INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

COMPILE-TIME PERFORMANCE PREDICTION OF SCIENTIFIC PROGRAMS

BY

GHEORGHE CĂLIN CAŞCAVAL

Dipl., Institutul Politehnic, Cluj-Napoca, 1991
M.S., West Virginia University, 1995

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2000

Urbana, Illinois

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: 9989955

UMI Microform 9989955

Copyright 2001 by Bell & Howell Information and Learning Company.
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

© Copyright by Gheorghe Călin Caşcaval, 2000

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

JUNE 2000
(date)

WE HEREBY RECOMMEND THAT THE THESIS BY

GHEORGHE CALIN CASCAVAL

ENTITLED COMPILE-TIME PERFORMANCE PREDICTION OF SCIENTIFIC PROGRAMS

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF DOCTOR OF PHILOSOPHY

Director of Thesis Research

Head of Department

Committee on Final Examination†

Chairperson

† Required for doctor's degree but not for master's.

O-517

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To my family.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgments

I would like to thank Professor David Padua for being the ideal advisor. His ability to turn my confused ideas into simple statements continues to amaze me, and I can only hope that some of his wisdom has been transfered to me throughout our collaboration.

I would also like to thank professors Samuel Kamin and Constantine Polychronopoulos and Daniel Reed for serving in my thesis committee and committing their time to consider my work.

My thanks go to the members of the Polaris group, past and present, for creating such a productive environment. Also I want to thank my office mates, José and the other "architecture guys", for bearing with my questions, and creating a wonderful atmosphere with endless discussions and jokes.

Special thanks to my best friend George. From our days at the university in Cluj when we were writing low-level drivers for DOS, to the database gateways, hot-cup models and the Visual Basic experience in West Virginia, from stack algorithms to changing transmissions, from Mat*marks* to Mindstorms and baking play-dough cars, it's been a lot of fun. I can just hope that the fun will continue.

None of this work would have been possible without support from my family. My two year old son, Dan, who just said that he's not upset with daddy gone to the office the whole day. My wife, Anca, who has shown me that it is possible to work, go to school and raise a child. Without her support this whole thesis would not have been possible. I also want to thank my parents for the way they raised us and for all the encouragement that we have gotten from them through the years.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table of Contents

Chapter 1 Introduction 1
- 1.1 Motivation 1
- 1.2 Problem Domain 5
- 1.3 Data Dependences 7
  - 1.3.1 Data Dependences in Loops 9
  - 1.3.2 Uniformly Generated Dependences 10
- 1.4 Contributions 11
- 1.5 Thesis Organization 13

Chapter 2 Related Work 14
- 2.1 Performance Prediction Environments 14
- 2.2 Compile-time Estimation of Cache Misses 17
- 2.3 Locality Metrics 21
- 2.4 Compilation Using Performance Hints 23

Chapter 3 Compile-time Performance Prediction 25
- 3.1 CPU Prediction 27
- 3.2 Memory Hierarchy Prediction 31
- 3.3 The Stack Distances Algorithm 33
  - 3.3.1 Introduction 33
  - 3.3.2 Algorithm Overview 36
  - 3.3.3 Iteration Space Partitioning 38
  - 3.3.4 Dependence Spans 41
  - 3.3.5 Array Sections Computation 43
  - 3.3.6 Stack Histogram 49
  - 3.3.7 Example - Matrix Multiplication 50
  - 3.3.8 Spatial Locality 53
  - 3.3.9 Associativity 54
- 3.4 Indirect Accesses Model 55
- 3.5 Summary 57

Chapter 4 Stack Distance and Stack Algorithms 58
- 4.1 The Stack Distance as a Metric for Locality 58
- 4.2 LRU Stack Processing Algorithms 63
  - 4.2.1 Naive Implementation 64
  - 4.2.2 Markers Algorithm 65
  - 4.2.3 Alternative Data Structures 65

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2.4 Bennett and Kruskal's Algorithm 67
4.2.5 Hole-based Algorithms 68
4.2.6 Interval Tree of Holes 69
4.2.7 Preallocated Tree of Holes 72
4.2.8 Experimental Evaluation 74
4.2.9 Notations 80
4.3 Summary 80

Chapter 5 Polaris Performance Prediction Framework 82
5.1 The Polaris Framework 82
5.2 Integration with SvPablo 87
5.3 Summary 92

Chapter 6 Experimental Results 93
6.1 Experimental Setup 93
6.2 Results 94
6.2.1 Cache Miss Prediction with the Indirect Accesses Model 94
6.2.2 Execution Time Prediction with the Indirect Accesses Model 99
6.2.3 Cache Miss Prediction with the Stack Distances Model 102
6.2.4 Execution Time Prediction with the Stack Distances Model 110
6.3 Summary 112

Chapter 7 Conclusions and Future Work 113

References 116

Curriculum Vitae 124

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Tables

1.1 Instruction cache misses in the SPECfp95 benchmarks 7

3.1 Operation groupings 28
3.2 Stack distances computation for matrix multiplication 52

4.1 Inter-reference distances and averages for memory references in Programs 1 and 2 . . 59
4.2 Stack distances and number of references in Programs 1 and 2 60
4.3 Temporal locality for matrix multiplication 63
4.4 Perfect-Club Benchmarks run times (seconds) 75

5.1 Method functionality in the Polaris performance prediction framework 85
5.2 Method functionality for memory cost estimators 86

6.1 Compile-time stack distances accuracy 104

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Figures

1.1 Compiler-based prediction environment 4
1.2 Example of data dependence graph. 8
1.3 Iteration vectors 10

3.1 Stack update when the currently referenced location has been previously accessed . . 33
3.2 Stack histogram for QCD 35
3.3 Iteration space partitioning algorithm 39
3.4 Partitioned iteration space for matrix multiplication. 40
3.5 Iteration space for matrix multiplication. The shaded shape represents the iterations spanned by loop-carried dependence with distance 1 in dimension i. 42
3.6 Dependence span computation algorithm 42
3.7 A dependence span projected onto array sections 44
3.8 Array section computation algorithm 45
3.9 Non-contiguous intervals: representation and calculation 48
3.10 Stack histogram computation algorithm 50
3.11 Matrix multiplication example. 51
3.12 Cache lines mapping on an array section 54
3.13 Sparse matrix vector multiplication 56

4.1 Fortran code for tiled matrix multiplication 61
4.2 Stack histograms for matrix multiplication and tiled matrix multiplication 62
4.3 Stack algorithm 64
4.4 Modified stack algorithm 67
4.5 A partial sum hierarchy 68
4.6 An interval tree 70
4.7 Updating the tree of holes 71
4.8 Interval tree update 73
4.9 Increase in execution time with respect to the optimized program of instrumented code. preallocated hole tree algorithm and Bennett and Kruskal's algorithm 77
4.10 Execution time breakdown for the preallocated hole tree algorithm 78

5.1 Polaris performance prediction framework 84
5.2 Polaris performance prediction interface to SvPablo 89

6.1 SpLib – L1 cache miss prediction for the small data set 95
6.2 SpLib – L1 cache miss prediction for the large data set 97
6.3 SpLib – L2 cache miss prediction for the large data set 98
6.4 SpLib – unoptimized execution time prediction accuracy 100

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.5 SpLib - optimized execution time prediction accuracy . . . 101
6.6 Jacobi – cache miss prediction on the R10000 . . . 103
6.7 SWIM - cache miss prediction on the R10000 . . . 107
6.8 TOMCATV - cache miss prediction on the R10000 . . . 108
6.9 SPECfp95 - cache miss prediction accuracy on the R10000 . . . 109
6.10 SPECfp95 - execution time prediction for selected loops in each benchmark using the -O2 optimization flag . . . 111

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 1

Introduction

1.1 Motivation

A major fraction of the time taken to develop scientific applications is spent in parallelization and performance tuning. This fraction is even larger if the application is required to run on several platforms. because specific architectural characteristics may require different optimization techniques for best performance.

In order to reduce development time we have seen. in recent years. a continuous effort to improve compilers to handle automatic parallelization and optimization. However. providing the compiler with a list of optimizations and applying these optimizations blindly is not enough. The optimized program may run slower than its unoptimized version. For example. consider the loop interchange optimization for the following loop nest:

```
do j = 1, n
   do i = 1, n
      a(j) = a(j) + b(j,i) * c(j)
   enddo
enddo
```

Assuming that the matrix b is stored in column major order. if we do not interchange. we can expect to have a cache miss every iteration. because b is not accessed with stride one. If we do

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

apply the interchange transformation. all the accesses are stride one. therefore there will be one cache miss every several iterations. depending on the size of the cache line. However. in the non-interchanged version. the array elements a and c can be stored in registers for all the iterations of loop i. therefore in the innermost statement. there will be only one load and two floating point operations. an add and a multiply. If we interchange. we need two extra loads and one store for each iteration of the inner loop. Thus. depending on the cache miss penalty. combined with the number of functional units in the processor. in this case floating point units and load/store units. the loop interchange optimization may actually hurt the performance. even though it reduces the number of cache misses in the loop.

The work presented in this dissertation is directed towards helping compilers do a better job in optimizations. By constructing a performance prediction model inside the compiler. we provide compiler writers with a non-empiric tool that will allow them to select the order in which the compiler applies optimizations to maximize performance.

The same performance model is used in the Delphi system [57] to statically predict performance. In the Delphi project. the goal is to create an integrated environment in which a user can develop. compile and tune the performance of applications in an efficient and transparent manner. Delphi integrates compilers with performance tuning and performance visualization tools. The static predictions presented in this thesis have been used as part of this project.

We propose to include the performance prediction model inside the compiler. The performance model consists of symbolic expressions with terms that account for the program constructs. the data set and the architecture. In the ideal case. in which all the loop bounds and branch frequencies in the program are known at compile time. the compiler can generate these expressions without using profiling information or user interventions. However. if profiling information is necessary. we have provided the necessary hooks so that the profiling information can be collected and used by the performance models. The advantages of using a symbolic performance prediction model are detailed in Chapter 3. Here we enumerate just a few. First. not all information is available statically. at compile time. Whenever the compiler encounters an unknown value. it can use its symbolic representation to continue building the model. If, in the end the value is still not resolved. the compiler could either use profiling data. or simply provide the performance information using

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the symbolic expression. The symbolic expression can be used either for run-time decisions or for scalability analysis. Also. by using symbolic expression the compiler avoids magnifying the prediction error of compounding estimates (the usual method employed in most of the prediction systems). since no approximations are made at any intermediate step. Of course one must pay for all these benefits. The costs are the need of a more complex compiler that includes an accurate symbolic expression manipulator as well as a slight increase in compilation time due to the symbolic manipulation. The symbolic expression mannipulator must perform simplification and comparison of algebraic expressions.

Figure 1.1. shows the architecture of an integrated compilation and performance tuning system built around a static performance prediction model. In this environment. the compiler analyzes the source code and synthesizes symbolic expressions representing performance data. There are several paths that can be taken to obtain an optimized program. The first path. representing the the ideal case. is shown with a thicker line. In this case the compiler is able to completely analyze the program. there are no unknown parameters in the performance expressions. and based on these. the compiler can decide which optimizations to apply.

Of course. the ideal case does not occur very frequently in practice. therefore a second path. using profiling information (shown with a dashed line) is provided. In this scenario. the performance prediction module uses available profiling information. such as true and false branch frequencies. or the number of iterations of a loop. Branch frequencies could be estimated at compile time [4]. however. in this work we have chosen to use profiling information because it is more accurate. The third path represents the case when profiling data is not available. and the system can be set up to collect such information and use it. In this scenario. the compiler analyzes the code. and it also places instrumentation code to extract the needed values. The instrumented code can be run with different data sets to extract the profiling data used as parameters for the performance expressions.

All the paths described in Figure 1.1 have been implemented as part of the Polaris compiler [8]. and the system has been used to generate the results presented in Chapter 6. as well as part of the Delphi system. In the current implementation we can access performance data inside the compiler. and if profiling information is needed. Polaris can generate code to collect the information. and use it in evaluating the symbolic expressions that represent performance prediction data.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

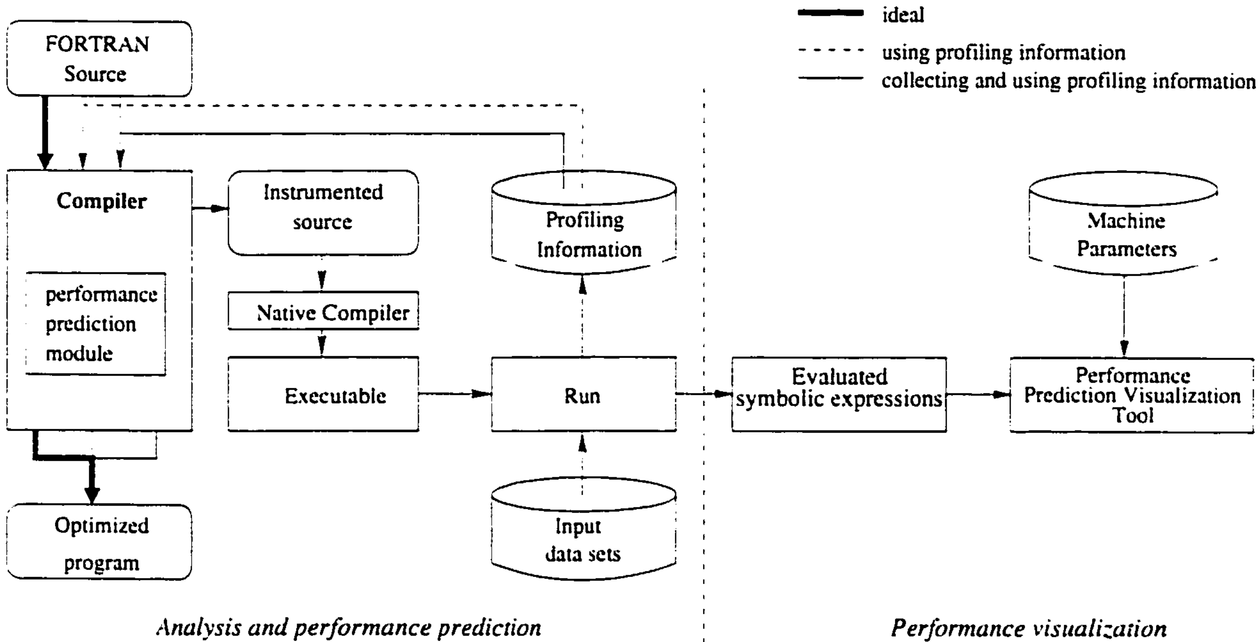


Figure 1.1: Compiler-based prediction environment

The right hand side of Figure 1.1 shows how the performance expressions can be visualized using a performance visualization tool. such as SvPablo [21]. In this scenario. the performance symbolic expressions are evaluated for a specific machine and the numbers obtained from the evaluation are displayed in a graphical user interface. This scenario is useful in comparative system evaluation. because different machine parameters can be substituted in the expressions and the user can study how machine characteristics affect application performance. System evaluation can be used by computer architects in the process of designing new machines. and by users to select the best platform suited for their application needs. Similarly. scalability analysis can be conducted using the symbolic performance expressions. Since the expressions contain variables for the data size. one can study how varying the size of the problem affects the performance of the application on a particular machine.

