FPTRAK_do300*. Push-back Sequence Parallelization based on the VEG achieved almost full parallelization of application *TRACK* and applied to *MDLJDP2/FRCBLD_do20* as well.

It also led to more accurate and lighter run time tests, by eliminating impossible scenarios at compile time, which would otherwise have to be verified at run time.

The contributions of the VEG are discussed in detail in Chapter IV.

2. USR Based Memory Reference Analysis

The majority of the important loops required one or more of the following techniques: interprocedural analysis of loops containing subprogram calls, array privatization, and reduction and pushback recognition and parallelization, None of these techniques are used by the Intel compiler, but they are all implemented with little effort using the Memory Reference Analysis based on USRs and PDAGs.

The USRs offer a precise interprocedural view of memory reference patterns

without the combinatorial explosion inherent to inline expansion. They represent linear patterns symbolically using LMADs, and pinpoint the exact location of non-linear expressions: nonlinear control conditions or nonlinear array subscripts or loop bounds.

USRs scaled well to loops over 1,000 lines of code (GLOOP_do1000). Their graph structure allows quick recognition of similar tests. The dependence tests of loops *RUN_do20*, *RUN_do30* and *RUN_do40* were found identical at compile time, thus two unnecessary run time tests were avoided.

The contribution of the USR is presented in detail in Chapter III.

3. PDAG Based Efficient Run Time Tests

We only fell back to the LRPD test in a small number of cases: *GWATER_do1000*, *NLFILT_do300* and *CFFT2D*_do130,150*. In all cases, the extracted PDAGs did not contain any information simpler than an LRPD test and a USR evaluation test appeared too expensive due to the total lack of symbolic memory reference aggregation. The contribution of the PDAG is presented in detail in Chapter III.

a. ADM/APSI

Loops *RUN_do20,...,100* can be parallelized only after arrays *SAVEX*, *SAVEY*, *HELP* and *HELPA* are proven privatizable. For the privatization of *SAVEY* we extract an optimistic condition *NY.LE.1*, and for all the arrays we generate run time tests based on USR evaluation. Although the number of memory references in each loop is $\Theta(nsteps*nx*ny*nz*nfact)$, the complexity of the run time test is $\Theta(nfact)$ (though with a large constant factor). The reduction factor comes from aggregation $\Theta(nx)$, loop invariance $(ny*nz)$, and test reuse $(nsteps)$.

b. MDG

The most important loop, *INTERF_do1000*, although always parallel, requires complex symbolic reasoning which appears not to be available in any of the compilers under test, thus the loop is not parallelized using static methods. The *Intel* compiler reports it as sequential. Both *Intel* and *SUIF* report minimal speedups for 2 respectively 4 processing units, while *Polaris* with Hybrid Analysis produces a speedup of more than 1.8 on the Core Duo and 3.5 on the Altix. Although SUIF shows significant parallelization coverage, this happens at very low granularity, which results in low performance gain.

c. DYFESM

All the important loops required run time tests because almost all data are referenced through indirection. However, most data are accessed in contiguous blocks, so tests based on reference instrumentation would be suboptimal. Hybrid Analysis generated successful tests based on checking a scalar symmetry condition (variable *nsymm* in loop SOLVH_do20) or based on interval trees in all other major loops.

d. OCEAN

Loop *FTRVMT_do109* cannot be found parallel at compile time because an expansion operation cannot be proved nonoverlapping. This is a particular case of the nonoverlapping intervals test in a multidimensional space with all the intervals having the same size.

4. Dynamic Parallelization, Privatization and Reduction

Chapter V presents a case study from *DYFESM/MXMULT_do10* in which privatization and reduction is decided at run time. Several other loops in *DYFESM* have similar patterns that can only be decided at run time: *SOLXDD_do4,10,3,50*, *FORMR0_do20* and *SOLVH_do10*. In almost all of them, privatization with dynamic last value or reduction parallelization are avoided at run time. Privatization with dynamic last value computation is also avoided in *TRFD/INTGRL_do140*.

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

Compiler Optimization research has, for the most part, taken to two directions. Static Analysis was always preferred because it does not cause execution overhead, but misses optimization opportunities when decisions are input dependent. Dynamic Analysis is precise but it incurs overhead that reduces the profitability of optimizations.

We proposed a hybrid compiler optimization model, a novel way to bridge static and dynamic memory reference analysis. Rather than making conservative decisions at compile time, the hybrid optimizer extracts predicates that can validate optimizing transformations at run time, often with minimal costs. Instead of only answering the question of whether an optimization is legal, it also generates the dynamic conditions under which it would be legal. These conditions are frequently inexpensive to evaluate at run time and thus increase the efficiency or run time optimization to the point where they are almost always profitable.

The advantage of Hybrid Analysis over traditional methods comes from its ability to use partial symbolic results. These results are often not sufficient to make a decision at compile time. On the other hand, they are ignored by run time methods, which redo the entire analysis process for each dynamic instance resulting in high overhead. Hybrid analysis extracts conditions from partially aggregated information which leads to run time tests of reduced complexity.

We implemented a full working Hybrid Optimization framework in the Polaris research compiler. Its backbone consists of an analytical representation for memory reference sets across arbitrarily large program contexts and of a predicate extraction technique that can extract sufficient conditions from identities involving sets of mem-

ory references. We also implemented a symbolic value comparison and logic reasoning module that is used to simplify the analytical memory reference set descriptors. The entire analysis process is interprocedural and control-flow sensitive.

We used this framework to implement a hybrid automatic parallelizer in the Polaris research compiler, which resulted in program speedups of at least 2 on 4 processors, on 18 out of 22 industry standard benchmark applications

`http://parasol.tamu.edu/compilers/ha`

A. Contributions

1. Program Representation

Classic symbolic memory reference analysis techniques resort to approximation when they fail to represent a reference set using linear constraints. In order to collect precise information, Hybrid Analysis needs a representation for memory references that can tolerate such static analysis failures and continue the analysis process without resorting to approximation, thus preserving all the opportunities for optimization. We proposed the *Uniform Set of References* (USR) as a representation for sets of memory references that is *closed* in a scalable manner with respect to all the operations performed by a large number of analysis techniques, over arbitrarily large program contexts.

We also proposed a representation for the flow of program values, the *Value Evolution Graph*, that can produce symbolic value range information with meaningful accuracy even in the presence of complex recurrences and control flow. This leads to a powerful symbolic calculator that performs comparisons of values such as USR offsets, strides and loop bounds, and that solves logic queries such as implication of control dependence predicates.

2. Program Analysis

Hybrid Analysis aggregates memory references as USRs across arbitrarily large program contexts. It also partitions the memory references three ways, based on the whether they are only read, written before read, or read and written, which is needed to preserve the original memory access order information. These results are used by the automatic parallelizer to formulate data dependence and data flow questions.

Hybrid Analysis formulates a dependence test as $\mathit{Dependence\ Set} = \emptyset$, where *Dependence Set* is the set of all dependent memory locations, expressed as a USR. When this identity cannot be verified at compile time, we extract the sufficient conditions under which it holds. The predicate extraction process follows the USR structure of the dependence set. It extracts predicates as simple as bound checks and as complex as dynamic reference instrumentation, which are organized in a cascade of simple-to-complex run time tests.

B. Future Work

1. Extending Hybrid Analysis

Although Hybrid Analysis has only been applied so far to programs written in *Fortran 77*, it is not limited to them because it is a paradigm of analysis and not a specific technique. However, the analysis and optimization of *Fortran 77* programs has traditionally been easier than that of programs written in languages with weaker aliasing restrictions. It is important to investigate the effectiveness of the hybrid paradigm beyond *Fortran 77*.

a. Hybrid Pointer Analysis

There are several methods to perform data dependence analysis when memory is referenced through pointers. When the pointers are initialized from the address of array elements, the analysis may reduce to substituting pointers with array references, and then applying the previously described methods [86, 57, 127]

In other cases, pointer references can be proved disjoint by proving that they point to disjoint memory spaces such as disjoint arrays [191, 192, 70, 193, 194, 195, 196, 156, 197], although this is in general an NP-hard problem [198] even for flow insensitive problems in intraprocedural contexts.

The pointer problems become more complex with dynamically linked data structures, when data dependence decisions are made based on the shape of the data structure and an associated traversal. [70] presents dependence analysis for recursive tree traversals (including tree modification), list-like traversals and arrays of pointers traversals. [71, 72, 73] present more research in symbolic shape analysis for linked data structures for dependence (and other) analysis.

[199] presents a general SSA numbering scheme for pointer dereferencing: it stores, for every pointer reference, the number of the reaching definition of the variable referenced by the pointer. [156] presents pointer analysis for thread-level speculation. [200] presents probabilistic points-to analysis to be used for data flow speculation. [201] presents static analysis of pointers and arrays for verification of C programs.

We intend to apply the hybrid paradigm to pointer analysis. Some cases when pointers are bound to arrays allocated statically or on the stack can be applied the techniques presented in this dissertation. However, dynamically allocated linked data structures require a different analysis model, such as escape or shape analysis. Traditional compile time pointer analysis often fails because it relies on imprecise symbolic

program information. On the other hand, reference based run time methods are not viable because they may require an amount of storage proportional to the number of locations that the pointer may address. We believe that a hybrid approach can increase the rate of success of static pointer analysis with reduced run-time overhead.

b. Hybrid Optimization for High Level Languages

Analyzing *C* programs is hard not only to optimizing compilers but also to the programmers who develop and maintain them. Modern *C++* programs make extensive use of standard library containers which limit aliasing in a way similar to *Fortran 77* arrays, while still reaping the benefits of linked structures such as lists and trees. We plan to investigate Hybrid Analysis techniques based on container semantics. This approach can be generalized to programs written in any high level language in which operations are implemented through standardized mechanisms (such as the *C++ Standard Library*). The fundamental advantage of such an approach is that it can rely on very high level semantic information guaranteed by the programming language standard, which would be otherwise impossible to extract from a syntax tree.

Checking the legality of optimizations is a particular case of automatic verification. We want to research the possibility of reducing the overhead of other types of verification such as correctness proofs or high level debugging for domain specific languages. For instance, we want to investigate the possibility of developing an automated data race violation checker for parallel programs written using parallel libraries such as MPI or STAPL.