Besides the *what-if* games. performance analysis tools can benefit from compiler generated models because they can generate results faster than by executing the code. Along with execution time figures. the compiler can provide the performance analysis tool with a wealth of information

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that will enable the tool to better relate the dynamic behavior of the application to the high level language code.

In recent years we have seen a new trend in processor and system design. a much closer integration between the architecture design and the compiler design. Modern processors rely heavily on the compiler to organize the code so that it takes advantage of the hardware features. For example. the IA-64 architecture proposed by Intel and Hewlett-Packard. relies on the compiler to create bundles of VLIW instructions that also contain specifications of which instructions can be executed in parallel. as opposed to current superscalar processors that try to discover the instruction level parallelism in hardware.

More recently. a new type of architecture has emerged. the intelligent memory architectures [49. 33. 37]. In these architectures the DRAM memory contains also processor logic. enabling low latency and high bandwidth between the processor-in-memory (PIM) and the memory. The PIMs act as co-processors that execute code when signaled by the host processor. In this architecture it is very important to decide what portions of code execute on each processor. as the host processor is more powerful and backed up by a deep cache hierarchy. but has higher memory latency. The PIMs are typically less powerful. have very low memory latency and no cache. The static prediction models presented in this dissertation have been used in recent work [64] to automatically map the code to the host or to the PIM based on performance prediction results.

In the remaining sections of this chapter we first discuss the problem domain on which we focus our modeling. Next we present a quick overview of data dependence information since data dependences provide the main foundation for our work. and we conclude by discussing the contributions of this thesis.

1.2 Problem Domain

The problem that this work proposes to solve can be formulated as follows:

> Model the performance of scientific applications on a computer system. inside a compiler. by looking at high level source code only.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The approach taken is mostly architecture independent. However. because of the complexity of the interactions between different parts of the system. as well as the optimizations done by the native compiler. some limitations apply.

On the architectural side. we decompose the computer system into parts that can be modeled relatively independent. Thus. we assume that the CPU. the memory hierarchy. and the I/O subsystem. can each be modeled separately. and their effects on the application performance are additive.

The CPU is assumed to be a superscalar processor with multiple functional units. The processor can issue several instructions per cycle. Each instruction can have a different latency. In Chapter 6 we present results for two different processors. the MIPS R10000 [56] and the UltraSparc II*i* [66]. The R10000 is an out-of-order processor while the UltraSparc is an in-order processor. Both can issue several instructions per cycle. Also. in [64] it has been shown that. by using our approach. it is possible to model statically the behavior of two types of processors with quite different characteristics. Even when the prediction was not very accurate (average prediction error of 30%). the static predictor based on our methods correctly predicted the relative execution time for these processors. In the IRAM case. this was sufficient to decide where to execute the code.

The memory hierarchy consists of several levels of cache and the main memory. The caches can have different cache line sizes and associativities. We model the data caches only. although the models could be extended for instruction caches. We chose to ignore the instruction cache misses since their impact on the performance of scientific codes is negligible (on average 0.17% of execution time for L1 and L2 instruction caches on the R10000 for the SPECfp95 benchmarks). For a breakdown on all the programs. see Table 1.1. Both the R10000 and the UltraSparc have two levels of cache. with the first level having separate instructions and data caches. and the second level consisting of a unified cache. The line sizes and the associativities differ between the two processors.

The application domain consists of scientific Fortran programs. such as codes in the SPECfp95 benchmark suite. There are two main reasons for focusing on Fortran. First. many of the codes performing computations for scientists are written in Fortran. This is illustrated by the fact that 10 out of 14 (71%) codes in the new SPECfp2000 benchmark suite are written in Fortran. Second.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Benchmark	Code size		L1 I-cache misses		L2 I-cache misses	
	lines	cycles	#	% exec	#	% exec
APPLU	2474	51247966388	1465086	0.05	715579	0.12
APSI	4238	2046974010	263773	0.23	50557	0.21
HYDRO2D	1667	66288394854	1353887	0.04	497572	0.06
MGRID	382	51019153876	1474727	0.05	637206	0.10
SU2COR	1444	31218347345	1075432	0.06	476323	0.13
SWIM	282	36220627134	316612	0.02	169086	0.04
TOMCATV	109	43131557047	638084	0.03	288793	0.06
TURB3D	1287	6443904684	129263	0.04	42511	0.06
WAVE5	6314	32019987230	1872195	0.11	497038	0.13
Average	2022.89	35515212507.56	954339.89	0.07	374962.78	0.10

Table 1.1: Instruction cache misses in the SPECfp95 benchmarks

although the techniques described in this work are not restricted to Fortran. the infrastructure tools that we used handle mainly Fortran.

We have selected scientific codes because of their relatively simpler control flow structure. In scientific codes most of the computation happens in loops accessing arrays. The prediction models focus on high level source code and since we don't know what low-level optimizations are performed by the native compiler. such as instruction scheduling and register allocation. we approximate the potential optimizations using heuristics. The heuristics presented in Chapter 3 are targeted towards scientific codes.

1.3 Data Dependences

Data dependences [5. 76] are used in the compiler to represent variable references that potentially access the same memory location. Most optimizing compilers use three types of data dependences: *flow*. *anti* and *output* dependences. Since we focus on memory behavior. we are also interested in *input* dependences. More formally. these types of dependences are defined as follows.

Definition 1.1 *Consider two statements S and T that both reference the same variable A (read or write). and T is executed after S. We say that:*

1. *T is flow-dependent on S if S writes A and T reads A;*

2. *T is anti-dependent on S if S reads A and T writes A;*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

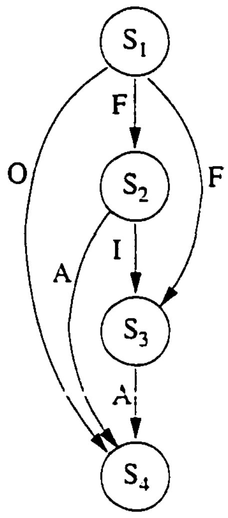


Figure 1.2: Example of data dependence graph.

3. *T is output-dependent on S if both S and T write to A;*

4. *T is input-dependent on S if both S and T read A.*

S is called the source of the dependence and T is called the target of the dependence.

For cache behavior it is more important to know which locations are accessed successively than the type of the access. read or write. Consider the following sample program:

(S_1)	A = 0
(S_2)	B = A + 1
(S_3)	C = A + D
(S_4)	A = 2

The data dependences for this program are shown in the data dependence graph in Figure 1.2. Each dependence is marked by its type. There is a flow dependence between statement S_1 and statement S_2 (marked F) because statement S_1 writes to variable A and statement S_2 reads A. There is an input dependence between S_2 and S_3 (marked I) because both statements read the variable A. There is an output dependence between S_1 and S_4 (marked O) because both statements write to A. And there is an anti-dependence between S_3 and S_4 (marked A) because statement S_3 has to read the variable A before it is written by statement S_4.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.3.1 Data Dependences in Loops

In loop bodies, statements are executed multiple times. Data dependence relations can exist from any instance of execution of a statement to any other statement, including itself. Since the compiler cannot represent all the instances of a statement (the number of iterations may be unknown at compile time), the data dependence graph is abstracted to represent in one node multiple instances of the same statement. Dependence edges are then annotated to identify the relative iterations in which the dependence relations occur. Based on the iterations of the source and target of the data dependence we can classify the dependences as:

- *loop independent dependence* - if both the source and the target of the dependence are in the same iteration of the loop;
- *loop carried dependence* - if the source and the target of the dependence are in different iterations of the loop

Another important concept is the *iteration space* associated with a loop nest. The iteration space is a polytope that contains one point for each iteration of the loop. For any loop carried dependence, there will be an edge from the source iteration to the target iteration in the iteration space dependence graph. Since compiler cannot always determine the number of iterations in the loop, the iteration space is expressed symbolically.

In order to identify the points in the iteration space we assign an *iteration vector* to each iteration. There are two kinds of iteration vectors described in literature, one based on loop index variables, *index variable iteration vectors* and one that enumerates the iterations, the *normalized iteration vectors*. In the index variable iteration vectors (Figure 1.3a), each element I_k represents the value of the loop index variable for the kth nested loop at that iteration. In the normalized iteration vectors (Figure 1.3b) the iterations of each loop are enumerated starting either at 0 or at 1, and these are the values used in the iteration vector. The advantage of using normalized iteration vectors is that later iterations have lexicographically larger valued vectors than earlier iterations, making it easier to order the iterations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\vec{i} = \begin{pmatrix} I_1 \\ I_2 \\ \vdots \\ I_n \end{pmatrix}$$

(a) Index variable iteration vector.

$$\vec{i} = \begin{pmatrix} 1 \\ n \\ \vdots \\ 2 \end{pmatrix}$$

(b) Normalized iteration vector.

Figure 1.3: Iteration vectors

The *dependence distance vector* is the vector difference between the iteration vectors of the target and source iteration. Thus, the dependence distance vector can be expressed as:

$$\vec{d} = \vec{i}^T - \vec{i}^S$$

where $\vec{i}^T$ is the target iteration vector, and $\vec{i}^S$ is the source iteration vector. We will represent the dependence distance vectors on the dependence graphs as a comma separated list of the elements of the distance vector, e.g. (0, 1, 0).

Since the dependence graph for loops represent in one node multiple instances of the same statement, the compilers cannot always compute an exact value for the dependence distance vectors, because the instances of the source statement can be at different distances of the corresponding instances of the target statement. In this case, the dependence distance vectors need to be summarized by using *dependence direction vectors*. An element of the dependence direction vector takes values in the set $\{<, =, >, *\}$, with $*$ representing an unknown direction. If the compiler can compute the distance vectors it can derive the direction vectors by taking the sign of the distance vector.

1.3.2 Uniformly Generated Dependences

Uniformly generated dependences [28] are data dependences for which the distance vectors are known and contain only constant values. If two references have a known distance vector between them, then the compiler has determined that the two references will touch the same memory location. In order to compute if the memory accesses reuse data in the cache, all that needs to be done is to compute the distance, in terms of memory references, between the two references.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and check if the distance is less than the cache size (for fully associative caches). Most of the dependences that cause reuse are uniformly generated [28, 52, 74]. Many of the other dependences, such as those with symbolic subscripts or index arrays, will rarely access the same location. For those dependences even coarser approximations of the reuse, such as [74, 14], usually yield good results.

1.4 Contributions

This work has originally started as a study of data locality metrics. In many cases optimizations for improving data locality are applied and the only measure of the "improved" performance is the execution time. We wanted to have a metric that quantifies precisely both temporal and spatial locality, as a function of the program only, that is, independent of the architecture on which the application will run. Up to this work most of the methods either need the cache parameters of the architecture, such as cache size, associativity, etc., or are not able to apply the same metric to either a loop nest, a subroutine, or an entire program. Therefore we started looking at stack algorithms.

Stack algorithms have been used previously to characterize paging behavior [43]. They have the inclusion property (the stack for a smaller cache is included in the stack for a larger cache) allowing estimations independent of the cache size. In addition, techniques have been developed to deal with set-associativity and different cache line sizes in one pass through the trace. The result of the stack algorithm is a histogram that counts the number of references at each distance from the top of the stack. The work presented in this thesis is based on the stack distances and the stack algorithms. We present a new algorithm to compute the stack histogram, faster than the best known algorithm. The development of this algorithm stemmed from the need for a faster stack processing method due to the length of the traces when the entries in the trace are memory references and not pages. We also present how the stack distances can be used to quantify locality, both temporal and spatial, at any program granularity.

When we started working on the Delphi project (an integrated system for performance measurement and tuning) and the need arose for a compile-time method to estimate cache behavior, we again turned to stack distances. Once more, the most attractive feature of the stack algorithm is its

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

architectural independence. One can predict the number of cache misses for any cache size once the stack histogram is computed. Thus the main contribution of this thesis was defined: a compile-time algorithm that computes the stack histogram based on data dependence distance vectors. Although the compile-time algorithm estimates the number of cache misses for fully-associative caches, the experimental results presented in Chapter 6 show that the estimations are quite accurate for 2-way set-associative caches. Moreover, the intended use for this method is to help drive compiler optimizations, in which case the relative performance of different code variants is more important than 100% accuracy.

The main contributions of this work are summarized as follows:

Compile-time model for estimating the number of cache misses. We present a new method for estimating the number of cache misses in a loop using the stack histogram. The stack processing algorithm and its result, the stack histogram, have been historically used to evaluate caches. In this work we describe a method that computes the stack histogram at compile time, based on the data dependence distance vectors. Besides being accurate, the method presented is also fast since it relies on data already available in the compiler (data dependences are calculated for other compiler optimizations), and applicable to more than 75% of the loops present in the SPECfp95 benchmark suite. Detailed results are presented in Chapter 6.