2. Applications of Hybrid Dataflow Analysis

Data flow information is crucial to other compiler technologies. Verification and symbolic debugging require understanding of the flow of values and the alias relations needed to compute it. Our proposed Hybrid Analysis techniques produce accurate data flow information can can be used in all the following applications. There are two types of data flow problems. First, they try to prove the lack of data flow between two statements. Hybrid Analysis solves this problem by extracting a PDAG from the equations $DF = \emptyset$, where DF is the USR that describes the exact array region on which there is data flow. Second, USRs can be used to describe the data flow relations necessary to generate communication for parallelization for distributed memory systems.

a. Generation of Communication Schedules

The LMAD [202], Last Write Trees [203] and a variety of other representations and/or techniques [204, 205, 206, 207, 208, 209] have been used to generate communication schedules for parallelization for distributed memory systems.

[141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152] presented run time methods to produce highly efficient communication schedules. Their techniques try to identify the exact locations of data flow source and destination, which allows them to optimize the placement of communication primitives.

b. Compiler-based Cache Coherence

[210] presents interprocedural array dataflow analysis to detect stale memory references on non-cache-coherent hardware and its uses in compiler-generated cache coherency. [118] presents memory behavior of compiler-generated parallel code and

parallelization artifacts that lead to false sharing. [211, 212] presents dataflow analysis for compilation for non-cache-coherent machines. [212] uses LMAD-based array analysis and employs get/put instead of send/recv. [124] presents a method for spatial locality improvement and reduction of false sharing.

c. Symbolic Debugging and Verification

[213, 214] discuss debugging issues for optimized or parallelized programs. [215] presents applications of dataflow analysis to implementing efficient checks for algorithm-based fault tolerance. [216] presents the creation of safe regions in Java programs, i.e. exception-checks free, in which Fortran-like optimization are possible. [217] introduces three analysis methods to remove runtime bound checks for Java arrays. [138] present a hybrid (static and dynamic) method to detect uninitialized variables. Array sections proved statically to be initialized before use are excluded from runtime checks. However, the array region analysis is conservative (memory region operations are not hybrid).

d. Other Uses

[33] presents array dataflow analysis with array regions as constraint sets with application to constant propagation. [218] presents iteration reordering for grouping references to the same memory bank together. The goal is to leave some banks untouched long enough so that they can be efficiently put in low-power mode. Other applications include global common subexpression elimination, scalarization of array references, live variable analysis for register allocation and memory exclusion for checkpoint size reduction.

3. Dynamic Compilation Based on Input Sensitivity

The Hybrid Optimization paradigm blurred the line between compile and run time. However, our current optimization framework relies entirely on compile time generation of highly parameterized code, which may lead to suboptimal performance due to additional logic, allocation of memory that may be never used, and increased binary size. Dynamic compilation solves this problem by generating only the optimized version that corresponds to the actual values, but incurs the additional dynamic code generation overhead that may offset its benefits.

We want to pursue a new dynamic compilation model in which recompilation is triggered by the *input sensitivity* of the validity and profitability of the optimization (the set of values that may influence the optimization decision), which can be computed using Hybrid Analysis. The definition site of the input sensitivity set pinpoints the recompilation point for producing specialized versions for a specific optimization and a given program slice.

REFERENCES

[1] L. Rauchwerger and D. A. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization." *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 2, pp. 160–180, 1999.

[2] S. Rus, L. Rauchwerger, and J. Hoeflinger, "Hybrid analysis: static & dynamic memory reference analysis," in *ICS '02: Proceedings of the 16th International Conference on Supercomputing*. New York: ACM Press, 2002, pp. 274–284.

[3] S. Rus, L. Rauchwerger, and J. Hoeflinger, "Hybrid analysis: Static & dynamic memory reference analysis." *International Journal of Parallel Programming*, vol. 31, no. 4, pp. 251–283, 2003.

[4] S. Rus, D. Zhang, and L. Rauchwerger, "Automatic parallelization using the value evolution graph." in *LCPC 2004: the 17th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 3602. New York: Springer, 2004, pp. 379–393.

[5] S. Rus, D. Zhang, and L. Rauchwerger, "The value evolution graph and its use in memory reference analysis." in *13th International Conference on Parallel Architectures and Compilation Techniques (PACT 2004)*. Washington, DC: IEEE Computer Society, 2004, pp. 243–254.

[6] S. Rus, G. He, and L. Rauchwerger, "Scalable array SSA and array data flow analysis," in *LCPC 2005: the 18th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. to appear. New York: Springer, 2005.

[7] S. Rus, G. He, C. Alias, and L. Rauchwerger, "Region array SSA," in *15th International Conference on Parallel Architectures and Compilation Techniques (PACT 2006)*. Washington, DC: IEEE Computer Society, 2006, pp. 43–52.

[8] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, Mass.: Addison-Wesley, 1986.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and K. Zadeck, "An efficient method of computing static single assignment form," in *16th Annual ACM Symposium on Principles of Programming Languages*, Austin, Tex., Jan. 1989, pp. 25–35.

[10] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein, "The Program Dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages," in *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, N.Y., June 1990, pp. 257–271.

[11] T. Fahringer and B. Scholz, "Symbolic evaluation for parallelizing compilers," in *ICS '97: Proceedings of the 11th International Conference on Supercomputing*. New York: ACM Press, 1997, pp. 261–268.

[12] K. Gargi, "A sparse algorithm for predicated global value numbering," in *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. New York: ACM Press, 2002, pp. 45–56.

[13] Y. Paek, J. Hoeflinger, and D. Padua, "Simplification of array access patterns for compiler optimizations," in *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. New York: ACM Press, 1998, pp. 60–71.

[14] J. Hoeflinger, Y. Paek, and K. Yi, "Unified interprocedural parallelism detection." *International Journal of Parallel Programming*, vol. 29, no. 2, pp. 185–215, 2001.

[15] Y. Paek, J. Hoeflinger, and D. Padua, "Efficient and precise array access analysis," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 1, pp. 65–109, 2002.

[16] T. N. Nguyen, J. Gu, and Z. Li, "An interprocedural parallelizing compiler and its support for memory hierarchy research." in *LCPC 1995: the 8th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 1033. New York: Springer, 1995, pp. 96–110.

[17] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S.-W. Liao, and M. S. Lam, "Interprocedural analysis for parallelization." in *LCPC 1995: the 8th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 1033. New York: Springer, 1995, pp. 61–80.

[18] W. Pugh and D. Wonnacott, "Eliminating false data dependences using the omega test," in *PLDI '92: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation.* New York: ACM Press, 1992, pp. 140–151.

[19] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam, "Array-data flow analysis and its use in array privatization," in *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* New York: ACM Press, 1993, pp. 2–15.

[20] K. Knobe and V. Sarkar, "Array SSA form and its use in parallelization," in *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York: ACM Press, 1998, pp. 107–120.

[21] D. Callahan, "The program summary graph and flow-sensitive interprocedual data flow analysis," in *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. New York: ACM Press, 1988, pp. 47–56.

[22] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," in *SIGPLAN '86: Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*. New York: ACM Press, 1986, pp. 152–161.

[23] P. Havlak and K. Kennedy, "Experience with interprocedural analysis of array side effects," in *Supercomputing '90: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 1990, pp. 952–961.

[24] M. W. Hall, J. M. Mellor-Crummey, A. Carle, and R. G. Rodríguez, "Fiat: A framework for interprocedural analysis and transfomation." in *LCPC 1993: the 6th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 768. New York: Springer, 1993, pp. 522–545.

[25] V. Maslov, "Lazy array data-flow dependence analysis," in *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York: ACM Press, 1994, pp. 311–325.

[26] B. Creusillet and F. Irigoin, "Interprocedural array region analyses." in *LCPC 1995: the 8th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 1033. New York: Springer, 1995, pp. 46–60.

[27] E. Duesterwald, R. Gupta, and M. L. Soffa, "Demand-driven computation of interprocedural data flow," in *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York: ACM Press, 1995, pp. 37–48.

[28] E. Duesterwald, R. Gupta, and M. L. Soffa, "A practical framework for demand-driven interprocedural data flow analysis," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 6, pp. 992–1030, 1997.

[29] J. Gu, Z. Li, and G. Lee, "Experience with efficient array data flow analysis for array privatization," in *PPOPP '97: Proceedings of the sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York: ACM Press, 1997, pp. 157–167.

[30] D. E. Maydan, S. Amarsinghe, and M. S. Lam, "Data dependence and data-flow analysis of arrays." in *LCPC 1992: the 5th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 757. New York: Springer, 1992, pp. 434–448.

[31] S. Moon, M. W. Hall, and B. R. Murphy, "Predicated array data-flow analysis for run-time parallelization," in *ICS '98: Proceedings of the 12th International Conference on Supercomputing*. New York: ACM Press, 1998, pp. 204–211.

[32] S. Moon and M. W. Hall, "Evaluation of predicated array data-flow analysis for automatic parallelization," in *PPoPP '99: Proceedings of the seventh ACM SIG-*

PLAN Symposium on Principles and Practice of Parallel Programming. New York: ACM Press, 1999, pp. 84–95.

[33] D. Wonnacott, "Extending scalar optimizations for arrays." in *LCPC 2000: the 13th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 2017. New York: Springer, 2000, pp. 97–111.

[34] R. Seater and D. Wonnacott, "Polynomial time array dataflow analysis." in *LCPC 2001: the 14th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 2624. New York: Springer, 2001, pp. 411–426.

[35] Y. Lin and D. Padua, "Compiler analysis of irregular memory accesses," in *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation.* New York: ACM Press, 2000, pp. 157–168.

[36] J. Knoop, B. Steffen, and J. Vollmer, "Parallelism for free: efficient and optimal bitvector analyses for parallel programs," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 3, pp. 268–299, 1996.

[37] J.-F. Collard and M. Griebl, "A precise fixpoint reaching definition analysis for arrays." in *LCPC 1999: the 12th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 1863. New York: Springer, 1999, pp. 286–302.

[38] E. Mehofer and B. Scholz, "A novel probabilistic data flow framework." in *CC 2001: the 10th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 2027. New York: Springer, 2001, pp. 37–51.

[39] M. R. Haghighat and C. D. Polychronopoulos, "Symbolic analysis for parallelizing compilers," *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 4, pp. 477–518, 1996.

[40] M. Wolfe, "Experiences with data dependence abstractions," in *ICS '91: Proceedings of the 5th International Conference on Supercomputing*. New York: ACM Press, 1991, pp. 321–329.

[41] E. Stoltz and M. Wolfe, "Detecting value-based scalar dependence." in *LCPC 1994: the 7th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 892. New York: Springer, 1994, pp. 186–200.

[42] S.-W. Liao, A. Diwan, J. Robert P. Bosch, A. Ghuloum, and M. S. Lam, "Suif explorer: an interactive and interprocedural parallelizer," in *PPoPP '99: Proceedings of the seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York: ACM Press, 1999, pp. 37–48.

[43] P. Feautrier, "Parametric integer programming," *Operations Research*, vol. 22, no. 3, pp. 243–268, 1988.

[44] K. Kyriakopoulos and K. Psarris, "Data dependence analysis techniques for increased accuracy and extracted parallelism." *International Journal of Parallel Programming*, vol. 32, no. 4, pp. 317–359, 2004.

[45] G. Goff, K. Kennedy, and C.-W. Tseng, "Practical dependence testing," in *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. New York: ACM Press, 1991, pp. 15–29.

[46] K. Psarris, "On exact data dependence analysis," in *ICS '92: Proceedings of the 6th International Conference on Supercomputing.* New York: ACM Press, 1992, pp. 303–312.

[47] P. M. Petersen and D. A. Padua, "Static and dynamic evaluation of data dependence analysis," in *ICS '93: Proceedings of the 7th International Conference on Supercomputing.* New York: ACM Press, 1993, pp. 107–116.

[48] Z. Li, P.-C. Yew, and C.-Q. Zhu, "Data dependence analysis on multi-dimensional array references," in *ICS '89: Proceedings of the 3rd International Conference on Supercomputing.* New York: ACM Press, 1989, pp. 215–224.

[49] M. Wolfe and C.-W. Tseng, "The Power test for data dependence," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 5, pp. 591–601, Sept. 1992.

[50] D. E. Maydan, J. L. Hennessy, and M. S. Lam, "Efficient and exact data dependence analysis," in *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation.* New York: ACM Press, 1991, pp. 1–14.

[51] J. Subhlok and K. Kennedy, "Integer programming for array subscript analysis." *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 6, pp. 662–668, 1995.

[52] W.-L. Chang and C.-P. Chu, "The i+ test." in *LCPC 1998: the 11th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 1656. New York: Springer, 1998, pp. 367–381.

[53] W. Pugh and D. Wonnacott, "An exact method for analysis of value-based

array data dependences." in *LCPC 1993: the 6th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 768. New York: Springer, 1993, pp. 546–566.

[54] W. Blume and R. Eigenmann, "The range test: a dependence test for symbolic, non-linear expressions," in *Supercomputing '94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. New York: ACM Press, 1994, pp. 528–537.

[55] V. Sarkar and S. J. Fink, "Efficient dependence analysis for java arrays." in *Euro-Par 2001: Proceedings of the 7th International European Conference on Parallel Processing*, 2001, pp. 273–277.

[56] P. Clauss and I. Tchoupaeva, "A symbolic approach to bernstein expansion for program analysis and optimization." in *CC 2004: the 13th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 2985. New York: Springer, 2004, pp. 120–133.

[57] R. A. van Engelen, J. Birch, Y. Shou, B. Walsh, and K. A. Gallivan, "A unified framework for nonlinear dependence testing and symbolic analysis," in *ICS '04: Proceedings of the 18th annual International Conference on Supercomputing*. New York: ACM Press, 2004, pp. 106–115.

[58] J. Hoeflinger and Y. Paek, "The access region test." in *LCPC 1999: the 12th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 1863. New York: Springer, 1999, pp. 271–285.

[59] P. C. Diniz, "Commutativity analysis: a new analysis technique for parallelizing compilers," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 6, pp. 942–991, 1997.

[60] P. Wu and D. A. Padua, "Beyond arrays - a container-centric approach for parallelization of real-world symbolic applications." in *LCPC 1998: the 11th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 1656. New York: Springer, 1998, pp. 197–212.

[61] P. Wu and D. A. Padua, "Containers on the parallelization of general-purpose java programs." in *1999 International Conference on Parallel Architectures and Compilation Techniques (PACT 1999)*, 1999, pp. 84–90.

[62] D. J. Quinlan, M. Schordan, Q. Yi, and B. R. de Supinski, "Semantic-driven parallelization of loops operating on user-defined containers." in *LCPC 2003: the 16th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 2958. New York: Springer, 2003, pp. 524–538.

[63] Q. Yi and D. J. Quinlan, "Applying loop optimizations to object-oriented abstractions through general classification of array semantics." in *LCPC 2004: the 17th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 3602. New York: Springer, 2004, pp. 253–267.

[64] Y. Lin and D. A. Padua, "Demand-driven interprocedural array property analysis." in *LCPC 1999: the 12th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 1863. New York: Springer, 1999, pp. 303–317.

[65] Y. Song and X. Kong, "Index-association based dependence analysis and its application in automatic parallelization." in *LCPC 2003: the 16th International*

Workshop on Languages and Compilers for High Performance Computing, ser. Lecture Notes in Computer Science, vol. 2958. New York: Springer, 2003, pp. 226–240.

[66] M. Gupta, S. Mukhopadhyay, and N. Sinha, "Automatic parallelization of recursive procedures." in *1999 International Conference on Parallel Architectures and Compilation Techniques (PACT 1999)*, 1999, pp. 139–148.

[67] P. Wu, A. Cohen, and D. A. Padua, "Induction variable analysis without idiom recognition: Beyond monotonicity." in *LCPC 2001: the 14th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 2624. New York: Springer, 2001, pp. 427–441.

[68] P. Wu, A. Cohen, J. Hoeflinger, and D. Padua, "Monotonic evolution: an alternative to induction variable substitution for dependence analysis," in *ICS '01: Proceedings of the 15th International Conference on Supercomputing*. New York: ACM Press, 2001, pp. 78–91.

[69] M. Arenaz, J. Touriño, and R. Doallo, "Towards detection of coarse-grain loop-level parallelism in irregular computations." in *Euro-Par 2002: Proceedings of the 8th International European Conference on Parallel Processing*, 2002, pp. 289–298.

[70] R. Ghiya, L. J. Hendren, and Y. Zhu, "Detecting parallelism in c programs with recursive darta structures." in *CC 1998: the 7th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 1383. New York: Springer, 1998, pp. 159–173.

[71] R. Wilhelm, S. Sagiv, and T. W. Reps, "Shape analysis." in *CC 2000: the 9th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 1781. New York: Springer, 2000, pp. 1–17.

[72] N. Rinetzky and S. Sagiv, "Interprocedural shape analysis for recursive programs." in *CC 2001: the 10th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 2027. New York: Springer, 2001, pp. 133–149.

[73] F. Corbera, R. Asenjo, and E. L. Zapata, "New shape analysis and interprocedural techniques for automatic parallelization of c codes." *International Journal of Parallel Programming*, vol. 30, no. 1, pp. 37–63, 2002.

[74] Y. Lin and D. A. Padua, "Analysis of irregular single-indexed array accesses and its applications in compiler optimizations." in *CC 2000: the 9th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 1781. New York: Springer, 2000, pp. 202–218.

[75] Y. Ben-Asher and G. Haber, "Parallel solutions of simple indexed recurrence equations." *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 1, pp. 22–37, 2001.

[76] D. Barthou, P. Feautrier, and X. Redon, "On the equivalence of two systems of affine recurrence equations (research note)." in *Euro-Par 2002: Proceedings of the 8th International European Conference on Parallel Processing*, 2002, pp. 309–313.

[77] M. Arenaz, J. T. no, and R. Doallo, "A gsa-based compiler infrastructure to extract parallelism from complex loops," in *ICS '03: Proceedings of the 17th annual International Conference on Supercomputing*. New York: ACM Press, 2003, pp. 193–204.

[78] L. Li and C. Verbrugge, "A practical mhp information analysis for concurrent java programs." in *LCPC 2004: the 17th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 3602. New York: Springer, 2004, pp. 348–362.

[79] P. Petersen and D. A. Padua, "Static and dynamic evaluation of data dependence analysis techniques." *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 11, pp. 1121–1132, 1996.

[80] W. Pugh and D. Wonnacott, "Constraint-based array dependence analysis," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 3, pp. 635–678, 1998.

[81] D. Niedzielski and K. Psarris, "An analytical comparison of the i-test and omega test." in *LCPC 1999: the 12th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 1863. New York: Springer, 1999, pp. 251–270.

[82] K. Psarris and K. Kyriakopoulos, "Data dependence testing in practice." in *1999 International Conference on Parallel Architectures and Compilation Techniques (PACT 1999)*, 1999, pp. 264–273.

[83] K. Psarris and K. Kyriakopoulos, "The impact of data dependence analysis on compilation and program parallelization," in *ICS '03: Proceedings of the 17th annual International Conference on Supercomputing.* New York: ACM Press, 2003, pp. 205–214.

[84] C. Ancourt and F. Irigoin, "Scanning polyhedra with do loops," in *PPOPP '91: Proceedings of the third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* New York: ACM Press, 1991, pp. 39–50.

[85] G. Balakrishnan and T. W. Reps, "Analyzing memory accesses in x86 executables." in *CC 2004: the 13th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 2985. New York: Springer, 2004, pp. 5–23.

[86] B. Franke and M. F. P. O'Boyle, "Compiler transformation of pointers to explicit array accesses in dsp applications." in *CC 2001: the 10th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 2027. New York: Springer, 2001, pp. 69–85.

[87] Z. Ammarguellat and I. W. L. Harrison, "Automatic recognition of induction variables and recurrence relations by abstract interpretation," in *PLDI '90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*. New York: ACM Press, 1990, pp. 283–295.

[88] M. Wolfe, "Beyond induction variables," in *PLDI '92: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. New York: ACM Press, 1992, pp. 162–174.

[89] M. P. Gerlek, E. Stoltz, and M. Wolfe, "Beyond induction variables: detecting and classifying sequences using a demand-driven SSA form," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 1, pp. 85–122, 1995.

[90] R. van Engelen, "Efficient symbolic analysis for optimizing compilers." in *CC 2001: the 10th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 2027. New York: Springer, 2001, pp. 118–132.

[91] M. Gupta, S. Mukhopadhyay, and N. Sinha, "Automatic parallelization of recursive procedures." *International Journal of Parallel Programming*, vol. 28, no. 6, pp. 537–562, 2000.