A new algorithm for stack processing. During our work with the stack processing algorithm we have come up with a new method to process a memory trace, that it is faster than the best current algorithm [6]. The new algorithm is presented in Section 4.2.

A new metric for locality, the stack histogram. The stack histogram provides a better metric for quantifying the locality in programs than previous work. This is based on the fact that the stack distance computes exactly how many *distinct memory locations* are accessed between accesses to the same location, as opposed to other methods that average over the number of memory locations accessed. This is discussed further in Section 4.1.

Integrating the performance modeling with the compiler. We present a new compiler framework in which performance data is available at compile time as a symbolic expres-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sion. independent of the architecture. A compiler writer can use this information to drive optimizations. We also show how we integrated performance modeling at compile-time with a performance visualization tool in Chapter 5.

1.5 Thesis Organization

This thesis is organized as follows: in Chapter 2 we present related work for each of the areas that we touch upon: performance prediction environments. compile-time estimation of cache misses. locality metrics and iterative compilation. In Chapter 3 we present the compile-time performance prediction model. We describe the entire model. and then we detail the CPU and memory hierarchy models. Chapter 4 has two parts. In the first part we present a new metric for program data locality based on the stack distances. The second part describes our experience with stack processing algorithms and a new algorithm for efficiently computing stack distances. In Chapter 5 we present the performance prediction framework implemented in Polaris. as well as the interaction between the framework and the SvPablo performance visualization system. Chapter 6 presents experimental results and we conclude in Chapter 7.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 2

Related Work

In this chapter we describe previous work that has been done in several areas related to this dissertation. We begin by presenting solutions for performance prediction environments that integrate compilers and runtime systems to aid compiler optimizations. These environments have similar goals to our Delphi project [57]. Next we focus on compile-time prediction of cache behavior. Several approaches are presented. some of them integrated in a performance prediction environment. others used for driving optimizations.

Then. we look at existing metrics for locality. We claim that the stack histogram proposed in this thesis is a more accurate metric than the cost models available in the literature. We conclude by presenting work that uses estimates of execution time inside a compiler to improve performance. We discuss these efforts to underline the need for and the applicability of an accurate static cache model.

2.1 Performance Prediction Environments

Fahringer [24. 22] describes P^3T. a performance estimation tool. He uses the Vienna Fortran Compilation System as an interactive parallelizing compiler. and the *WeightFinder* and P^3T tools to feedback performance information to both the compiler and the programmer. Our work differs from his in both the estimation of computation time and the estimation of the number of cache misses. First. to estimate computation time. we use compile-time analysis and micro-benchmarking as opposed to his pattern matching benchmarking against a library of kernels. The combination of compile-time analysis and micro-benchmarking is better suited to hide the underlying architecture

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

than pattern matching. especially if dependence analysis is used to detect overlapping operations.

The authors classify the benchmarking kernels into four categories: *primitive operations*. which contain basic operations such as +. *. etc. and elementary array access kernels: *primitive statements* such as DO loop headers. conditional statements. etc.: *intrinsic functions* and *code patterns* which include standard code patterns amenable to recognition such as elementary operations of linear algebra (matrix multiplication. matrix inversion. determinant calculation. etc.) and commonly used stencils such as Jacobi relaxation. LU decomposition. etc. Each kernel is subsequently ran for different data sizes. on the machine for which the performance estimation is desired. to measure its execution time. The performance estimator parses the program and detects existing library kernels. For each kernel. the premeasured execution time is accumulated to obtain an overall execution time. The authors underline the difficulties they encountered while developing the kernel library for two machines. the Intel i860 and MasPar MP-1. and they recognize the fact that it is very difficult to obtain complete. portable kernels. A second difference is in the cache modeling. We present their cache model in detail in Section 2.2.

Saavedra et al. [60. 59. 58] has done extensive work in the area of performance prediction for uniprocessors. In [60]. the authors present the micro-benchmarking concept to measure architectural parameters. Micro-benchmarking consists of a set of kernels. each kernel targeted at one particular feature of the machine. The kernels are written in such a way that they try to isolate one feature and measure its characteristics by minimizing the effect of other features. We use their micro-benchmarking approach to measure operation costs and cache latencies. In [59]. the authors present an abstract machine model that characterizes the architecture and the compiler. Their early model does not consider memory hierarchy effects. They consider such effects in [58] by combining the measurements of cache and TLB timings obtained through micro-benchmarking with cache and TLB miss ratios obtained through simulation by Gee et al. [29]. The results are used to evaluate how the execution time prediction improve when memory delays are incorporated and how much impact the cache and TLB configurations have on the overall performance of the machine. The main difference between our work and theirs is that we predict both the CPU time and the cache misses at compile-time. while their method predicts the CPU time and uses previously published miss ratio data.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Wang [69] develops a performance prediction framework for superscalar-based computers. His framework. as ours. is designed to be used inside an optimizing compiler. to guide program transformations. The following requirements are listed as critical for a good performance prediction tool:

- *precise* - the prediction must be accurate for the compiler to make correct decisions.
- *efficient* - the compiler will make repeated calls to the prediction module. therefore the prediction pass should be very efficient.
- *robust* - the framework should be able to handle programs with unknowns in control structures and unknown branch probabilities.

The key idea to satisfy all these requirements is to use symbolic expressions to represent performance data. The symbolic expressions will minimize the effects of compounded estimations for multiple basic block by delaying the evaluation of unknowns. therefore increasing the accuracy of the prediction. In addition. they will be more efficient to evaluate for different data sizes. and will allow for the presence of unknowns. The model decomposes the total performance cost into CPU cost and memory hierarchy cost. To estimate memory access times the author uses the cache cost model developed by Ferrante et al. [27]. discussed in the next section.

For the processor cost. the framework contains an instruction translation module. which has four tables that are used to translate high level language constructs into costs on a specific machine in two translation steps. In the first step. called *operation specialization mapping*. a high level operation table is used to map language dependent constructs into language independent operations. stored in a basic operations table. In the second step. the *atomic operation mapping*. translates the basic operations into costs for the processor based on two other tables. the atomic operation table and the atomic operation cost table. which contain the low level operations and their costs. respectively. The framework relies on information passed by the compiler to estimate the optimizations performed by the compiler back-end. such as instruction scheduling and register allocation. While this can give accurate results for a particular compiler. it also makes the system less portable. since the module needs to be re-implemented for each supported compiler.

Adve et al. [1] presents an integrated environment for predicting performance on multipro-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cessors. They integrate compile-time information with dynamic instrumentation to predict the execution time for entire programs. However. their compiler stores only information about dependences and decisions made in applying transformations. so that their performance visualizer can relate the measurements back to the source code but it does not actually predict the performance at compile-time.

2.2 Compile-time Estimation of Cache Misses

There are many efforts targeted towards estimating the cache behavior of programs within a compiler framework. However. many factors. such as limited compiler information. algorithms complexity and hardware unpredictability. have made the problem so challenging that none of the proposed solutions is a complete solution.

Porterfield [52] presents one of the first static models of memory performance based on data dependences. He defines the *Overflow Iteration*. $O(i)$. for a particular loop. as the maximum number of iterations of that loop that can have all the data accessed maintained in the cache at the same time without encountering any cache misses. The overflow iteration can help determine when a reference will be a miss during program execution. because it provides a measure of how much data is accessed between the end points of a dependence. Any dependence that requires more iterations of the loop than the overflow iteration will access more distinct blocks than available and will result in a series of misses during execution. Once the overflow iteration is known. every reference can have its hit ratio computed based on the dependence edges. Unfortunately. for overflow iterations to be effectively generated. precise interprocedural information should be available. and at the time. PFC did not contain that information. This has not permitted Porterfield to implement his algorithm. and besides a manual coded example for matrix multiplication. his thesis provides only speculative results. Later. Ferrante et al. [27]. used Porterfield's overflow iteration to estimate the number of cache misses at compile-time.

In [27]. Ferrante. Sarkar and Thrash consider automatic analysis of a program's cache usage to achieve greater cache effectiveness when used to guide program transformations. such as loop interchange. To determine the number of cache misses for a given loop nest. an upper bound on the number of distinct cache lines (DL) accessed in the innermost loop is determined. The method

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

works towards the outermost loop. computing DL at each loop level. The innermost loop that causes the cache to overflow is called the *overflow loop.* An upper bound for the total number of misses is obtained by multiplying the DL value for all the loops contained within the overflow loop. by the product of the number of iterations of the overflow loop and all its enclosing loops. For set associative caches the bound might need further adjustments to take into consideration set conflicts.

There are several assumptions made about the program and the architecture that can be handled using this method. The authors assume a set of normalized. perfectly nested loops. The array references considered in the analysis must have subscripts that are linear functions of the loop indices. otherwise each access is considered a miss. Execution profiling can be used if conditionals are present. however the paper does not provide a discussion on how the profiling data can be integrated with the analysis technique.

The algorithm bounds the number of distinct array elements for each array reference and uses these to compute an upper bound on the number of cache lines accessed by each array reference. It then combines the bounds for several references to compute DL for the loop. The paper presents exact formulae for the number of distinct array elements accessed when the subscript of the array reference is a function of one or two loop variables. and provides an upper bound for a more general subscript function. This approach is less costly than Porterfield's [52]. since the authors use the GCD test and Banerjee's inequalities instead of data dependence distance vectors. However. if data dependence distance vectors are already computed for other compiler passes. then. as we show later in this work. most of the overhead in using data dependences in analyzing cache behavior is already paid. We can not readily compare the accuracy of our algorithm versus theirs because the experimental results presented in their paper is restricted to matrix multiplication. which both algorithms predict correctly. They present results for matrix multiplication only because they use simulation to collect the actual number of misses. and thus are constrained by time.

Fahringer [25] presents an algorithm that estimates the number of cache misses for sequential and data parallel Fortran programs. The algorithm is based on the analysis of all array references in loop nests. classifying them with respect to data reuse and computing a cost function for the array classes that describes the cache behavior of the program. The author shows how to extend

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the cost function to procedures and entire programs. although no experimental results for entire programs are presented.

The cache misses estimation takes into consideration cache line size. cache sizes and data types in computing the number of cache misses. Two array references in the same array class with respect to a loop nest if they access some common memory location in the same array dimensions and reuse occurs across loop iterations. The method used is very similar to the one employed by Ferrante et al. [27]. in the sense that it uses the *overflow loop* to determine an upper-bound on the number of cache lines accessed by a loop. The algorithm iterates through the loops in a loop nest. starting from the innermost loop. and at each loop computes the array classes and the number of cache lines accessed. The algorithm ends when there are more cache lines accessed than available in the system. There are two differences that make our algorithm more practical. First. it is not easy to see how Fahringer's algorithm can be extended to estimate inter-nest misses. And second. his algorithm needs the cache size as a parameter. while ours can estimate the number of cache misses for all the cache sizes based on the stack histogram. Again. it is very hard to see how effective is his algorithm compared to ours. because the only experiment presented in the paper is Jacobi relaxation. for which both algorithms are very accurate.

McKinley[44. 45] uses a very simple cache model to drive optimizations for data locality and parallelism. In this model. the references with group-spatial and group-temporal locality are grouped in equivalence classes using simple heuristics. Two references exhibit group-temporal locality if the references are dependent. and the dependence is either loop independent or loop carried with a very small distance (< 2). Two references are group-spatial dependent if access the same array and their subscripts differ by a constant smaller than the cache line size in the dimension along the cache line.

The cost of a loop is given in terms of cache lines accessed by placing the loop as the innermost loop in the nest. For each reference class a representative array reference is considered and the cost is computed as follows: if the reference is loop invariant. there is one cache line accessed in the whole loop; if the reference has spatial locality it accesses one cache line every cache line size iterations; all other references are considered accessing one cache line per iteration. Although the model is very approximative. it works quite well in practice. and it is accurate for double nested

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

loops. because it always finds the correct relative ordering of the loops in the nest.

Ghosh. Martonosi and Malik [30] have introduced the *Cache Miss Equations* (CMEs) as a mathematical framework that precisely represents cache misses in a loop nest. They count the cache misses in a code segment by analyzing the number of solutions of a system of linear Diophantine equations extracted from reuse vectors. where each solution corresponds to a potential cache miss. For each reuse vector. two kinds of equations are generated: *compulsory equations*. that represent cold misses. and *replacement equations*. which represent the interferences with other references. The number of cache misses is computed by traversing the iteration space and solving the system of equations at each iteration point. Although solving these linear systems is an NP-hard problem. the authors claim that mathematical techniques for manipulating the equations allow them to relatively easily compute and/or reduce the number of possible solutions without solving the equations. Our algorithm differs from theirs because in one single pass we can compute the stack histogram which can be subsequently used to estimate the number of cache misses for any cache size. thus avoiding the repeated execution of the expensive part of the algorithm.

Vera et al. [68] propose a solution based on sampling techniques to speed-up solving CMEs. Statistical sampling allows them to approximate the absolute miss ratio for each reference by analyzing only a small subset of the iteration space. Results are given with a confidence interval. parameterizable by the user.

Gannon. Jalby and Gallivan [28] propose program transformations to improve cache and local memory behavior assuming software control over the cache management. They use data dependences to compute what memory locations have to be kept in the cache for best performance.

The general method is to define a *reference window* for each dependence. which contains the current set of elements that must be in the cache. i.e. those that will be used again. To compute the reference window size. they study several cases of data dependences. and classify dependences as:

- *uniformly generated dependence* – the data dependence distance vector can be computed exactly and all its elements are constants;
- *uniquely generated dependence* – a restricted case of uniformly generated dependence. in which there is only one dependence distance vector with constant elements;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- cyclic self-dependences – characterize input and output self dependences in loops. such as reductions. where the same array element is accessed in each iteration. These dependences are very important for cache management.

The reference window size is a good measure of how many elements have to fit in the cache for best performance. However. if the number of array elements exceeds the cache size. they have to decide which reference windows to keep in the cache and which to evict (remember the proposed architecture allows cache management by the compiler). Based on which windows are kept. the compiler can determine the hit ratios. They use this mechanism to study the effect of loop interchange and tiling on locality.