[92] W. Blume and R. Eigenmann, "Demand-driven, symbolic range propagation." in *LCPC 1995: the 8th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 1033. New York: Springer, 1995, pp. 141–160.

[93] P. Tu and D. Padua, "Gated SSA-based demand-driven symbolic analysis for parallelizing compilers," in *ICS '95: Proceedings of the 9th International Conference on Supercomputing*. New York: ACM Press, 1995, pp. 414–423.

[94] Y. Wu, "Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching," in *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. New York: ACM Press, 2002, pp. 210–221.

[95] H. Yu and L. Rauchwerger, "Adaptive reduction parallelization techniques," in *ICS '00: Proceedings of the 14th International Conference on Supercomputing*. New York: ACM Press, 2000, pp. 66–77.

[96] A. L. Fisher and A. M. Ghuloum, "Parallelizing complex scans and reductions," in *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. New York: ACM Press, 1994, pp. 135–146.

[97] D. Callahan, J. Dongarra, and D. Levine, "Vectorizing compilers: a test suite and results," in *Supercomputing '88: Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 98–105.

[98] K. Kennedy and K. S. McKinley, "Loop distribution with arbitrary control flow," in *Supercomputing '90: Proceedings of the 1990 ACM/IEEE Conference*

on Supercomputing. Washington, DC, USA: IEEE Computer Society, 1990, pp. 407–416.

[99] L. L. Smith, "Vectorizing c compilers: how good are they?" in *Supercomputing '91: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. New York: ACM Press, 1991, pp. 544–553.

[100] M. W. Hall, K. Kennedy, and K. S. McKinley, "Interprocedural transformations for parallel code generation," in *Supercomputing '91: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. New York: ACM Press, 1991, pp. 424–434.

[101] W. F. Appelbe and K. Smith, "Determining transformation sequences for loop parallelization." in *LCPC 1992: the 5th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 757. New York: Springer, 1992, pp. 208–222.

[102] Z. Li, "Array privatization for parallel execution of loops," in *ICS '92: Proceedings of the 6th International Conference on Supercomputing*. New York: ACM Press, 1992, pp. 313–322.

[103] K. Kennedy and K. S. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution." in *LCPC 1993: the 6th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 768. New York: Springer, 1993, pp. 301–320.

[104] K. S. McKinley, "Evaluating automatic parallelization for efficient execution on shared-memory multiprocessors," in *ICS '94: Proceedings of the 8th Inter-*

national Conference on Supercomputing. New York: ACM Press, 1994, pp. 54–63.

[105] J. T. Oplinger, D. L. Heine, and M. S. Lam, "In search of speculative thread-level parallelism." in *1999 International Conference on Parallel Architectures and Compilation Techniques (PACT 1999)*, 1999, pp. 303–313.

[106] A. Bhowmik and M. Franklin, "A general compiler framework for speculative multithreaded processors." *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 8, pp. 713–724, 2004.

[107] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen, "Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York: ACM Press, 2005, pp. 269–279.

[108] P. Feautrier, "Array expansion," in *ICS '88: Proceedings of the 2nd International Conference on Supercomputing*. New York: ACM Press, 1988, pp. 429–441.

[109] P. Tu and D. A. Padua, "Automatic array privatization." in *LCPC 1993: the 6th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 768. New York: Springer, 1993, pp. 500–521.

[110] J. Janssen and H. Corporaal, "Controlled node splitting." in *CC 1996: the 6th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 1060. New York: Springer, 1996, pp. 44–58.

[111] P.-Y. Calland, A. Darte, Y. Robert, and F. Vivien, "On the removal of anti- and output-dependences." *International Journal of Parallel Programming*, vol. 26, no. 2, pp. 285–312, 1998.

[112] D. Barthou, A. Cohen, and J.-F. Collard, "Maximal static expansion." *International Journal of Parallel Programming*, vol. 28, no. 3, pp. 213–243, 2000.

[113] R. Eigenmann, J. Hoeflinger, Z. Li, and D. A. Padua, "Experience in the automatic parallelization of four perfect-benchmark programs." in *LCPC 1991: the 4th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 589. New York: Springer, 1991, pp. 65–83.

[114] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. A. Padua, P. Petersen, W. M. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford, "Polaris: Improving the effectiveness of parallelizing compilers." in *LCPC 1994: the 7th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 892. New York: Springer, 1994, pp. 141–154.

[115] M. H. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam, "Detecting coarse-grain parallelism using an interprocedural parallelizing compiler," in *Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM)*. New York: ACM Press, 1995, p. 49.

[116] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam, "Interprocedural parallelization analysis in suif," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 4, pp. 662–731, 2005.

[117] S. Krishnan and L. V. Kale, "Automating parallel runtime optimizations using post-mortem analysis," in *ICS '96: Proceedings of the 10th International Conference on Supercomputing*. New York: ACM Press, 1996, pp. 221–228.

[118] E. Torrie, M. Martonosi, C.-W. Tseng, and M. W. Hall, "Characterizing the memory behavior of compiler-parallelized applications." *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 12, pp. 1224–1237, 1996.

[119] M. T. Kandemir, J. Ramanujam, and A. N. Choudhary, "Compiler algorithms for optimizing locality and parallelism on shared and distributed memory machines." in *1997 International Conference on Parallel Architectures and Compilation Techniques (PACT 1997)*, 1997, pp. 236–.

[120] Y. Paek, A. G. Navarro, E. L. Zapata, and D. A. Padua, "Parallelization of benchmarks for scalable shared-memory multiprocessors." in *1998 International Conference on Parallel Architectures and Compilation Techniques (PACT 1998)*, 1998, pp. 401–.

[121] R. Eigenmann, J. Hoeflinger, and D. A. Padua, "On the automatic parallelization of the perfect benchmarks." *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 1, pp. 5–23, 1998.

[122] N. Mukherjee and J. R. Gurd, "A comparative analysis of four parallelisation schemes," in *ICS '99: Proceedings of the 13th International Conference on Supercomputing*. New York: ACM Press, 1999, pp. 278–285.

[123] A. G. Navarro, E. L. Zapata, and D. A. Padua, "Compiler techniques for the distribution of data and computation." *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 6, pp. 545–562, 2003.

[124] M. T. Kandemir, A. N. Choudhary, J. Ramanujam, and P. Banerjee, "Reducing false sharing and improving spatial locality in a unified compilation framework." *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 4, pp. 337–354, 2003.

[125] G. Zhang, P. Unnikrishnan, and J. Ren, "Experiments with auto-parallelizing spec2000fp benchmarks." in *LCPC 2004: the 17th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 3602. New York: Springer, 2004, pp. 348–362.

[126] K. Ishizaka, T. Miyamoto, J. Shirako, M. Obata, K. Kimura, and H. Kasahara, "Performance of oscar multigrain parallelizing compiler on smp servers." in *LCPC 2004: the 17th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 3602. New York: Springer, 2004, pp. 319–331.

[127] B. Franke and M. F. P. O'Boyle, "A complete compiler approach to auto-parallelizing c programs for multi-dsp systems." *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 3, pp. 234–245, 2005.

[128] L. Rauchwerger and D. Padua, "The privatizing doall test: a run-time technique for doall loop identification and array privatization," in *ICS '94: Proceedings of the 8th International Conference on Supercomputing.* New York: ACM Press, 1994, pp. 33–43.

[129] L. Rauchwerger and D. Padua, "The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization," in *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation.* New York: ACM Press, 1995, pp. 218–232.

[130] D. Patel and L. Rauchwerger, "Principles of speculative run-time parallelization." in *LCPC 1998: the 11th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 1656. New York: Springer, 1998, pp. 323–337.

[131] B. So, S. Moon, and M. W. Hall, "Measuring the effectiveness of automatic parallelization in suif," in *ICS '98: Proceedings of the 12th International Conference on Supercomputing*. New York: ACM Press, 1998, pp. 212–219.

[132] D. Patel and L. Rauchwerger, "Implementation issues of loop-level speculative run-time parallelization." in *CC 1999: the 8th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 1575. New York: Springer, 1999, pp. 183–197.

[133] H. Yu and L. Rauchwerger, "Techniques for reducing the overhead of run-time parallelization." in *CC 2000: the 9th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 1781. New York: Springer, 2000, pp. 232–248.

[134] T. Chen, J. Lin, X. Dai, W.-C. Hsu, and P.-C. Yew, "Data dependence profiling for speculative optimizations." in *CC 2004: the 13th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 2985. New York: Springer, 2004, pp. 57–72.

[135] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar, "Min-cut program decomposition for thread-level speculation," in *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. New York: ACM Press, 2004, pp. 59–70.

[136] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai, "A cost-driven compilation framework for speculative parallelization of sequential programs," in *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. New York: ACM Press, 2004, pp. 71–81.

[137] G. C. Necula, "Proof-carrying code," in *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York: ACM Press, 1997, pp. 106–119.

[138] T. V. N. Nguyen, F. Irigoin, C. Ancourt, and F. Coelho, "Automatic detection of uninitialized variables." in *CC 2003: the 12th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 2622. New York: Springer, 2003, pp. 217–231.

[139] W. Pugh and D. Wonnacott, "Nonlinear array dependence analysis," in *LCR '95: Proceedings of the third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, 1995.

[140] J.-F. Collard, "Array SSA for explicitly parallel programs." in *Proceedings of the International Euro-Par Conference*, 1999, pp. 383–390.

[141] J. H. Saltz, R. Mirchandaney, and K. Crowley, "Run-time parallelization and scheduling of loops," *IEEE Transactions on Computers*, vol. 40, no. 5, pp. 603–612, May 1991.

[142] R. Ponnusamy, J. Saltz, and A. Choudhary, "Runtime compilation techniques for data partitioning and communication schedule reuse," in *Supercomputing '93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. New York: ACM Press, 1993, pp. 361–370.

[143] S.-T. Leung and J. Zahorjan, "Improving the performance of runtime parallelization," in *PPOPP '93: Proceedings of the fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* New York: ACM Press, 1993, pp. 83–91.

[144] S. D. Sharma, R. Ponnusamy, B. Moon, Y. S. Hwang, R. Das, and J. Saltz, "Run-time and compile-time support for adaptive irregular problems," in *Supercomputing '94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing.* New York: ACM Press, 1994, pp. 97–106.