2.3 Locality Metrics

Lilja et al. [42]. in a discussion about the memory referencing behavior of multiprocessors. introduce the *inter-reference distance*. the number of memory references that occur between two references to the same memory location. They use the inter-reference distance to measure temporal locality. By averaging the inter-reference distances for all the variables in the program. they obtain a single number. that can be used as a metric to characterize locality as follows: as the temporal locality in the program increases. the value of the metric decreases. We shall see in Chapter 4 that this metric does not work in all the cases. and we shall propose the stack distances as a more precise metric to characterize locality. However. the inter-reference distance can be useful in predicting program referencing behavior and improving replacement algorithms [51]. The inter-reference distance was also used by Pyo et al. [55] to guide loop transformations in several routines in the Perfect Club benchmarks [7].

Wolf and Lam [73. 72] studied data locality and how data locality can be used to guide unimodular compiler transformations. In [73] they present a mathematical formulation of data locality based on the concept of reuse vector space. They define four types of reuse: *self-temporal* – a static reference accesses the same memory location. *self-spatial* – a static reference accesses memory locations in the same cache line. *group-temporal* – several distinct static references access the same location. and *group-spatial* – several distinct static references access memory locations in the same cache line. The metric used to quantify locality is the number of memory accesses (i.e.. cache

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

misses) in one iteration of the innermost loop (the loops considered in the paper are perfectly nested loops). For each type of reuse they compute an estimate of the actual number of memory accesses generated by each reference. The reuse vector space is compared against the localized iteration space (the iteration space of a loop that exploits reuse) to see if the reuse actually happens. Data locality for a program improves when as many as possible of the reuse vectors are included in the localized iteration space. without violating any dependences.

Most of the exploited reuse is temporal reuse and spatial reuse between uniformly generated references. To quantify the spatial reuse. especially group-spatial reuse. the references that operate on the same array and are uniformly generated are partitioned in equivalence classes. called uniformly generated sets. The number of memory accesses is computed for each uniformly generated set and the sum over all sets gives the metric for data locality. While this metric works well for guiding loop transformations. it is not clear how the metric can be extended to quantify inter-loop reuse or reuse across entire programs.

McKinley and Temam [46. 47] did an extensive study of locality for programs in the SPEC'95 and Perfect Club benchmarks. They employ a reuse classification similar to the one developed by Wolf and Lam [73]. but they quantify. through simulation. the locality for different program granularities: intra-nest. inter-nest and entire program. They also discuss the impact of their results on some popular assertions about program behavior with respect to caches. Both studies are mostly quantitative. in the sense that they do not propose any optimizations or algorithms. but they present a very detailed description of where and what types of misses happen in these benchmark suites. They conclude that both types of reuse. spatial and temporal. happen mostly intra-nest. while inter-nest reuse is mostly temporal reuse. They also observe that both capacity and conflict misses happen. although. not very often for the Perfect Benchmarks due to their small working-set size. Also conflict misses happen mostly intra-nest. while capacity misses happen mostly inter-nest.

Another conclusion is that many memory references within numerical codes are uniformly generated and most spatial locality is exploited with stride one. This observation is in concordance with our own observations. and we present in Table 6.1 a summary of the loops in the SPECfp95 benchmarks. From these results we conclude that for about 75% of the loops in the suite. the com-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

piler can compute dependence distance vectors. and most distances have value 1. This behavior was also observed by Petersen [50]. The results are somewhat in contrast with those reported in [62. 41] mainly because our data dependence test based on the Omega library can handle symbolic subscripts. and thus. it reduces considerably the number of "unknown" variables.

2.4 Compilation Using Performance Hints

The work presented in this section is not directly related to ours. but it underlines the need for compile-time models for performance prediction to drive compiler optimizations.

Haghighat and Polychronopoulos [32. 31] present one of the first approaches to use symbolic analysis inside the compiler to predict loop execution time. They show how by using performace data the compiler can generate better schedules for parallel loops. The new scheduling scheme. *balanced chunk scheduling* uses the compile time estimation of the execution time of an iteration to balance the work executed by each processor. Since each processor executes consecutive iterations (chunks) it benefits from increased locality. The scheme is shown to outperform other loop scheduling techniques because it both balances the work and exploits locality.

Wolf. Maydan and Chen [74] present the design and implementation (inside the MIPSpro compiler) of a compiler algorithm that applies loop permutation. outer unrolling. tiling. fission and fusion taking into account cache behavior. instruction scheduling and register allocation. They enumerate the search space of all possible transformations. selecting the set of transformations that are estimated to give the best possible overall performance. Their transformation algorithm depends upon having an evaluation function that can estimate how many cycles a given (possibly transformed) loop nest will take to run on the target machine. The estimation function combines estimates from two models. one for the processor and the other for the cache.

The processor model estimates three types of constraints: computational resources. latencies and registers. To estimate the computational resource needs they count the number of operations at high level. by walking the abstract syntax tree. and ignore operation dependences and common subexpressions. The processor model also allows for multiple functional units. To model latencies. an operation dependence graph is constructed. and algorithms for software pipelining are used to estimate the number of cycles.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The cache model has two tasks. to select a good tile size and to compute the loop overhead introduced by tiling. The model computes a formula for the loop cost. in cycles per iteration. of the tiling transformation. as a function of the unknown tile sizes. It then attempts to minimize this function. The model sorts the references into uniformly generated sets. and computes a footprint (the number of bytes in the cache used by the reference or set of references) for all the sets. It aggregates the footprints for the sets into footprints for each loop nest.

The goal of iterative compilation. Kisuki et. al [38]. is to construct a search space consisting of permutations of different optimizations and trying to find a minimum in this optimizations space. The search space can grow very large since it includes as separate optimizations variations of the same optimization with different parameters. for example. tiling with different tile sizes. The process of finding the minimum consists of a grid-based search algorithm (in order to reduce the number of points that need to be checked) that applies the set of optimizations at a search point. runs the program. collects the results. and decides which points to search next. While promising. this solution has two major drawbacks: first. it is very time consuming. The larger the number of optimizations. the larger the search space. and the number of points for which the program needs to be executed. The second drawback is that one can optimize codes only for the specific machine on which this compiler/optimizer runs. since different architectures have different characteristics that can impact the performance.

ATLAS [71] presents another approach. In this system. a set of linear algebra routines is optimized at installation time. by selecting the best parameters for the machine on which the code will run. ATLAS constructs a search space for tiling parameters based on cache parameters hints. or alternatively. if no hints are available. a covering range. It then compiles the code and runs it. measuring its performance. The best performing tiling parameters are then integrated in the library installed on the system. While the installation process can take hours or days. the code is highly tuned to that specific machine. and thus. any software that uses routines in the package will benefit from the performance of the building blocks. This method could be appropriate for building optimized libraries. but not necessarily for optimizing general purpose code.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3

Compile-time Performance Prediction

The problem of predicting program performance at compile-time is inherently difficult. There are many factors that make this problem hard. First. critical information needed by the compiler often depends on the input data of the program. Second. modern architectures are implemented so that the hardware optimizes execution using different techniques. such as exploiting instruction level parallelism (ILP). out-of-order execution. instructions and data caching. etc. A performance prediction model needs to consider all these techniques for an accurate estimate of actual performance. Multiprocessor systems add another dimension because of data distribution and communication between processors. Third. there is the issue of the low-level optimizations performed by the compiler. Typical optimizations are code scheduling to exploit ILP and register allocation. A compile-time performance predictor is usually invoked much earlier than the code generation phase. therefore it needs to either implement or estimate the low-level optimizations. Next. there is the problem of prediction accuracy. If the predictor approximates a piece of code. and uses that value to predict a larger chunk of code. compounding the estimates may magnify the error significantly. Another problem is cross-machine prediction: we envision our system being used to compare different systems. It is not always possible to have access to all the machines for which one wants the evaluation because some of the machines may not exist. Therefore. it is desirable for the prediction system to allow for machine independent prediction with the possibility to customize it for architectural parameters.

Many different strategies have been tried to address all these problems. such as: using heuristics [4]. profiling [77. 61]. run-time measurements [22. 3], analytical models [23. 69. 12] and combinations of these. The results have been mixed. showing that much work is still needed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In this chapter we present our approach, which represents performance data symbolically in the form of expressions containing variables for machine parameters, operation counts and input data values. The compiler synthesizes performance expressions and instruments the code to extract values unknown at compile-time. Our symbolic expressions decompose the overall performance into four parts: CPU, memory, communication and I/O. The total execution time is represented as:

$$T_{total} = T_{CPU} + T_{MEM} + T_{COMM} + T_{I/O} \quad (3.1)$$

where T_{CPU} is the computation time spent by the processor itself, T_{MEM} is the time spent accessing the memory hierarchy, T_{COMM} is the interprocess/thread communication time, and $T_{I/O}$ is the time spent doing I/O. In this work we shall model the first two terms only.

Each term in Equation (3.1) consists of a symbolic expression, i.e., a mathematical formula expressed in terms of program input values and perhaps some profiling information, such as branch frequencies. The expression involves parameters representing characteristics of the target machine and thus, is a function of the source code, the input data and the target machine.

To estimate the execution time of a program, we start by estimating the execution time of each basic block. The symbolic expressions obtained are aggregated into expressions for compound statements.

The problems mentioned above are addressed as follows:

- *missing information at compile-time* – the prediction system models unknown values as symbolic variables. The performance can be expressed either symbolically, or if the execution time is desired as a precise value, the variables can be substituted with values obtained by profiling.

- *portability across machines* – hardware parameters are represented as variables in the symbolic expressions. There are some assumptions made about the organization of the target machine, such as the number and type of the functional units, the size and number of levels of caches, but the actual details are represented symbolically and evaluated on demand, based on a machine description file.

- *compiler low-level optimizations* – to address this problem we use heuristics, explained later

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in this chapter.

In the following sections we detail the prediction models for the processor and the memory hierarchy.

3.1 CPU Prediction

In this section we describe the compile-time model of the processor. T_{CPU} in Equation (3.1) estimates the time spent by the processor doing computation. We assume a superscalar processor that is capable of issuing and executing several operations per cycle. We also assume that all the memory load and store operations are cache hits. The time to access the memory hierarchy is estimated separately. and we shall detail the modeling of cache accesses in Section 3.2.

The compiler counts the number of operations in the high level language code. These operations include: integer arithmetic and logical operations. floating point operations. and load and store operations assuming no cache misses. In addition. it considers as basic operations Fortran intrinsic functions. such as square root (many current processors have functional units that execute square root operations. and can be estimated using micro-benchmarking otherwise) and trigonometric functions. We also consider as basic operations function calls and loop overheads. thus taking into account the cost of branching operations and bookkeeping operations such as parameter passing and loop index testing.

The prediction is expressed as a symbolic expression of the form:

$$T_{CPU} = CycleTime \times \sum_{i=1}^{\#groups} (count_i \times cost_i). \tag{3.2}$$

where $count_i$ are symbolic expressions representing the number of operations in group i (we explain the groups of operations shortly). and $cost_i$ represents the hardware cost for the operations in group i. The hardware costs. $cost_i$. can be obtained either from the processor's manual. design specifications. or by using microbenchmarking [60]. The latter is usually the most convenient way to get the values associated with intrinsic functions and loop overheads if the machine is available. For the experimental results presented in Chapter 6 we actually use both the processor manuals and micro-benchmarking.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

No	Group Name	Operations	$cost_i$ (cycles)	
			R10000	Ultra IIi
1	Integer Add	Integer addition and subtraction	0.5	1
2	Integer Mult	Integer multiplication	6	18
3	Integer Div	Integer Division	35	37
4	Fl. Add	Single precision addition, subtraction and multiplication	1	3
5	Fl. Div	Single precision division	14	12
6	Dbl. Add	Double precision addition, subtraction and multiplication	1	3
7	Dbl. Div	Double precision division	21	22
8	Sqrt	Square root	27	25
9	Trig	Trigonometric operations	60	80
10	Intrinsic	Minumum, maximum, absolute value, etc.	1.5	9
11	Fun. Call	Function calls	1	5
12	Loop ovhd	Includes increment and branch and compare	41	18
13	Scalar load	Integer and single precision load	1	1
14	Scalar store	Integer and single precision store	1	1
15	Dbl. load	Double precision load	1	1
16	Dbl. store	Double precision store	1	1
17	Array load	One dimensional array load (includes index computation)	5	5
18	Array store	One dimensional array store (includes index computation)	5	5
19	N dim array load	Multidimensional array load	10	10
20	N dim array store	Multidimensional array store	10	10

Table 3.1: Operation groupings

To reduce the number of independent variables in the symbolic expressions, operations are grouped into sets based on the operation type and the data size on which they operate. For example, for the machines considered in this work, we group together single precision addition and multiplication since, on most current architectures, these instructions have similar latencies being executed in the same or identical functional units. We distinguish between multiplication and division since the division operation usually has longer latency than multiplication. For other processors, the groupings could be adapted, but we consider the groups presented in Table 3.1 as a reasonable base-line for current processors. We have used this grouping in a prototype that models the MIPS R10000 and the UltraSparc IIi processors. The table enumerates the groups and for each group presents the latencies that we used in our predictions for the two processors.

Using simple symbolic arithmetic, the expressions for basic blocks are combined to generate the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cost of operations for each statement or block of statements. For example. consider a loop of the form:

```
DO i = 1, m
  DO j = 1, n
    S1
    S2
  ENDDO
ENDDO
```

After we estimate the cost for S1 and S2. C_{S1} and C_{S2}. we estimate the cost for loop j as follows:

$$C_{DO_j} = n \times (Loop\ ovhd + C_{S1} + C_{S2})$$

The cost for loop i is:

$$C_{DO_i} = m \times (Loop\ ovhd + C_{DO_j})$$

Consider another example. an IF statement of the form:

```
IF cond THEN
  S1
ELSE
  S2
ENDIF
```

the cost is:

$$C_{IF} = Branch\ ovhd + C_{cond} + C_{S1} \times freq(S1) + C_{S2} \times freq(S2)$$

If the branch frequencies $freq(S1)$ and $freq(S2)$ are known at compile time (through profiling information or user annotations). the values can be substituted. Otherwise the symbolic values are carried in the prediction expressions.

Using this method. the cost expressions for different levels of granularity in the program (blocks

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of statements. loops. procedures) are combined until a unique expression could be generated for the entire program.

Although this is a very simple strategy. it has proven reasonably accurate when no compiler optimizations are applied. as can be seen in the experimental results presented in Chapter 6.

In order to accurately predict the performance for optimized codes we have to apply. or at least approximate in our model. the low-level optimizations performed by the native compiler. We have chosen to approximate these optimizations by using heuristics applied at high level source code. We found that the following heuristics approximate best the optimizations performed on the set of benchmarks that we studied:

- *Eliminate loop invariants.* This is a simple optimization applied by all optimizing compilers and it can be done at high level.

- *Consider only the floating point operations.* Based on the observation that. in scientific codes. the useful computation is done in floating point and in optimized code integer operations are used mostly for control flow and index computation. we assume that superscalar processors can overlap the cost of index computation with the floating point operations. We take into account the control flow operations (branching) in the form of loop overheads.

- *Ignore all memory accesses that are not array references.* The reason for this heuristic is that scalar references occur infrequently in scientific codes and. if they do. modern processors often have enough registers to buffer them.

- *Overlap operations.* For multiple issue architectures with multiple functional units. we must allow operations in different categories to overlap execution. For example. on the MIPS R10000 processor. there can be 4 instructions issued in one cycle chosen among: 2 integer operations. 2 floating point operations. 1 memory operation or 1 branch.

Note however that these optimizations are mostly suited for scientific codes. in which. most of the computation is done in loops accessing arrays and executing floating point operations. To accurately model optimizations done for integer codes more research is needed.

Using these approximations we obtain a lower bound on the processor's execution time. We tend to underestimate the execution time in the processor since we consider all the operations inde-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

pendent. and therefore we "exploit" more instruction level parallelism that may actually be present in the code. However. we do not consider other low-level optimizations. such as loop unrolling and register allocation. nor we consider hardware reordering of the operations. These optimizations usually reduce the number of operations and/or increase the potential for ILP. However it is both very difficult and too machine and compiler specific. to consider these optimizations at high level language code.

The model could be improved by using an operations dependence graph. that takes into consideration dependences between operations to compute the overlapping. However. such a model will have increased complexity. up to a point where the predictor duplicates the code scheduler from the compiler. We have compromised some accuracy for the simplicity of the model.

Two main characteristics set this model apart from other related work: the machine independence and the compiler independence. The machine independence is realized by using symbolic expressions to represent hardware costs for groups of operations. This is opposed to the method used by Balasundaram et al. [3] and Fahringer [22] of measuring kernels. and trying to match the code to the kernels. The compiler independence is achieved by using heuristics to approximate the low-level optimizations that could be applied by the compiler. This is in contrast to the approach used by Saavedra and Smith [59] in which they tried to account for the compiler low-level optimizations in the hardware costs of the operations. It is also different from Wang's approach [69]. in which the predictor must have access to the compiler's low-level optimizations. Both these methods need to be reimplemented when the undelying compiler changes. however in some cases they can be more precise than ours.

3.2 Memory Hierarchy Prediction

The term T_{MEM} in Equation (3.1) estimates the time spent accessing memory locations in the memory hierarchy. As we mentioned before. when estimating the execution time of basic operations we assume all memory references are cache hits in the first level cache. However. many accesses are not served from the first level cache. in part because applications have data sets much larger than the cache.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

T_{MEM} can be expressed as follows:

$$T_{MEM} = CycleTime \times \sum_{i}^{\#levels} (M_i \times C_i). \tag{3.3}$$

where M_i represents the number of accesses that miss in the ith level of the memory hierarchy, and C_i represents the penalty (in machine cycles) for a miss in the ith level of the memory hierarchy. C_i is computed using micro-benchmarking, as in [58]. We briefly discuss the cache micro-benchmarking here.

The micro-benchmarks (narrow spectrum of benchmarks) are a set of experiments used to measure memory hierarchy characteristics and performance. In particular, they measure primary and secondary cache characteristics and the TLB for a uniprocessor. Each experiment measures the average time per iteration required to read, modify, and write a subset of elements belonging to an array of known size. The number of misses will be a function of the size of the array and the stride between the array element accessed. From the number of references and the number of misses, as the stride and the size of the array are varied, we can compute the relevant memory hierarchy parameters, including the cache size, the cache line size, the time needed to satisfy a cache miss, and the associativity. For example, assume a machine that has a cache with a C 4-byte words size, a cache line size of b words, and an associativity a. Furthermore, consider a one-dimensional array of size N 4-byte elements. A subset of the array elements is accessed in a loop that contains a simple floating-point operation. Each subset is generated by traversing the array with a certain stride. Therefore, each experiment is characterized by N (the array size) and by s (the stride). After plotting all the experiments on a graph, we can determine different regimes, from which the unknown parameters of the cache are derived.

Once the cache parameters are defined and the number of cache misses for a loop is estimated, we can translate the number of misses in execution time. In the remainder of this chapter we propose two models for estimating the number of cache misses at compile-time. Depending on the amount of compile-time information we estimate the number of cache misses using an accurate model, the Stack Distances Model (Section 3.3), or, if not enough data dependence information is available, we estimate using the Indirect Access Model (Section 3.4).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

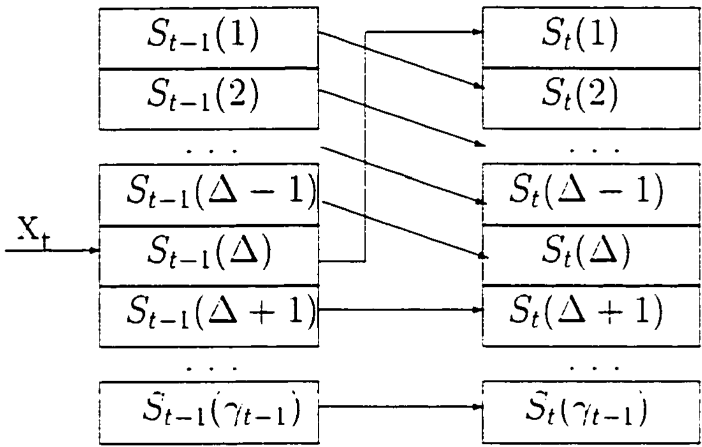


Figure 3.1: Stack update when the currently referenced location has been previously accessed

3.3 The Stack Distances Algorithm

The Stack Distances Model (SDM) is based on the stack processing algorithm. The classical stack processing algorithm [43]. generates a stack histogram for a program by analyzing a trace of the memory references. The trace can be analyzed either off-line. after the program has finished executing. or on-the-fly - during the program execution. Or. when enough information is known at compile time. such as all the data dependence distance vectors. affine array subscripts. etc.. we propose to generate the stack histogram at compile time.

3.3.1 Introduction

The stack processing algorithm takes a trace of memory references. cache line references or virtual page references in a program. and builds a stack as follows: if a memory location has been previously referenced (stack hit). we record the distance. Δ. from the top of the stack to the position at which the reference is found. and move the reference on the top of the stack. all the references between the top and position Δ being pushed down one position. The references below the current reference position are not affected. as shown in Figure 3.1. If the reference is the first access to that memory location. we let the stack distance $\Delta = \infty$ and push the reference on the stack. using a normal push operation.

The result of the stack processing algorithm is a histogram that counts the number of accesses for all stack distances. Figure 3.2 shows a histogram computed in this way.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This histogram can be used to calculate the number of out-of-core page references, or equivalently, the number of cache misses, for any memory or cache size. For a physical memory of size C, all the accesses at stack distances of less than C are in-core (Equation (3.4)), and all the others accesses are out-of-core (Equation (3.5)). Splitting the stack depth histogram at C, the area under the histogram curve at the left of C is the number of in-core references, whereas the area to the right of C represents the out-of-core accesses.

$$H_i(C) = \sum_{\delta=0}^{C} s(\delta) \tag{3.4}$$

$$M_i(C) = \sum_{\delta=C+1}^{\infty} s(\delta) \tag{3.5}$$

where δ is the stack distance, and $s(\delta)$ is the number of references at stack distance δ.

In the same way, the stack histogram can be used to predict the number of cache hits and cache misses that occur in a loop nest. In order to generate the stack histogram at compile-time, we must compute two things: the stack distances at which references occur, (all the points on the x axis of the stack histogram), and the number of references that occur for each stack distance (the points on the y axis of the stack histogram).

Before going into details, we discuss the design choices and the limitations of the current implementation of the algorithm.

As we mentioned before, we focus on scientific programs, therefore we consider for inclusion in the stack histogram only references to array elements. Again, the motivation is that, in scientific codes, scalar variables are mostly used for indexing and thus reside in registers. We consider only array references with affine subscripts for two reasons: first, the Omega test (the data dependence test that we use) works only on affine subscripts, for all other subscritps it assumes that the dependence exists; and second, our algorithm for computing array sections (presented in Section 3.3.5) can handle only affine subscripts. However, as shown in [50] for the Perfect Club benchmarks and from our own study of the SPECfp95 benchmarks, affine subscripts constitute more than 80% of the subscripts in these suites. The other most common form is subscripts of subscripts, i.e., the subscript expression is another array element reference, which occur mostly in sparse matrix operations. We handle this type of subscripts using the Indirect Accesses Model (see Section 3.4).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

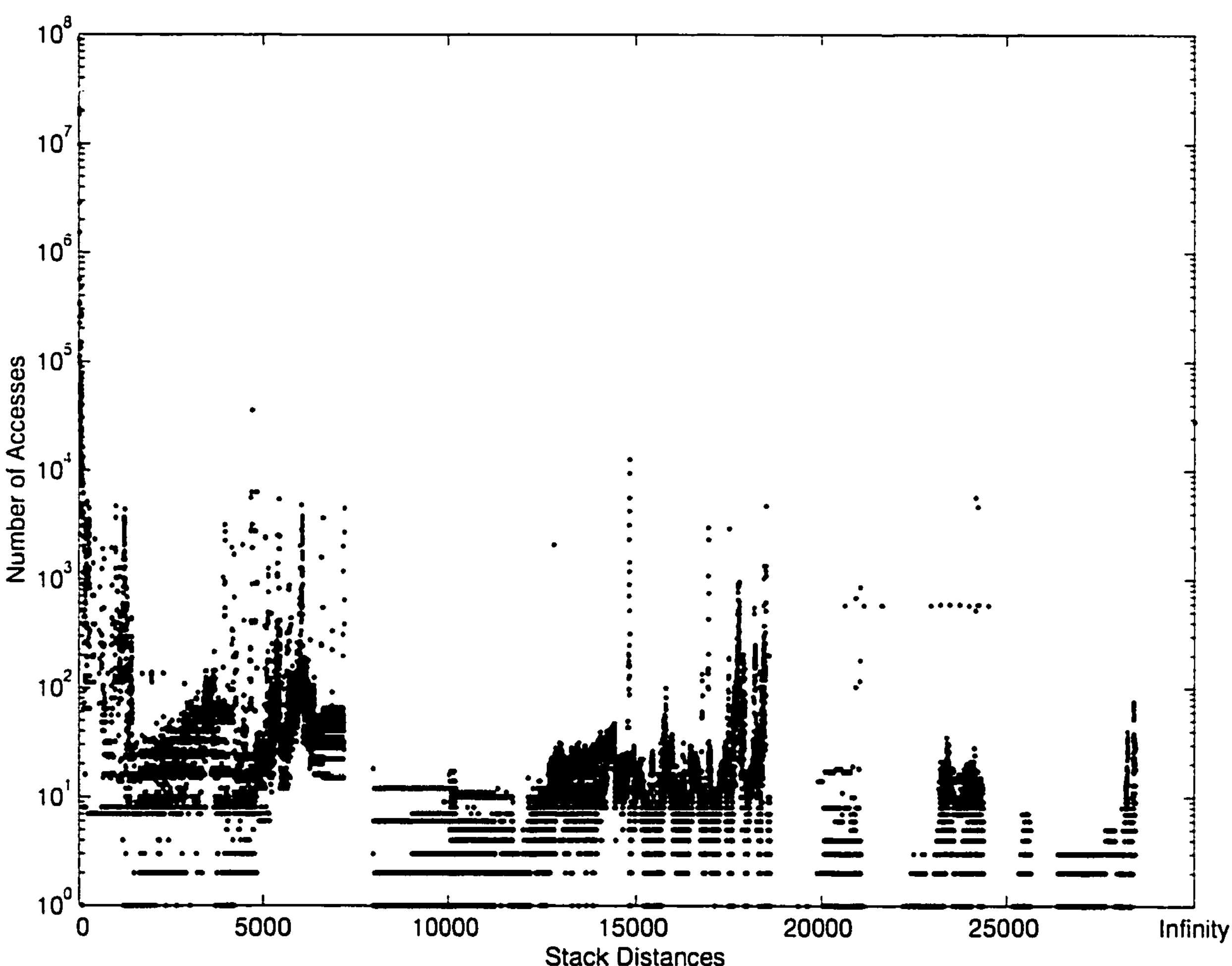


Figure 3.2: Stack histogram for QCD

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Although the stack histogram can be used to predict the number of cache misses at different granularities in a program, i.e. loop nests, routines, etc., the compile-time algorithm presented in this work generates stack histograms for loop nests only. We consider only nests for which the data dependences are uniformly generated [28], that is, the distance vectors are defined and constant. We could apply the algorithm for dependences where the distance vectors have a lower bound, by considering the minimum distance in our calculations, however, the estimation will no longer be totally accurate. Depending on the use of the estimation, this loss of accuracy can be tolerated, and the algorithm applied with success. Further research needs to be done to enable the estimation on multiple loop nests.