[145] D. K. Chen, J. Torrellas, and P. C. Yew, "An efficient algorithm for the run-time parallelization of doacross loops," in *Supercomputing '94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing.* New York: ACM Press, 1994, pp. 518–527.

[146] R. Das, P. Havlak, J. Saltz, and K. Kennedy, "Index array flattening through program transformation," in *Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM).* New York: ACM Press, 1995, p. 70.

[147] G. Agrawal, A. Sussman, and J. Saltz, "Compiler and runtime support for structured and block structured applications," in *Supercomputing '93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing.* New York: ACM Press, 1993, pp. 578–587.

[148] G. Agrawal, J. Saltz, and R. Das, "Interprocedural partial redundancy elimination and its application to distributed memory compilation," in *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation.* New York: ACM Press, 1995, pp. 258–269.

[149] G. Agrawal and J. Saltz, "Interprocedural compilation of irregular applications for distributed memory machines," in *Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM)*. New York: ACM Press, 1995, p. 48.

[150] G. Agrawal, A. Acharya, and J. Saltz, "An interprocedural framework for placement of asynchronous i/o operations," in *ICS '96: Proceedings of the 10th International Conference on Supercomputing*. New York: ACM Press, 1996, pp. 358–365.

[151] G. Agrawal, "Interprocedural partial redundancy elimination with application to distributed memory compilation." *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 7, pp. 609–625, 1998.

[152] G. Agrawal, "A general interprocedural framework for placement of split-phase large latency operations." *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 4, pp. 394–413, 1999.

[153] I. H. Kazi and D. J. Lilja, "Coarse-grained thread pipelining: A speculative parallel execution model for shared-memory multiprocessors." *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 9, pp. 952–966, 2001.

[154] L. Rauchwerger, N. M. Amato, and D. A. Padua, "Run-time methods for parallelizing partially parallel loops," in *ICS '95: Proceedings of the 9th International Conference on Supercomputing*. New York: ACM Press, 1995, pp. 137–146.

[155] F. H. Dang, H. Yu, and L. Rauchwerger, "The R-LRPD test: Speculative parallelization of partially parallel loops." in *16th International Parallel and Distributed Processing Symposium (IPDPS 2002), CD-ROM/Abstracts Proceedings*, 2002.

[156] A. Bhowmik and M. Franklin, "A fast approximate interprocedural analysis for speculative multithreading compilers," in *ICS '03: Proceedings of the 17th annual International Conference on Supercomputing.* New York: ACM Press, 2003, pp. 32–41.

[157] M. H. Cintra and D. R. L. Ferraris, "Design space exploration of a software speculative parallelization scheme." *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 6, pp. 562–576, 2005.

[158] Y. Ding and Z. Li, "An adaptive scheme for dynamic parallelization." in *LCPC 2001: the 14th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 2624. New York: Springer, 2001, pp. 274–289.

[159] W. Pugh, "The Omega test: A fast and practical integer programming algorithm for dependence analysis," in *Supercomputing '91*, Albuquerque, N.M., Nov. 1991, pp. 4–13.

[160] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam, "Maximizing multiprocessor performance with the suif compiler," *IEEE Computer*, vol. 29, no. 12, pp. 84–89, December 1996.

[161] J. Hoeflinger, "Interprocedural Parallelization Using Memory Classification Analysis," Ph.D. dissertation, University of Illinois, Urbana-Champaign, August, 1998.

[162] B. Creusillet and F. Irigoin, "Exact versus approximate array region analyses." in *LCPC 1996: the 9th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 1239. New York: Springer, 1996, pp. 86–100.

[163] V. Sarkar and K. Knobe, "Enabling sparse constant propagation of array elements via array ssa form." in *Static Analysis Symposium*, 1998, pp. 33–56.

[164] U. Banerjee, *Dependence Analysis for Supercomputing.* Norwell, Mass.: Kluwer Academic Publishers, 1988.

[165] X. Kong, D. Klappholz, and K. Psarris, "The I test: An improved dependence test for automatic parallelization and vectorization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 342–349, July 1991.

[166] P. Wu, A. Cohen, and D. Padua, "Induction variable analysis without idiom recognition: Beyond monotonicity," in *2001 Workshop on Languages and Compilers for Parallel Computing*, Cumberland Falls, KY, 2001, pp. 427–441.

[167] S. Rus, D. Zhang, and L. Rauchwerger, "The value evolution graph and its use in memory reference analysis," in *13th Conference on Parallel Architecture and Compilation Techniques.* Washington, DC: IEEE Computer Society, 2004, pp. 243–254.

[168] D. kai Chen, J. Torrellas, and P.-C. Yew, "An Efficient Algorithm for the Run-time Parallelization of DOACROSS Loops," *To appear in Proceedings for Supercomputing '94, Washington D.C., November 14-18, 1994*, October 1994.

[169] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen, "Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation.* New York, NY, USA: ACM Press, 2005, pp. 269–279.

[170] M. J. Wolfe, *High Performance Compilers for Parallel Computing*, C. Shanklin

and L. Ortega, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[171] S. Moon, B. So, M. W. Hall, and B. R. Murphy, "A case for combining compile-time and run-time parallelization," in *LCR '98: Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*. London, UK: Springer-Verlag, 1998, pp. 91–106.

[172] H. Yu and L. Rauchwerger, "Run-time parallelization overhead reduction techniques," in *Proc. of the 9th International Conference on Compiler Construction (CC2000), Berlin, Germany*. Lecture Notes in Computer Science, Springer-Verlag, March 2000.

[173] R. Triolet, F. Irigoin, and P. Feautrier, "Direct parallelization of Call statements," in *ACM SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, Calif., June 1986, pp. 175–185.

[174] J. Gu, Z. Li, and G. Lee, "Symbolic array dataflow analysis for array privatization and program parallelization," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*. ACM Press, 1995, p. 47.

[175] J. R. Reif and H. R. Lewis, "Symbolic evaluation and the global value graph," in *4th ACM Symposium on Principles of Programming Languages*, 1977, pp. 104–118.

[176] W. Blume and R. Eigenmann, "Symbolic Range Propagation," Univ of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, Tech. Rep. 1381, October 1994.

[177] J. JàJà, *An Introduction Parallel Algorithms.* Reading, Massachusetts: Addison–Wesley, 1992.

[178] Y. Lin and D. A. Padua, "On the automatic parallelization of sparse and irregular fortran programs," in *Proceedings of the 1998 Workshop on Languages, Compilers, and Run-TimeSystems for Scalable Computers (LCR98)*, 1998, pp. 41–56.

[179] L. Rauchwerger and D. A. Padua, "Parallelizing WHILE Loops for Multiprocessor Systems," in *Proceedings of 9th International Parallel Processing Symposium*, April 1995.

[180] M. Spezialetti and R. Gupta, "Loop monotonic statements," *IEEE Transactions on Software Engineering*, vol. 21, no. 6, pp. 497–505, 1995.

[181] S.-C. Chen and D. J. Kuck, "Time and parallel processor bounds for linear recurrence systems," *IEEE Transactions on Computers*, vol. 24, no. 7, pp. 701–717, 1975.

[182] D. Callahan, "Recognizing and parallelizing bounded recurrences," in *1991 Workshop on Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, no. 589. Santa Clara, Calif.: Berlin: Springer Verlag, Aug. 1991, pp. 169–185.

[183] W. Pottenger and R. Eigenmann, "Parallelization in the presence of generalized induction and reduction variables," Univ. of Illinois at UrbanaChampaign, Center for Supercomp. R&D, Tech. Rep. 1396, January 1995.

[184] A. M. Ghuloum and A. L. Fisher, "Flattening and parallelizing irregular, recurrent loop nests," in *Proceedings of the fifth ACM SIGPLAN symposium on*

Principles and practice of parallel programming. Santa Barbara, CA: ACM Press, 1995, pp. 58–67.

[185] R. Gupta, "Optimizing array bound checks using flow analysis," *ACM Letters on Programming Languages and Systems*, vol. 2, no. 1–4, pp. 135–150, 1993.

[186] H. Yu, "Run-time optimization of adaptive irregular applications," Ph.D. dissertation, Texas A&M University, College Station, TX, 2004.

[187] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering." *Softw., Pract. Exper.*, vol. 30, no. 11, pp. 1203–1233, 2000.

[188] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu, "Advanced Program Restructuring for High-Performance Computers with Polaris," *IEEE Computer*, vol. 29, no. 12, pp. 78–82, December 1996.

[189] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. 10, no. 4, pp. 352–357, July 1984.

[190] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millenium," *IEEE Computer*, vol. 33, no. 7, pp. 28–35, July 2000.

[191] S. Horwitz, P. Pfeiffer, and T. Reps, "Dependence analysis for pointer variables," in *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*. New York: ACM Press, 1989, pp. 28–40.

[192] R. P. Wilson and M. S. Lam, "Efficient context-sensitive pointer analysis for c programs," in *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference*

on Programming Language Design and Implementation. New York: ACM Press, 1995, pp. 1–12.

[193] M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural pointer alias analysis," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 4, pp. 848–894, 1999.

[194] R. Rugina and M. Rinard, "Symbolic bounds analysis of pointers, array indices, and accessed memory regions," in *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation.* New York: ACM Press, 2000, pp. 182–195.

[195] B. G. Ryder, W. A. Landi, P. A. Stocks, S. Zhang, and R. Altucher, "A schema for interprocedural modification side-effect analysis with pointer aliasing," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 2, pp. 105–186, 2001.

[196] P. Wu, P. Feautrier, D. Padua, and Z. Sura, "Instance-wise points-to analysis for loop-based dependence testing," in *ICS '02: Proceedings of the 16th International Conference on Supercomputing.* New York: ACM Press, 2002, pp. 262–273.

[197] J. Zhu and S. Calman, "Symbolic pointer analysis revisited," in *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation.* New York: ACM Press, 2004, pp. 145–157.

[198] S. Horwitz, "Precise flow-insensitive may-alias analysis is np-hard," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 1, pp. 1–6, 1997.

[199] C. Lapkowski and L. J. Hendren, "Extended SSA numbering: Introducing SSA properties to language with multi-level pointers." in *CC 1998: the 7th International Conference on Compiler Construction*, ser. Lecture Notes in Computer

Science, vol. 1383. New York: Springer, 1998, pp. 128–143.

[200] P.-S. Chen, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee, "Interprocedural probabilistic pointer analysis." *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 10, pp. 893–907, 2004.