The compile-time stack algorithm can estimate the number of misses for fully associative caches with the LRU replacement policy. The replacement policy constraint is actually a constraint imposed by the stack processing method, in order to satisfy the inclusion property (the stacks for memories of size C or lower are included in the stack for memory of size $C + 1$). We also consider each loop nest to start with a cold cache, i.e., none of the array elements accessed in the loop are present in the cache when the loop starts.

In explaining the algorithm we will consider cache lines of one array element in order to keep the algorithms "simple". We remove this restriction in Section 3.3.8. Also, in presenting the algorithms we assume that the loops are normalized, i.e., they have the step equal to 1. This restriction is just to keep the equations simpler, and the implementation supports loop increments different from 1.

3.3.2 Algorithm Overview

The stack distance is, by definition, equal to the number of distinct memory locations accessed between two references to the same memory location, or ∞ if there is no previous reference to the memory location. Inside the compiler, the fact that two references access the same memory location is represented by a data dependence (including input dependences). Therefore, we want to compute for each dependence the number of distinct array elements accessed between the source and the target of the dependence (we call this the number of distinct array elements spanned by the dependence). However, a dependence can span different numbers of distincat array elements depending of the iteration point in which the target is accessed. This follows from the fact that a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

static array reference may access a distinct array element or not. depending on the other incoming dependences on the array reference for the particular iteration point in which the dependence is considered.

The compile-time algorithm for computing stack distances has the following major steps:

- *iteration space partitioning* - the iteration space is partitioned according to which dependences are legal at each iteration point (Section 3.3.3). Within a partition all the references will have the same set of valid dependences.
- *dependence span computation* - for each incoming dependence. in each partition. we compute the iteration points are executed between the source and the target of the dependence (Section 3.3.4)
- *array sections computation* - we compute for each dependence and for each array reference. the number of distinct elements in the array. accessed between the source and the target of the dependence. We also compute the sum over of distinct elements over all array references and use the sum to label the dependence (Section 3.3.5)
- *stack histogram computation* - the stack histogram is computed using previously determined information (Section 3.3.6)

Note that iteration space partitions. dependence spans and array sections are all sets of integral elements. we call them regions. It is desirable to use a common representation for all these regions. so that we can optimize the algorithms that operate on them. The following operations need to be defined on the regions: union. intersection. difference and projection. Union. intersection and difference are the usual set operations. Projection is the operation that maps an iteration vector to the array element accessed in the iteration. The mapping is subject to the array indexing function.

We had several options available to represent the regions. Below we discuss some of the advantages and disadvantages of each notation. and our chosen method based on the triplet notation.

There is a substantial amount of work that has been done for representing array sections. Four major directions have evolved for representing subsets of array elements: linear constraint based polytopes [53. 54. 19. 20. 16]. reference lists [10. 40]. triplets [11. 34] and linear memory access descriptors (LMADs) [36].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The linear constraint-based techniques are very powerful. general. and the most accurate notation. However. by using Fourier-Motzkin elimination to solve the linear inequalities system. there are several drawbacks: first. the theoretical complexity of the problem is exponential. and second. it requires that the linear inequalities form a convex hull. forcing a loss of accuracy when some regions must be widened to make them convex. The method also reports all the solutions. not only the integer ones. We are studying the posibility of using Ehrhart polynomials [15]. which generates the set of integer solutions for a linear system. as a more accurate and faster technique to replace our current implementation.

The reference list techniques rely on linearizing the arrays and making a list to represent each individual array reference in a code section. This method was not designed to summarize array access information. and therefore. is very cumbersome to use for our purpose.

The triplet notation is a simple representation for a set of integer values for each dimension. which start at a lower bound and proceed to the upper bound via a stride. Each dimension is represented by a triplet. $[l : s : u]$. where l. s and u represent the lower bound. stride. and upper bound of the array section. Triplets representation of array sections is very popular. although there are instances. such as array accesses in triangular loops. or some coupled subscripts in which the array sections lose some accuracy. We have implemented our representation based on triplet notation because of its simplicity. We extended the notation to cover for some of the drawbacks. such as allowing the loop index in the stride expression. However. there are cases in which this notation in not totally accurate. and we will discuss some of these cases in Section 3.3.5.

Linear Memory Access Descriptors (LMADs) combine a generalized triplet notation with constraints. At the time of this work the LMADs were not fully developed. and more research needs to be done to use them in this work.

We present now the steps of the stack distances algorithm in more detail.

3.3.3 Iteration Space Partitioning

In this section we present the iteration space partitioning algorithm. The iteration space is partitioned into regions for which the dependence spans are the same for each dependence at all the iteration points in the partition. This allows us to reduce the number of points at which dependence

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Input:

The bounded iteration space for a loop nest of depth k $IS = \prod_{i=1}^{k}[L_i.U_i]$.
where L_i and U_i are the lower and upper bounds of the ith nested loop. respectively
The data dependence graph with distance vectors computed

Output:

The partitioned iteration space $PartIS$. such that in each partition.
all incoming dependences are the same.

Method:

set $PartIS = \{IS\}$
foreach dependence δ compute the valid space $VS(\delta)$ as follows:
  if δ is loop independent **then**
    $VS(\delta) = IS$
  if δ is loop carried with distance vector $d = (d_1.d_2.\ldots.d_k)$ **then**

$$VS(\delta) = \prod_{i=1}^{k} \begin{cases} [L_i + d_i.U_i] & \text{if } d_i > 0 \\ [L_i.U_i - d_i] & \text{if } d_i < 0 \end{cases}$$

  $temp = \emptyset$
  foreach $x \in PartIS$
    $temp = temp \cup \{x - (x \cap VS(\delta)\} \cup \{x \cap VS(\delta)\}$
  end foreach
  $PartIS = temp$
end foreach

Figure 3.3: Iteration space partitioning algorithm

spans are computed. because we compute one span per partition for each dependence. as opposed to computing one span per iteration point. In general. we do not know how many iterations are in a loop. except symbolically. Therefore. we partition the iteration space. such that for all the iteration points in one partition. all the array references have exactly the same incoming dependences. For example. if there is a loop carried dependence on an array reference with a positive distance d. the array elements accessed in the first d iterations of the loop will not have that incoming dependence. therefore the iterations in which these elements are accessed will be in a separate partition from the rest of the iterations in the loop. After partitioning the iteration space. we compute one array section for each array reference and each dependence that spans the reference in each partition. The partitioning algorithm is presented in Figure 3.3.

The input to the algorithm consists of the iteration space IS. which is a polytope that contains one point for each iteration of the nest. We express the iteration space as the cartesian product of the integer intervals $[L_i.U_i]$ (also called *domains* in literature). where L_i and U_i are the lower

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

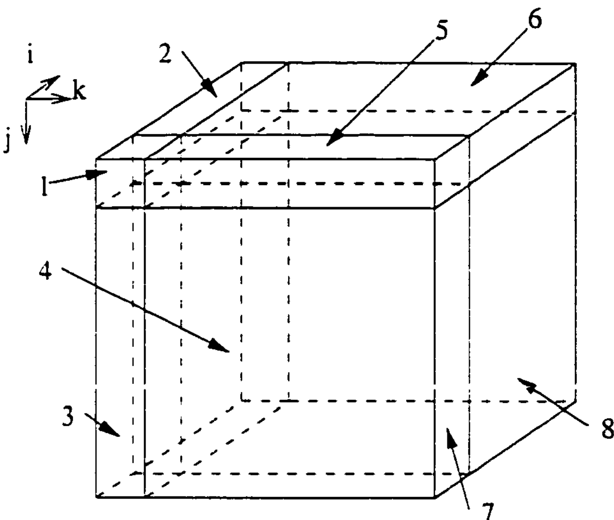


Figure 3.4: Partitioned iteration space for matrix multiplication.

and upper bounds of the ith nested loop respectively. Another input to the partitioning algorithm is the data dependence graph, with all the distance vectors computed. Assumming a loop carried dependence with distance d, that is carried by only one loop, the algorithm splits the iteration space into two partitions, one that contains all the iterations from L to $L+d$, and one that contains the other iterations, from $L+d$ to U. If the distance d is negative, the partitions are from $U-d$ to U and from L to $U-d$. The second partition in both examples is called the valid space of the dependence, because only the array elements referenced in the iterations contained in this partition have this incoming dependence. If there are multiple loops that carry a dependence, the same operation of splitting the iteration space is performed for each loop. There will be two partitions for each dependence, only the partitions will no longer be rectangular.

In general, the number of partitions is less than 2^N, where N is the number of loop-carried dependences in the loop nest, because some dependences may generate the same partitions. In Figure 3.4 we show the partitions for the matrix multiplication code. There are 6 loop carried dependences in this loop nest, but only three of them generate distinct partitions, and thus there are 8 partitions.

Note that all the loop bounds and iteration space partitions are expressed symbolically. However, we require that the dependences are uniformly generated, i.e., all the elements of the distance vectors to be constant. We'll show later, that, even when the dependence distance vectors are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

not constant but a lower bound can be computed, we can use the lower bound to approximate the distance and usually obtain a reasonable estimation for the number of distinct array elements accessed.

Next we need to compute the iteration points spanned by each dependence in each partition, and thus, the number of array elements spanned by each dependence. In the following discusion we'll present our examples on the whole iteration space, since considering the partitions will just complicate the figures. However, the reader should keep in mind that we do these computations for every partition in the iteration space.

3.3.4 Dependence Spans

For each dependence we need to compute the number of distinct array elements that are accessed between the source and the target of the dependence. We can do this by determining which iterations are executed between the source and the target of the dependence, and taking the union of all array elements accessed in those iterations.

We define the *dependence span* as being the set of iteration points between the source iteration and the target iteration of the dependence. Geometrically, the dependence span is a shape in the iteration space that encloses all these iteration points. For example, the shaded region in Figure 3.5 represents the dependence span defined by the input dependence on reference `B(k,j)` (shown in Figure 3.11). This dependence is carried by the outermost loop of a three-nested loop with distance 1. The iteration points spanned by this dependence are: the remaining iterations of loop `k` in the same iteration of `i` and `j`, i.e. $[i, j, k : n]$, the remaining iterations of loop `j` in the same iteration of loop `i`, which includes all the iteration of loop `k`, $[i, j+1 : n, 1 : n]$, the iterations previous to iteration `j` in the iteration `i+1`, $[i+1, 1 : j-1, 1 : n]$, and the iterations previous to iteration `k`, $[i+1, j, 1 : k]$.

Dependence spans are computed using the algorithm presented in Figure 3.6.

The algorithm takes a dependence δ that has the source iteration $(I_1, \ldots, I_n)$ and the target iteration $(I_1 + d_1, \ldots, I_n + d_n)$, where d_i are elements of the distance vector $D(\delta) = (d_1, \ldots, d_n)$. It computes the set of iterations spanned by the dependence in three steps. First, it completes all remaining iterations for the loops enclosed by the outermost loop carrying the dependence. Next,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

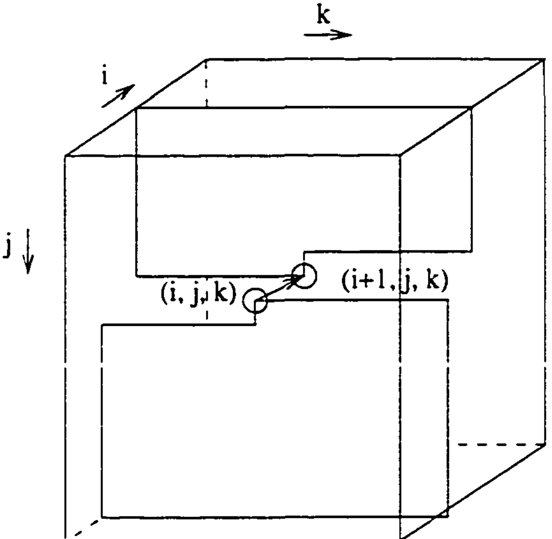


Figure 3.5: Iteration space for matrix multiplication. The shaded shape represents the iterations spanned by loop-carried dependence with distance 1 in dimension i.

Input:
A loop nest L of depth n and its iteration space
An incoming dependence δ with source iteration $(I_1, \ldots, I_n)$ and
dependence distance vector $D(\delta) = (d_1, d_2, \ldots, d_n)$

Output:
The dependence span $DS(\delta)$ for the dependence

Method:
$DS(\delta) = \phi$
let l be the outermost loop carrying the dependence in L
/* collect the all iterations up to the next iteration of l */
foreach loop i starting from the innermost loop to l
$DS(\delta) = DS(\delta) \cup (I_1, I_2, \ldots, I_i, I_{i+1} : U_{i+1}, L_{i+2} : U_{i+2}, \ldots, L_n : U_n)$
end foreach

/* collect all the iterations up the the target iteration */
foreach loop i starting from l to the innermost
if $I_i + d_i - 1 \leq I_i + 1$ **then**
$DS(\delta) = DS(\delta) \cup (I_1, I_2, \ldots, I_{i-1}, I_i + d_i, L_{i+1} : I_{i+1} + d_{i+1}, L_{i+2} : U_{i+2}, \ldots, L_n : U_n)$
else
$DS(\delta) = DS(\delta) \cup (I_1, I_2, \ldots, I_{i-1}, I_i + 1 : I_i + d_i - 1, L_{i+1} : U_{i+1}, \ldots, L_n : U_n) \cup$
$(I_1, I_2, \ldots, I_{i-1}, I_i + d_i, L_{i+1} : I_{i+1} + d_{i+1}, L_{i+2} : U_{i+2}, \ldots, L_n : U_n)$
end if
end foreach

Figure 3.6: Dependence span computation algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

it collects all the iterations in the loop carrying the dependence, up to the target iteration. And last, it collects the iterations of the loops enclosed by the dependence carrying loop up to their respective target iterations. Note that the second step is performed only if the dependence distance of the carrying loop is greater than the step of the loop. In the algorithm, the last two steps are merged.

In the next section we show how to use the dependence spans to compute the number of distinct array elements accessed in each iteration contained in the dependence span, and then, the total number of distinct array elements spanned by a dependence.

3.3.5 Array Sections Computation

Once the dependence spans are computed, they can be used to compute the array sections covered by the dependences. An *array section* is, by definition, the set of array elements that are accessed by all the iterations in a dependence span.

Intuitively, if we can identify every array element accessed in each iteration point contained in a dependence span, we can compute how many distinct array elements were accessed between the source iteration and the target iteration of the dependence. For each iteration we may have several memory accesses, one for each array reference in the body of the loop. By computing exactly which array element is accessed in each iteration, we can compute the array sections.

This intuition is illustrated in Figure 3.7. The outisde rectangles represent the arrays accessed in the matrix multiply loop nest (A, B, and C). The cube represents the iteration space, with the shaded region denoting the dependence span of the loop-carried input dependence on reference B. The shaded regions in each array region represent the array sections spanned by this dependence. Thus, from iteration (i, j, k) to iteration (i+1, j, k) the following array regions are accessed: $A(i:i+1, 1:n)$, $B(1:n, 1:n)$, and $C(i, j:n) \cup C(i+1, 1:j)$.

The array section, $AR(\delta)$, spanned by a dependence is computed by substituting in the array index functions the ranges of the induction variables taken from the dependence span. In geometrical terms, the dependence span is projected onto the array space, as illustrated in Figure 3.7. The algorithm in Figure 3.8 computes array sections.

We start the algorithm by considering that the entire array space is accessed. Then, for each

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

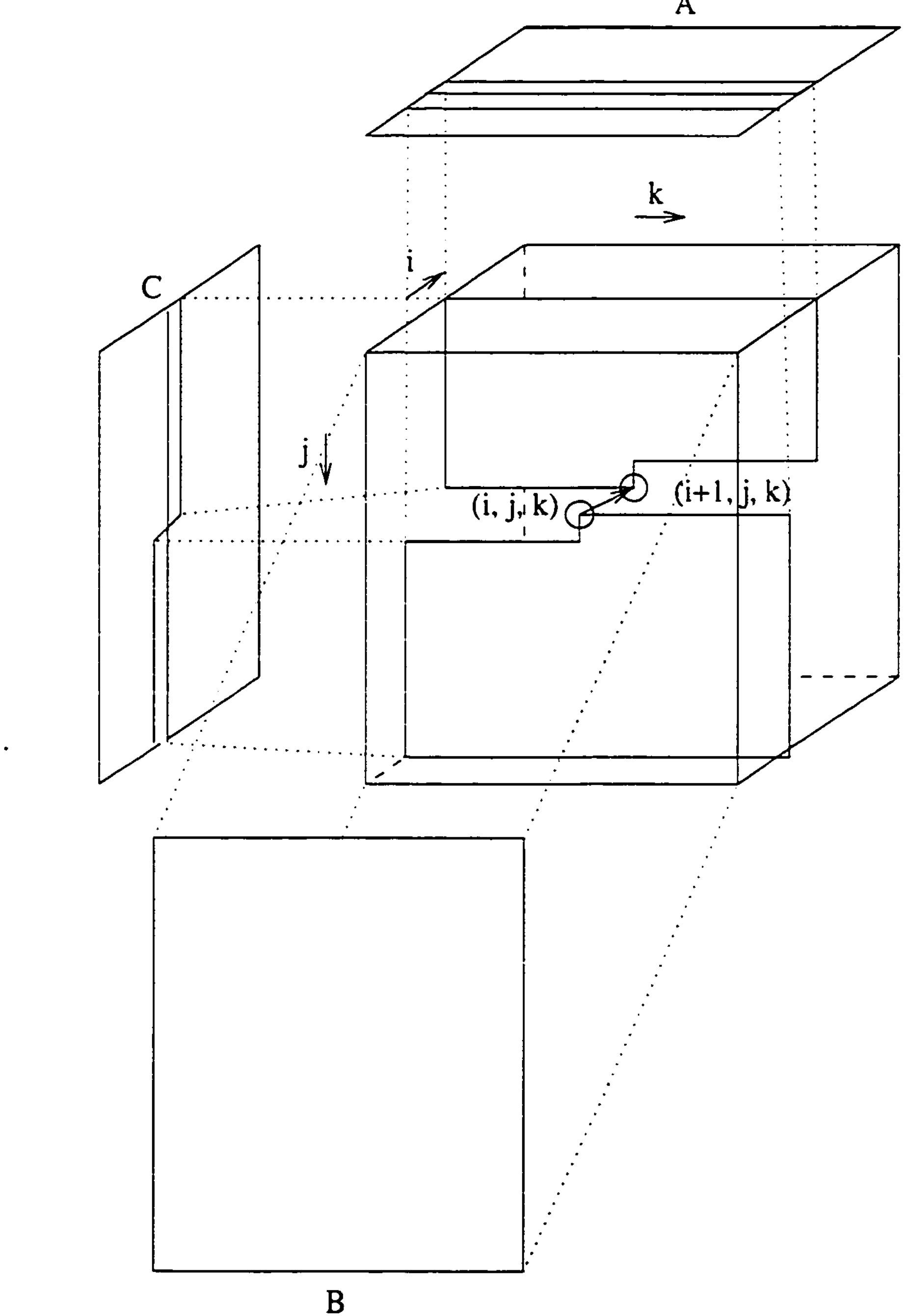


Figure 3.7: A dependence span projected onto array sections. For each array, the shaded areas show which elements contribute to the number of distinct accesses between two iterations of loop i.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Input:

A dependence δ with the dependence span $DS(\delta) = [l_1 : u_1, \ldots, l_m : u_m]$
in an m-level nested loop

An n-dimensional array reference $a(f_1(\vec{i}), \ldots, f_n(\vec{i}))$

Output:

The array section spanned by the dependence $AR(a, \delta)$

Method:

$AR(a, \delta) = [L_1 : U_1, \ldots, L_n : U_n]$ /* entire array space */

foreach $j = 1, n$ /* for each array dimension */

/* compute the extent in that dimension */

$x = \emptyset$

foreach $I_k \in f_j(\vec{i})$ /* for each loop index */

let $y = range_{DS(\delta)}(I_k) = [l_k : u_k]$

$x = x \uplus f_j\left(\vec{i}\,|_{I_k = y}\right)$

end foreach

$AR(a, \delta) = AR(a, \delta) \cap [L_1 : U_1, \ldots, L_{j-1} : U_{j-1}, x, L_{j+1} : U_{j+1}, \ldots, L_n : U_n]$

end foreach

The $\uplus$ operation is defined as follows ($expr$ is a constant or a loop invariant variable):

1. $expr + [l : s : u] = [expr + l : s : expr + u]$

2. $expr * [l : s : u] = [expr * l : expr * s : expr * u]$.

3. $$[l_1 : s_1 : u_1] + [l_2 : s_2 : u_3] = \begin{cases} [l_1 + l_2 : \gcd(s_1, s_2) : u_1 + u_2] & \text{if } s_1 \mid s_2 \vee s_2 \mid s_1 \\ \{l_1 + l_2 : \gcd(s_1, s_2) : u_1 + u_2, M\} & \text{otherwise.} \end{cases}$$

where $\{l : s : u, m\}$ denotes a non-contigous interval.

Figure 3.8: Array section computation algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

array dimension. we compute the range of the index in that dimension (the extent of the array in the dimension). by substituting the ranges of the loop index variables into the subscript expression. The ranges of the loop index variables are taken from the dependence span. because we are interested in those iterations that are spanned by the dependence. Interval arithmetic is used to compute the extents. The operation $\uplus$ in Figure 3.8 presents the operations supported when combining intervals. We also discuss these operations next.

The operations supported for computing the extents are as follows: interval addition and multiplication with a constant or a loop invariant variable. and addition of two intervals.

Rule 1 handles the interval addition with a constant or a loop independent variable. i.e.. the subscript expression has the form $i + expr$. where i is a loop index variable and $expr$ is either a constant or a loop invariant variable. Examples of such subscript functions are: $i+1$. $i-3$ or $i+c$. In this case the extent is the same as the range of variable i shifted by $expr$. Assuming the range of the loop index variable i is $[4 : N]$. the ranges of the above subscript functions are: $[5 : N + 1]$. $[1 : N - 3]$. and $[4 + c : N + c]$ respectively.

The second rule treats the multiplication of an interval with a constant. The subscript has the form $expr * i$. where i is a loop index variable and $expr$ is either a constant or a loop invariant variable. Examples of such subscripts are $2i$ or $c * i$. In this case the computed extent is an expansion of the original range. an interval that has boths bounds and the step multiplied with the expression. Thus. assuming the range of the loop index variable i is $[2 : N]$. the range for the subscript functions $2i$ and $c * i$ are $[4 : 2 : 2 * N]$ and [2*c:c:c*N] respectively.

Both addition and multiplication with a constant or loop invariant variable preserve the accuracy of the range.

The last rule handles the case of coupled subscripts. i.e.. subscripts in which two or more loop index variables occur in the same subscript expression. In general. coupled subscripts occur in less than 20% of the subscripts. as shown by Shen et al. [50] who studied a large number of benchmarks and kernels from scientific codes. Moreover. subscripts containing multiplications of loop index variables are even less frequent. and we have not encountered it in the SPECfp95 benchmarks. We treat only the case in which the loop index variables are added in the subscript expressions.

Rule 3 handles the case when two loop index variable expressions are added together. such as

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

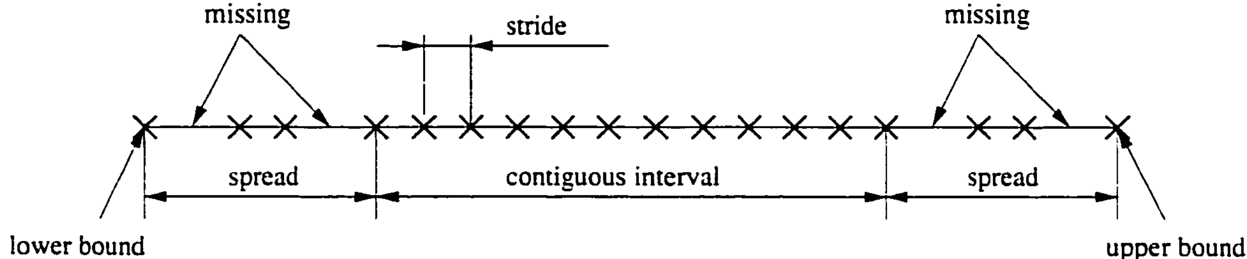
$2i + 3j + 10$. Note that the extent of this subscript expression can be computed by successive applications of the three rules. as follows: rule 2 to obtain the extent for the sub-expression $2i$. rule 2 for $3j$. rule 3 for $2i + 3j$ and rule 1 for $(2i + 3j) + 10$.

When adding two intervals there are two cases: (i) one of the strides is divisible with each the other and (ii) the strides are not divisible. When the strides are divisible the resulting extent is an interval with the bounds computed as the sum of the bounds of the two terms. and the stride is equal to the greatest common divisor of the strides. There are some cases in which the extent might have a different number of elements than it should. for example. when the upper bound of the inner loop is less than the upper bound of the outer loop. This is one of the limitations of the triplet notation. We hope to eliminate this inaccuracy by using a linear constraint-base representation for the regions.

The second case. when the strides are not divisible can produce even more inaccurate representations. To address this problem where we extended the triplet notation to handle "non-contiguous intervals". A non-contiguous interval (shown in Figure 3.9) is a set of integers that has a lower bound. an upper bound and a stride to traverse the elements. just like in triplet notation. Additionaly. there are elements at the two ends of the intervals that even if they are specified by the stride. they are not traversed. For these elements. we introduce a "missing" factor. that specifies how many elements are not actually traversed over a "spread" region at both ends of the interval. Note that a non-contiguous interval might not represent exactly which elements are accessed in the spread region. but since we are interested only in the number of elements that are accessed. the accuracy of a non-contiguous interval is sufficient for our purpose. To compute the missing elements and the spread. we have observed that if the strides are factored. the access pattern for the resulting interval is given by the access pattern of the greatest prime factors from each stride. Therefore the subroutine in Figure 3.9 enumerates the elements that are accessed for the prime factors. and uses that information to compute "missing" and "spread".

Whenever there are inaccuracies in computing the array sections. we mark the sections accordingly. The measure of accuracy propagates further in the cost model. such that. when the prediction expressions are evaluated. together with the performace figure we also provide a "confidence" measure for the prediction.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.




```
void computeNonContig(int icoef, int jcoef, int &missing, int &spread)
{
  int n = (icoef+jcoef)*(icoef+jcoef);
  int *a;

  a = (int *)calloc(n+(icoef+jcoef), sizeof(int));

  for(int i = 1; i <= icoef+jcoef; i++)
    for(int j = 1; j <= icoef+jcoef; j++)
      a[i*icoef + j*jcoef] = 1;

  int distElems = 0;
  for(int i = n; i >=0; i--)
    distElems += a[i];

  missing = ((n - (icoef+jcoef) + 1 - distElems)/2);
  int i = icoef+jcoef;
  while(missing > 0)
    if(a[i++] == 0) { missing--; }

  int firstContig = i;
  missing = ((n - (icoef+jcoef) + 1 - distElems)/2);
  spread = firstContig -(icoef+jcoef);
  free(a);
}
```

Figure 3.9: Non-contiguous intervals: representation and calculation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

One dependence might span more than one reference to the same array. In this case, the array section spanned by the dependence is the union of the array sections for each reference. Therefore two more operations are defined on intervals, union and intersection. These operations are well defined for the triplet notation. The number of distinct array elements spanned by a dependence is then computed by summing the size of array sections for all arrays referenced between the source and the target of the dependence.

$$AS(\delta) = \sum_{\text{distinct arrays } r} \left| \bigcup_{\text{all refs to } r} AR(r, \delta) \right|$$

Note that we must keep both the array sections for distinct arrays and for individual array references. The sections for the arrays are used to compute the stack distances, while the array sections for each reference are used to compute the number of dynamic references to a particular location. This computation is described in the next section.