[201] A. Venet and G. Brat, "Precise and efficient static array bound checking for large embedded c programs," in *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. New York: ACM Press, 2004, pp. 231–242.

[202] J. Zhu, J. Hoeflinger, and D. A. Padua, "Compiling for a hybrid programming model using the lmad representation." in *LCPC 2001: the 14th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 2624. New York: Springer, 2001, pp. 321–335.

[203] S. P. Amarasinghe and M. S. Lam, "Communication optimization and code generation for distributed memory machines," in *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. New York: ACM Press, 1993, pp. 126–138.

[204] J. M. Anderson and M. S. Lam, "Global optimizations for parallelism and locality on scalable parallel machines," in *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. New York: ACM Press, 1993, pp. 112–125.

[205] R. Ponnusamy, J. H. Saltz, A. N. Choudhary, Y.-S. Hwang, and G. Fox, "Runtime support and compilation methods for user-specified irregular data distributions." *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 8, pp. 815–831, 1995.

[206] M. Gupta and E. Schonberg, "Static analysis to reduce synchronization costs in data-parallel programs," in *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York: ACM Press, 1996, pp. 322–332.

[207] M. Gupta, E. Schonberg, and H. Srinivasan, "A unified framework for optimizing communication in data-parallel programs." *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 7, pp. 689–704, 1996.

[208] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy, "A global communication optimization technique based on data-flow analysis and linear algebra," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 6, pp. 1251–1297, 1999.

[209] Y. Ding and Z. Li, "A compiler analysis of interprocedural data communication," in *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 11.

[210] L. Choi and P.-C. Yew, "Interprocedural array data flow analysis for cache coherence." in *LCPC 1995: the 8th International Workshop on Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 1033. New York: Springer, 1995, pp. 81–95.

[211] S. Moon, B. So, and M. W. Hall, "Evaluating automatic parallelization in suif." *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 1, pp. 36–49, 2000.

[212] Y. Paek, A. G. Navarro, E. L. Zapata, J. Hoeflinger, and D. A. Padua, "An advanced compiler framework for non-cache-coherent multiprocessors." *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 241–259, 2002.

[213] R. Wismüller, "Debugging of globally optimized programs using data flow analysis," in *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. New York: ACM Press, 1994, pp. 278–289.

[214] P. P. Pineo and M. L. Soffa, "A practical approach to the symbolic debugging of parallelized code." in *CC 1994: the 5th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 786. New York: Springer, 1994, pp. 339–356.

[215] R. Narasimhan, D. J. Rosenkrantz, and S. S. Ravi, "Using data flow information to obtain efficient check sets for algorithm-based fault tolerance." *International Journal of Parallel Programming*, vol. 27, no. 4, pp. 289–323, 1999.

[216] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira, "Automatic loop transformations and parallelization for java," in *ICS '00: Proceedings of the 14th International Conference on Supercomputing*. New York: ACM Press, 2000, pp. 1–10.

[217] F. Qian, L. J. Hendren, and C. Verbrugge, "A comprehensive approach to array bounds check elimination for java." in *CC 2002: the 11th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 2304. New York: Springer, 2002, pp. 325–342.

[218] V. M. DeLaLuz, I. Kadayif, M. T. Kandemir, and U. Sezer, "Access pattern restructuring for memory energy." *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 4, pp. 289–303, 2004.

APPENDIX A

USER MANUAL

Overview

The automatic parallelizer is run as:

```
polaris -f switches-file sequential.f | list2src > parallel.f
```

The result is then passed to an OpenMP compiler. Using Intel's *ifort*, the command line is:

```
ifort -openmp parallel.f -o parallel.x
```

The code can then be run in parallel:

```
setenv OMP_NUM_THREADS 4
time ./parallel.x
```

Configuration Files

Polaris will look for two files in the current directory. *ipa_framework.routines* must contain a list of subprogram names, one on each line, in capitals. These routines (and the slices of the call graph below them) are the only parts of the program that get analyzed. If the file is not present, the whole program will be analyzed. *ipa_framework.loops* must contain loop names (Polaris convention), one on each line, followed by a blank space and then a numeric value. The number can be the sum of any subset of the following flags:

- 1: analyze this loop
- 2: generate parallel code (and associated run time tests if necessary)

- 4: insert timing instrumentation around this loop
- 8: generate an HTML report (only compile time reports are generated at this time although there is capability for run time reports in HTML as well)

If *ipa_framework.loops* is missing, all the loops will be analyzed but (at this time) no parallelization is performed (the equivalent of listing all loops with flag 1).

Compiler Switches

Here is the list of compiler switches required to run the automatic parallelizer:

- *hybrid_analysis*: this is the master switch. 0 = off, 1 = on.
- *ha_parallelization*: 0 = no parallelization, 1 = hybrid, 2 = static only.
- *ha_scope*: 1 = limit the analysis to the program slice below the loops listed in file *ipa_framework.loops*. If the file does not exist, still limit the analysis to the parts of the program that are in some loop (possibly interprocedurally).
- *ha_debug*: 0 = off, 1 = on.
- *ha_redisplay_call_graph_delay*: the number of seconds after which the analysis progress report is updated. The report is in the form of a PostScript file named *IPA_framework.progress.ps*. This file can be loaded in *gv* and monitored using the *watch file* setting. It displays the nodes of the call graphs using four colors:
  - black: will not be analyzed
  - blue: will be analyzed
  - red: is currently being analyzed
  - maroon: was analyzed

A node is colored even if it analyzed only partially.

- *rt_lmad_display_flags*: a sum of any subset of the following flags that control how a USR is displayed.
  - 1=meaning: some human readable meaning as a string that can be attached by the programmer. It is currently used to mark important nodes such as $RO \cap WF$ when displaying dependence equations.
  - 2=registers: the number of the USR 'register' numbers used when generating code to evaluate them at run time.
  - 4=sources: the sets of statements that the USRs were extracted from. This is not supported anymore.
  - 8=estimates: the overestimate and the underestimate as lists of LMADs.
  - 16=input: the input sensitivity set of the USR.
  - 32=referred: the list of variables referenced by the USR.
  - 64=reference counter: the reference counter of the USR nodes.
  - 128=address: the unique USR identification number.
  - 256=size: the size of the USR in bytes (not supported anymore).
  - 512=enclosing dimensions: in case the USR is completely enclosed in a subspace such as a line or plane, the coordinates of that plane.
  - 1024=detailed descriptors: do not abbreviate (by default there is a maximum character length for any description in order to make it look better when displayed as a graph).

```
      Program test
      Integer a(1000), n, i
      Read *, n
      Do 10 i = 1, 100
10       a(i) = 3*a(100+i)
      Do 20 i = 1, 100
20       a(i) = 2+a(i-1)
      Do 30 i = 1, n
30       a(i) = a(i)+2* a(i+100)
      Print *, a(n/2)
      End
```

(a)

```
      PROGRAM test
      INTEGER*4 a, i, n
      INTEGER*4 numprocs, test_do20_a_indep
      COMMON /test_do20_is_indep_indeps/
     *        test_do20_a_indep
      DIMENSION a(1000)

      READ (UNIT = *, FMT = *) n
      CALL pti_test_do30_a_indep_o_1(n)
!$OMP PARALLEL
CSRD LOOPLABEL 'TEST_do10'
!$OMP DO
      DO i = 1, 100, 1
        a(i) = 3*a(100+i)
      ENDDO
!$OMP END DO
!$OMP END PARALLEL
CSRD LOOPLABEL 'TEST_do20'
      DO i = 1, 100, 1
        a(i) = 2+a((-1)+i)
      ENDDO
***** continued from left column *****
      test_do30_indep = test_do30_a_indep
      IF (.NOT.test_do20_indep) THEN
        numprocs = omp_get_numthreads(1)
        CALL omp_set_numthreads(1)
      ENDIF
!$OMP PARALLEL
CSRD LOOPLABEL 'TEST_do30'
!$OMP DO
      DO i = 1, n, 1
        a(i) = a(i)+2*a(100+i)
      ENDDO
!$OMP END DO
!$OMP END PARALLEL
      IF (omp_get_numthreads().EQ.1) THEN
        CALL omp_set_numthreads(numprocs)
      ENDIF
      PRINT *, a(n/2)
      STOP
      END

      SUBROUTINE pti_test_do30_a_indep_o_1(n)
      INTEGER*4 n, test_do30_a_indep
      COMMON /test_do30_is_indep_indeps/ test_do30_a_indep
      test_do30_a_indep = (-100)+n.LE.0
      END
```

(b)

Fig. 78. Parallelization example. (a) Original sequential code. (b) After automatic parallelization.

Example

Fig. 78 presents an example with three loops. The first one, *TEST_do10* is found parallel at compile time. The second one, *TEST_do20* is found sequential at compile time. The third one needs a run time test.

Visualization of Parallelization Information

Fig. 79 presents the compile time report produced by the automatic parallelizer based on Hybrid Analysis. The report is accessible at location *./halog/html_files/ct_parallel_report*. The left frame shows loop level summaries, while the right frame shows details for the selected loop (in this case *SOLVH_do20*.

The underlined keywords contain links to symbol level details, either USRs such as the dependence test for array *XE* or PDAGs such as the solvers (ordered by complexity) for the same array *XE*.

The USR and PDAG graph representation require a *GraphViz dot* plugin. At this time, we simply convert the *.dot* files to *.ps* using GraphViz (`dot -Tps filename.dot -o filename.ps`), and configure the browser to use *gv* as an external viewer for PostScript files.

Some sample USRs and PDAGs are shown in Figs. 58, 60, 59 and 61. Many more examples are available by following link *Compile-time Diagnostics Table* on page

`http://parasol.tamu.edu/compilers/ha/`

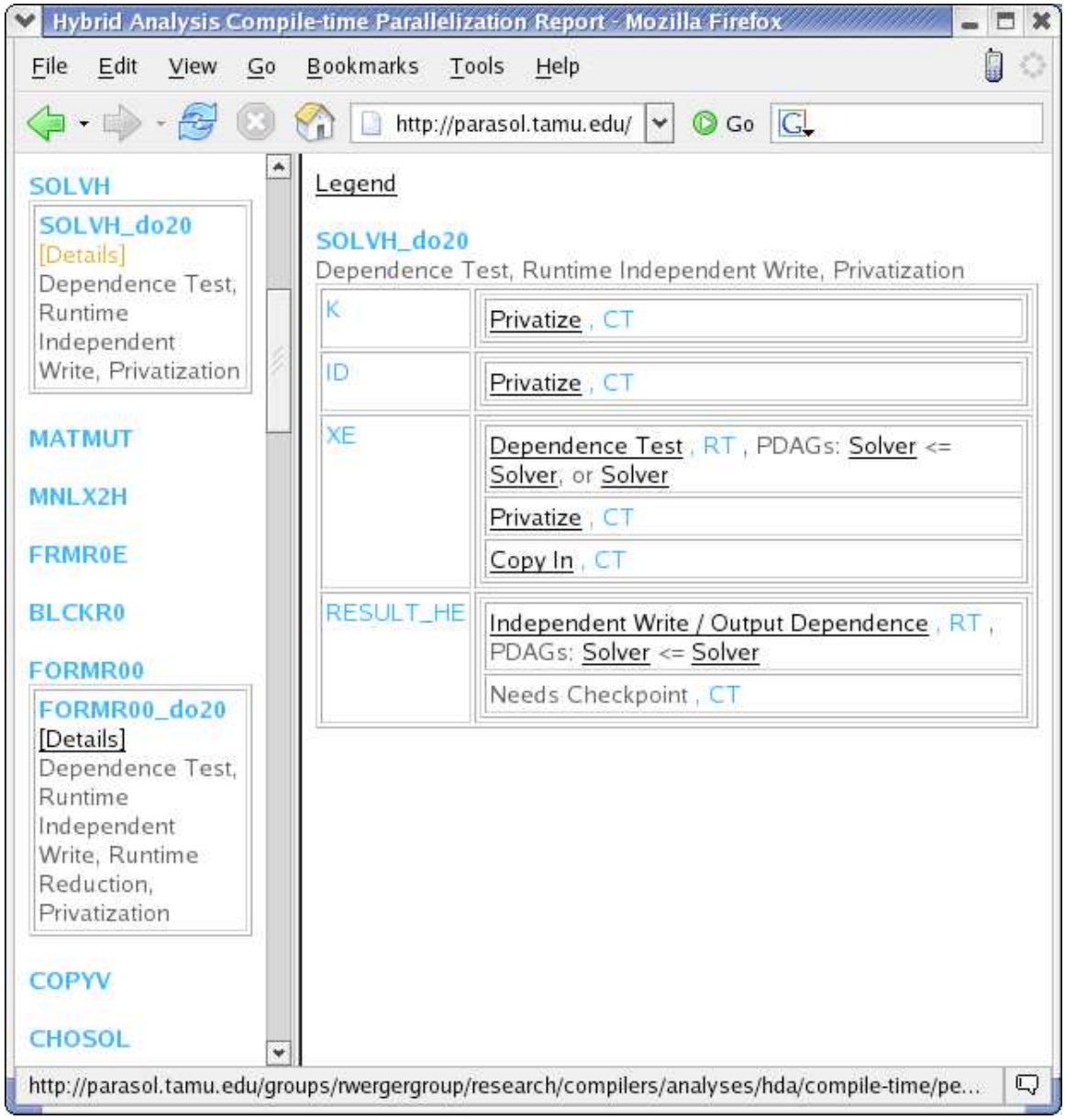


Fig. 79. Parallelization report after the compile time phase of Hybrid Analysis for DYFESM.

APPENDIX B

REFERENCE MANUAL

This appendix presents the implementation details at the level of a reference manual for compiler writers who develop code based on Hybrid Analysis.

Organization

The Polaris source code tree roots at directory *cvdl*. The code that implements Hybrid Analysis and automatic parallelization resides in three subdirectories:

- *ipa_framework*: a framework for interprocedural analysis. It contains generic program traversal algorithms that can is to implement Memory Classification Analysis.
- *rt_lmad*: the definitions of the USR and the PDAG including construction and manipulation routines.
- *ipa_rt_lmad*: the implementation of the MCA algorithms as well as automatic parallelization.

The symbolic value analysis code is in directory *base/Evolution*. Several filters are applied to make the code fit our program model. The filters can be found in subdirectory *filters*.

Interprocedural MCA using USRs

Memory Classification Analysis is implemented as a single pass over the program. Class *IPA_bu_program* manages a generic bottom-up traversal of the call graph. Class

IPA_bu_routine manages a generic bottom-up traversal of the CDG of a program unit. As of now, these routines expect the program to be structured, i.e., to contain no GOTO or RETURN statements. Such statements are removed by *filters*. In rare cases when they cannot be removed, all the code regions that may be affected are excluded from analysis.

The generic traversal mechanism requires the programmer to provide an implementation of class *INFO_base*. This class should provide a container for all the information produced by the analysis process. The generic traversal mechanism contains empty slots for *information* and *actions*. The first refers to the data computed by the traversal (such as MCA partitions) and the second one to the actions taken (such as dependence analysis).

Information

To implement MCA, we designed class *IPA_RT_LMAD_info*, which is derived from *INFO_base*. It is organized as an associative container (map), in which the key is a symbol and the value is a triplet (RO, WF, RW) of USRs. The USR is implemented by class *RT_LMAD*. An *IPA_RT_LMAD_info* object is associated with every node in the Control Dependence Graph and it contains the classified and aggregated memory references that take place at that node and in its children recursively.

Actions

In addition to collecting information, the generic traversal provides means to perform actions (decisions) at important points. The most important action (parallelism detection) takes place at loop level. Other actions are mostly for bookkeeping, such as freeing repositories associated with a program unit after processing all matching call sites. Actions are also used for debugging.

```
Algorithm run_one_block
  Input: StmtList as list of statements
  Output: Info as INFO_base
Info = ∅
For (Stmt in StmtList) Do
  LocalInfo = Call run_one_stmt(Stmt)
  Info.add_info(LocalInfo)
EndFor
End

Algorithm run_one_stmt
  Input: Stmt as Statement
  Output: Info as INFO_base
Switch (Stmt.stmt_class())
  Case AssignmentStmt:
    Info.get_info(RegStmt, Stmt)
  Case DoStmt:
    LocalInfo = ∅
    LocalInfo.get_info(RegStmt, Stmt)     // references to loop bounds and index
    LocalInfo = run_one_block(loop body) // references as function of index
    Info.get_info(DoStmt, LocalInfo)      // references across iteration space
  Case CallStmt:
    LocalInfo = ∅
    LocalInfo.get_info(RegStmt, Stmt)     // references in actual expressions
    LocalInfo = run_one_block(sub body)   // references as functions of formals
    Info.get_info(CallStmt, LocalInfo)    // references as functions of actuals
  ...
EndSwitch
End
```

Fig. 80. Interprocedural analysis framework as collection of information in a bottom-up traversal of the program. The first argument to the polymorphic method *get_info* selects the correct code to process the given information.

USR Class

For historical reasons, the USR is named RT_LMAD in the code. However, we will use the name USR in this description in order to keep the dissertation consistent. A USR object contains either a list of LMADs (implemented by the *AbstractAccess* class, or a symbolic representation of an operation on USRs. Translation and expansion operations have a single USR operand (named *left*), while union, intersection and set difference have two operands, named *left* and *right*.

Once created, a USR cannot be modified. If a modification is necessary, then the whole USR must be cloned and modified by a specialized method. This design

choice made it possible to overlay large parts of the trees in memory using reference counting and guarantees linear memory usage scalability.

There are a few important entities associated with USRs.

- *Referenced Symbols*: every USR references some symbols, for instance when loop bounds are symbolic names rather than integer constants. These symbols are important to know when moving a USR from one context to another such as when creating an inspector.

- *Input Values*: every USR has a set of variables that it is sensitive to, i.e. that determines its value (together with the operators it is made of). Input values are different from referenced symbols. For instance, the USR that describes the effect of a loop on an indirect access will reference the loop index. However, with respect to the context outside the loop, the only symbols that the USR is sensitive to are the indirection array and the ones present in the loop bounds (the loop index is not even defined outside the loop).

  Knowing this set precisely is crucial to decide which USRs are loop invariant (they are invariant if their sensitivity set is invariant). Some USRs depend on subscript arrays. In some cases, they depend only on a subregion of a subscript array. In order to represent this accurately, the *InputValues* class is represented as a set of pairs (variable name, location set), where location set the set of indices of the subscript array that the USR is sensitive to. The location sets are themselves represented as USRs. In order to avoid cycles and lengthy chains of dependences we restricted the recursion to 2, i.e., the USRs that describe the input values of another USR can only store input values as variables, and not pairs (variable, location set).

- *Estimates*: an overestimate and underestimate of the USR as a list of LMADs.

They are used in compile time comparisons as well as to extract run time tests.

- *Enclosing Dimensions*: the LMADs lose information about the dimension bounds present in array declarations. This information is crucial to prove independence based on the Fortran standard provision that forbids accessing outside the declared bounds. For each USR we maintain the set of subdimensions in which it is included, if such a subdimension (point, line, plane, hyperplane) exists.

The USRs provide several operations:

- Composition. All these operations take as operand a USR or two and result into another USR. They are highly optimized to keep the resulting USRs as simple as possible. When expanding $\{i\}$ over iteration space $i = 1, n$, rather than creating an operator node and returning $\otimes^{\cup}_{i=1,n}\{i\}$, we instead return $[1 : n]$.
  - Union
  - Intersection
  - Set difference
  - Expansion over an iteration space
  - Predication
  - Translation from a called subprogram to the caller at a call site
- Comparison
  - Equality: tests whether two USRs represent the same set of locations.
  - Inclusion: tests whether a USR is included in another. It is a powerful recursive algorithm based on set algebra properties as well as on approximations (overestimates and underestimates).

MCA: Putting it All Together

MCA is started when the Polaris driver invokes function *ipa_rt_lmad* defined in file *ipa_rt_lmad/ipa_rt_lmad.cc*. This function builds an object of type *IPA_bu_program* and then invokes its *run()* method. This method will build *IPA_bu_routine* objects which will run their *run_one_block()* method described in Fig. 80.

Automatic Parallelization

Whenever the bottom up traversal of the program arrives at a loop header, an *ACTION_base::after_loop()* action gets triggered. We have programmed this action in *IPA_RT_LMAD_actions* to perform static data dependence analysis, generate run time test if necessary and parallelize the loop if possible.

Dependence Analysis

The dependence sets such as $RO \cap WF$ are built in function *symbol_parallel_info()*. The analysis of the dependence sets takes place in function *symbol_parallel_diagnostic()*. Here is a description of the information collected for each symbol (class *SymbolCGPI*):

- `bool is_pushback`: true if this symbol is a pushback array, false otherwise.
- `const RT_LMAD* pushback_d`: pushback footprint (only defined if is_pushback is true).
- `bool is_pushback_ptr`: if true, this symbol is the stack top for some pushback.
- `const RT_LMAD* dependence_d`: there are dependences on these locations that could not be eliminated.
- `Solvers* dependence_solvers`: PDAGs for this symbol's runtime dependence tests.

- `IPA_RUN_TIME_INFO dependence_t`: dependence flag: Yes = dependent, No = independent, Unknown = runtime test.
- `const RT_LMAD* reduction_d`: reduction pattern on these locations.
- `Solvers* reduction_solvers`: runtime test to decide whether this is a reduction or just an independent update.
- `IPA_RUN_TIME_INFO reduction_t`: Yes = reduction, No = independent update, Unknown = runtime test.
- `REDUCTION_OP reduction_op`: reduction operator (+, *, MAX, MIN etc).
- `const RT_LMAD* output_d`: there are output dependences on these locations.
- `Solvers* output_solvers`: runtime test to decide whether this is an independent write or it is dependent and needs last value assignment computation.
- `IPA_RUN_TIME_INFO output_t`: Yes = needs last value, No = independent write, Unknown = runtime test necessary.
- `const RT_LMAD* privatize_d`: These locations must be privatized. Some dependences may have been removed based on this assumption.
- `const RT_LMAD* copy_in_d`: these locations must be copied in.
- `LV_TYPE last_value_t`: the type of last value assignemnt (none, static, dynamic).
- `bool needs_ckpt`: this object needs to be backed up before speculative execution.

PDAG Class

The PDAG class implements the concept of runtime test. Internally, a PDAG is:

- A Polaris expression such as $n \leq 200$ or $false$.
- A tuple $recurrence(i), op, PDAG(i)$, where op is $\vee$ or $\wedge$.
- A tuple $op, list\ of\ PDAGs$, where op is $\vee$ or $\wedge$.
- A tuple $recurrence(i), interval(i)$, where $interval(i)$ are intervals that must be proved disjoint.
- An unsolved equation $USR = \emptyset$, for which a USR evaluation test or an LRPD test will be generated.

PDAGs are built from equations such as $dependence\_d = \emptyset$ by calling function *solve()*, defined in file *rt_lmad/solver.cc*.

Generation of Run Time Test Code

The generation of run time tests (the Fortran translation of the PDAG) is performed within the *IPA_RT_LMAD_actions::after_program()* method. It needs to be after the whole analysis process, but while the program is still in SSA. The code to evaluate a PDAG is generated either as an inspector, in a function named *pti_LoopName_VarName_TestName_Complexity*, or in the loop body for speculative execution. In either case, they will build a variable named *LoopName_VarName_TestName*.

PDAGs are translated into Fortran code based on an attribute grammar. Logical expressions are inserted verbatim. Logical operations over an iteration space are implemented as parallel loops using a reduction operator (*.AND.* or *.OR.*) on the accumulator. Interval disjointness tests are implemented as calls to the run time

library, although a small stub is generated inline to store the interval bounds in an array (which is passed to the run time library). USR evaluation code is generated inline down to the level of USR operations, which are implemented as calls to the run time library.

Inspectors are generated using an in-house interprocedural forward slicer.

At this point, LRPD tests are generated in a later phase because their generation library expects the code to not be in SSA format. However, they are connected to the PDAG tests through control variables and control flow constructs.

Generation of Parallel Code

The parallel code generation takes place after the whole program is analyzed and the tests are generated. Parallelization information is communicated to the *parallel back end*. This module generates parallelization directives which will be interpreted by the parallel machine vendor compiler (at this point only OpenMP is fully supported in Polaris).

The code communicates with the parallel back end through Polaris assertions. We have developed several new assertions to implement the concepts of runtime privatization, copy-in, last-value and reduction, as well as to implement the parallelization of pushbak sequences. The definitions of these assertions are in directory *base/Directive* and the code that manages the generation of *OpenMP* directives is in *postpass/-PostPassOpenMP.cc*.

The generation of LRPD tests is also managed through directives at this point. Their actual code generation is handled by routines in directory *rttest/*.

Run Time Library

The code generation phase relies on a run time support library.

USR Evaluation

Implements USR operations: union, intersection, difference, relocation (for translation across subroutine boundaries). It also implements operations based on a USR mask: copy in, copy out, zero out and partial reduction.

The run time USR data structure packs a list of LMADs into a two-dimensional array. This discussion considers row major order.

- Element (0,0) contains the size of each row.
- Element (0,1) contains the number of rows (one LMAD per row).
- Element (i, 0) contains the number of dimensions of the i-th LMAD.
- Element (i, 1) contains the starting offset of the i-th LMAD.
- Element (i, 2*j) contains the stride of the i-th LMAD in dimension j.
- Element (i, 2*j+1) contains the span of the i-th LMAD in dimension j.

The union, intersection and difference operations rely on aggressive simplification based on the interleaving, coalescing and contiguous LMAD aggregation techniques introduced by [13]. The current implementation tries to solve simple problems in place, keeping the descriptors in their original two-dimensional arrays, based on simple heuristics. Harder problems may require complex manipulation of LMADs and fall back to a linked list representation. For performance purposes, the current implementation uses a custom memory allocator (stack based), with a global deallocation phase after each call to the library. The USR evaluation library is not thread safe at this time, except for the masked operations (assuming they are called on different arguments).

The masked operations consist of asymptotically as many operations as the number of array elements described by the USR. This may be dramatically lower than using a mask based on dense shadow arrays, which would be proportional to the size of the array.

LRPD Test

There are two modules, one for marking and the second for analysis. The marking functions are usually inlined so only the second module is normally linked in.

Other PDAG Evaluation

As of now, the only other PDAG evaluation operations that are neither simple logical expressions, USR evaluation or LRPD, is an interval disjointness test. The decision function expects two arguments: a vector containing the intervals as pairs (begin, end), and the interval count (all integers). The implementation is based on sorting the intervals using their beginning as the key, and then making sure that $end_i < begin_{i+1}$.

Support Library

Memcpy module. It implements memory copies that are used either directly for checkpointing or to implement operations based on a USR mask. The current implementation is based on the standard *memcpy* call.

Instrumentation module. It implements timing and counting functionality. The interface consists of three functions: *timer_init(int*)*, *timer_start(int*)* and *timer_stop(int*,int*)*. Upon the first call to *timer_init*, the library opens and reads a file named *TIMERS*. This file must contain a list of names (one name per line) corresponding to the timers in the application. Timer 0 will correspond to the first

name and so on. The first parameter of each call to the timing routines must be this timer id. The second parameter to *timer_stop* is a possibly dynamic event type, such as "the loop was parallel".

The instrumentation module registers a display routine with the *libc atexit* mechanism. A timing/counting summary is printed upon successful program execution to file *timing.out*.

Debugging Run Time Tests

USR Evaluation. There are four compile time options that can control performance and debug options.

- *-D_STACK_ALLOCATOR=...* specifies that the library will use the fast stack allocator rather than the *libc malloc*. The argument is the size of the stack. The option we use is *-D_STACK_ALLOCATOR=5000000*.

- *-DHEAP_MONITOR* turns on a rudimentary mechanism that catches memory leaks. The default is not to use this flag as it incurs additional overhead.

- *-DDISPLAY_DIAGNOSTICS* turns on the HTML display. The results of each USR evaluation is recorded in file *rt_parallel_report.html*. The size of the file can be very large if partial operations or test inspectors cannot be reused (each dynamic instance will be recorded).

- *-DDEBUG_OPERATIONS* turns on the text mode display. It lets you follow the order of evaluation of USR operations as well as their operands, result and destination register. This flag should be used in conjunction with the USR display flag 2 (see user manual).

LRPD. The LRPD marking operations can be debugged by inserting tracing

instructions in the marking routines. It is better to link in the library rather than inline it so that modifications to the library do not require recompilation of the application. The inlining flag is *rt_inline_sub*.

Instrumentation. The timing routines have built-in consistency checks. They verify that the *TIMERS* file exists and has the right format. There is also basic verification of the way the timers are used. For instance, a run time error will occur if a timer is started twice. These run time checks must be switched on at compile time using compilation flag *-DDEBUG_TIMERS*.

Value Evolution Graph

The Value Evolution Graph (VEG) is implemented in directory *cvdl/Evolution*. In addition to the *EvolutionGraph* class, the directory contains a set of higher-level information routines, such as *eg_compare* or *eg_range*. The VEGs can only be built when the program is in SSA. The construction of a VEG for a loop will trigger the construction of all the VEGs in all inner loops interprocedurally.

Filters

The filters are independent passes over the whole program. They all reside in directory *filters/*. Filters are run by their own driver. Each filter must be registered with the driver. At registration, the filter must be given the list of other filters that it depends on (that must be run before it).

The driver reads the switches file, resolves dependences, orders the filters and run them. The filters require the program not to be in SSA.

VITA

After graduating from Liviu Rebreanu High School in Bistrita, Romania, Silvius Vasile Rus was accepted to the Department of Mathematics and Informatics at the Babes-Bolyai University in Cluj, Romania. He graduated in June 1997, and after a year working as a Systems Analyst for TIM SA Cluj, was accepted to the Department of Computer Science at Texas A&M University in College Station, TX in August 1998. He held a teaching assistant position in the fall of 1998, and was a research assistant to Dr. Lawrence Rauchwerger from 1999 to 2005. He received a best student paper award at the International Conference of Supercomputing in New York, June 2002. He spent the summer of 2002 at the IBM Thomas J. Watson Research Center, where he worked on porting the MPI library to the BlueGene/L supercomputer. He was awarded the annual best student research prize by the Department of Computer Science at Texas A&M University and held a GAANN fellowship from fall 2005 through summer 2006. He graduated with a Ph.D. in Computer Science from Texas A&M University in 2006. His dissertation was directed by Dr. Lawrence Rauchwerger.

Dr. Rus can be reached by mail at Silvius Rus c/o Lawrence Rauchwerger, Texas A&M University, Department of Computer Science, TAMUS 3112, College Station, TX 77843-3112. His email address is silvius.rus@gmail.com.

The typist for this thesis was Silvius Rus.