3.3.6 Stack Histogram

Once the array sections are computed for each dependence span in each partition, the data required to compute the stack histogram is available. The stack histogram is composed of two sets of values, the stack distances and the number of accesses at that particular stack distance. Both these sets of values are computed symbolically, based on the array sections calculated in the previous section.

Each array reference contribution to a stack distance determined by its incoming dependences, or is ∞ if there are no incoming dependences. The number of accesses contributed by each array reference is determined by the number of dynamic executions of the reference. The algorithm to compute the stack histogram is shown in Figure 3.10.

For each partition of the iteration space we consider all the incoming dependences that are valid in the partition. We compute, for each array reference in the loop body, the minimum on the distinct number of array elements spanned by the incoming dependences, Δ, and we take this minimum as the stack distance. If there is no incoming dependence, all the accesses for that reference are "cold misses", i.e., happen at distance ∞. To compute the number of accesses at each distance, we compute how many times the statement that contains the reference is executed by

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Input:

A loop nest with the data dependence graph augmented with dependence spans and with array sections computed for each dependence

The partitioned iteration space $PartIS$ for the loop

Output:

The symbolic stack histogram S

Method:

foreach partition $p \in PartIS$

foreach static array reference r in the loop body

let $\Delta = \begin{cases} min_\delta(AS(\delta)) & \text{if } \exists\delta \text{ s.t. } target(\delta) = r \text{ and } \delta \text{ is valid in } p \\ \infty & \text{otherwise.} \end{cases}$

where $AS(\delta)$ represents the number of distinct array elements in the dependence span

$S(\Delta)+ = |p|$. Since each array reference is accessed in each iteration point. the size of the partition (the number of iteration points considered) gives us the number of dynamic accesses to r.

end foreach

end foreach

Figure 3.10: Stack histogram computation algorithm

taking the product of the ranges of the loops enclosing the statement. where the loop index ranges are given by the partition under consideration.

Once the stack histogram is computed symbolically. there are several approaches that can be taken to evaluate the expressions and estimate the number of cache misses. These approaches are described in Section 5.1.

3.3.7 Example - Matrix Multiplication

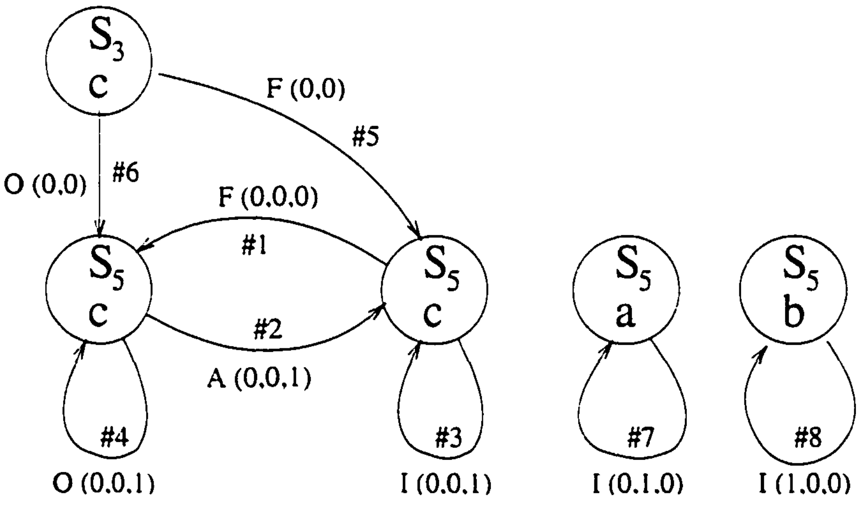
In this section we work through an example of using dependence spans and array sections (their projections) to compute the stack distances exactly. Figure 3.11 presents the Fortran code for our example. as well as the data dependences labeled with the types and dependence distances for each loop. and numbered in the order in which they are be presented in Table 3.2.

Table 3.2 shows for each dependence the dependence span. the array sections spanned by the dependence. and finally the stack distance that is computed for the dependence.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
S1:   do i = 1, n
S2:      do j = 1, n
S3:         C(i,j) = 0
S4:         do k = 1, n
S5:            C(i,j) = C(i,j) + A(i,k) * B(k,j)
S6:         end do
S7:      end do
S8:   end do
```

(a) Fortran code



(b) Data dependences

Figure 3.11: Matrix multiplication example.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Dep	Dependence Span		$AR(A,\delta)$	$AR(B,\delta)$	$AR(C,\delta)$	Stack Distance
#1		(i,j,k)	(i,k) 1	(j,k) 1	(i,j) 1	3
#2		$(i,j,k:k+1)$		1	(i,j)	1
#3	-- " --	$(i,j,k:k+1)$	(i,k) 1	(j,k) 1	(i,j) 1	3
#4		$(i,j,k:k+1)$	"	"	"	3
#5		$(i,j,1:k)$	$(i,1:k-1)$ $k-1$	$(j,1:k-1)$ $k-1$	(i,j) 1	$2k-1$
#6		$(i,j,1:k)$	$(i,1:k)$ k	$(j,1:k)$ k	(i,j) 1	$2k+1$
#7		$(i,j,k:n)\cup(i,j+1,1:k)$	$(i,k:n)\cup(i,1:k)$ n	$(j,k:n)\cup(j+1,1:k)$ n	$(i,j-1:j)$ 2	$2n+2$
#8		$(i,j,k:n)\cup(i,j+1:n,1:n)\cup(i+1,1:j-1,1:n)\cup(i+1,j,1:k)$	$(i,k:n)\cup(i,1:n)\cup(i+1,1:n)\cup(i+1,1:k)$ $\begin{cases} 2n & 1<j<n \\ n & j=1 \\ n-k & j=n \end{cases}$	$(j,k:n)\cup(j+1:n,1:n)\cup(1:j-1,1:n)\cup(j,1:k)$ n^2	$(i,j)\cup(i,j+1:n)\cup(i+1,1:j-1)\cup(i+1,j)$ $n+1$	$\begin{cases} n^2+3n+1 & [1<j<n] \\ n^2+2n+1 & [j=1] \\ n^2+2n+1-k & [j=n], k=\overline{1,n} \end{cases}$

Table 3.2: Stack distances computation for matrix multiplication

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To show how the algorithm works, consider dependence #7, the input dependence on reference $A(i,k)$ carried by loop j. In the first two columns in Table 3.2, the dependence span is shown both graphically and geometrically. Because the dependence is carried by loop j with distance 1, the dependence span consists of the remainder of the iterations in loop k in iteration j and the iterations up to iteration k in iteration $j + 1$. The array elements accessed in these iterations are shown in the next three columns. For A, these elements are an entire row of the matrix, elements from k to n on row i for iteration j and elements from 1 to k on row i for iteration $j + 1$. The total number of distinct elements accessed in array A is n. Considering the array B, again there are a total of n distinct array elements accessed, distributed on two columns of the matrix, j and $j + 1$. And finally, there are only two elements accessed in array C, $C(i,j)$ and $C(i,j + 1)$. Thus, the number of distinct array elements spanned by this dependence is $2n + 2$. The number of distinct array elements spanned by the other dependences is computed similarly.

When the stack histogram is computed, since this dependence is the only incoming dependence on array reference $A(i,k)$, there will n^2 references at distance ∞, which occur in the first iteration of loop j in each iteration of loop i. The other $n^3 - n^2$ references to $A(i,k)$ will happen at distance $2n + 2$.

3.3.8 Spatial Locality

In the previous discussion we considered the cache lines to be of only one array element. In order to compute the stack histogram for real cache line sizes, we need to determine the number of distinct cache lines that are spanned by a dependence. Since we already computed the number of distinct array elements spanned by a dependence, we just have to translate that number into cache lines. In other words, we need to determine the cache lines layout for the array sections.

Figure 3.12 shows an example. Assuming a two dimensional array A, with $MMAX \times NMAX$ elements, we show the potential mapping of cache lines in column major order (such as Fortran). Also, assume that some dependence spans the $M \times N$ array section shown as a shaded region in the figure.

We compute $LDA = \left\lceil \frac{MMAX}{LS} \right\rceil$, the number of cache lines that cover one column of the matrix, where LS is the size of the cache line expressed in number of array elements. The number

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

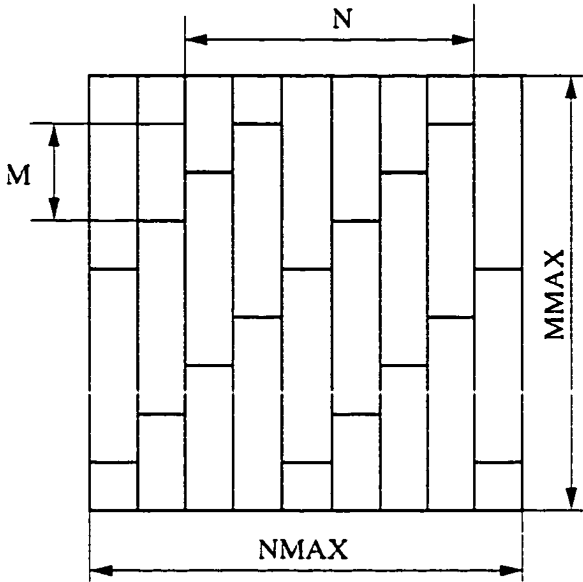


Figure 3.12: Cache lines mapping on an array section

of cache lines covering an array section with dimensions M and N is given by the equation:

$$DL = \sum_{i=1}^{N} \left\lceil \frac{(i \times LDA)\%LS + M + offset}{LS} \right\rceil \tag{3.6}$$

where $offset = 0$ if the first element of the array maps at the beginning of a cache line.

The stack histogram is computed using the same algorithm presented in Section 3.3.6, except that instead of computing the number of distinct array elements accessed, we compute the number of distinct cache lines accessed. That is, the array section area AS is returned in terms of cache lines. Of course, the expressions denoting both stack distances and array references will contain a symbolic variable for the cache line size. This symbolic variable is treated like all the other hardware parameters that are used in the performance expressions.

3.3.9 Associativity

It has been previously shown [63, 35] that set-associative miss ratios can be closely estimated from the fully-associative miss-ratio. The compile-time stack distances algorithm estimates the number of misses for fully-associative LRU caches. Therefore, in order to estimate the number of misses for a real machine, the number of misses for a set associative cache are deduced, using a probabilistic argument, from the number of misses for fully associative cache. For details, see [35] Section V.B.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.4 Indirect Accesses Model

When not enough compile-time information is available to compute the data dependence distance vectors, other methods are required to estimate the number of cache misses since the stack distance algorithm cannot be applied. The most common case in which the compiler fails to compute data dependences is when indirect array accesses are present (the subscript of the array is a reference to another array). Therefore, we call the model that is used in the presence of indirect array accesses, the *indirect accesses model*. This model can be applied to any array references in a loop nest, however, it usually overestimates the number of array elements accessed.

The main idea behind this model is to estimate the number of array elements accessed by computing the number of iterations that access the array, i.e., the total number of references to the array, similar to the work of Porterfield [52] and Ferrante et al. [27]. We limit this number by the size of the array, since it is obvious that there can not be more distinct array elements accessed than there are elements in the array. One can contrive examples in which this estimation will approximate very badly the actual behavior, but in most cases encountered in practice, the method approximates quite well the measured data. We know of no other method that estimates the number of cache misses at compile-time in the presence of indirect accesses.

Using this method, the number of cache misses for level i in the memory hierarchy, M_i in Equation 3.3, is computed as follows:

$$M_i = \sum_{distinct\ A} \frac{\min(refs_A, size_A) \times elemSize_A}{BlockSize_i}. \quad (3.7)$$

where, $refs_A$ represents the number of references to array A, $size_A$ represents the number of elements in array A (since we use Fortran 77, the size of the arrays is known at compile-time), and $elemSize_A$ is the size, in bytes, of one element of the array A. The element size dependes on the declared data type of array A. $BlockSize_i$ is the size of the cache line for level i of the memory hierarchy.

For example, consider the code in Figure 3.13. It implements a sparse matrix-vector multiplication operation, and it is taken from the Splib package [9], a library of sparse functions developed at Indiana University. The loop multiplies the matrix A, stored in compressed sparse row storage,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
L1      do i = 1. m
S1          Y(i) = 0.0d0
L2          do k = ia(i). ia(i+1)-1
S2              Y(i) = Y(i) + A(k) * X(ja(k))
            end do
        end do
```

Figure 3.13: Sparse matrix vector multiplication

with the dense vector X. and stores the result in the vector Y. The vectors ia and ja store the row and column indices in the matrix A. respectively. Since ia and ja depend on the input data set (matrix). many of the accesses to array X in statement S2 can map to the same element. depending on the value of the array element $ja(k)$. The best we can do at compile time is to approximate the number of references to X by the minimum between the number of iterations of the loop and the size of the vector X. which is known to be equal to the column size of A.

Obviously. in this case. not even the number of iterations is known at compile time. However. by using profiling information we can estimate it. In fact. the follwing code shows how Polaris generates instrumentation to collect the profiling information needed.

```
REAL*8  a(*), x(n), y(m)
INTEGER ia(*), ja(*)

_delphi_cm = 0
_delphi_count_x = 0
DO i = 1, m, 1
   y(i) = 0.0D0
   DO k = ia(i), ia(i+1)-1, 1
      y(i) = y(i)+a(k)*x(ja(k))
   ENDDO
   _delphi_count_x = _delphi_count_x+(ia(1+i)+(-ia(i)))
   _delphi_cm = _delphi_cm+(8+(-12)*ia(i)+12*ia(1+i))
ENDDO
_delphi_cm = _delphi_cm+MIN(_delphi_count_x, n)*8
_delphi_cm = _delphi_cm+8*m
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.5 Summary

In this chapter we have presented compile-time models to estimate the performance of scientific codes. An overall performance prediction model for a computing system is decomposed into parts that model the CPU. the memory hierarchy. the I/O system and inter-processor communication. The common representation of the performance data as symbolic expressions. with variables for program constructs. input data set. and architecture. allows for machine independent performance estimation at different program granularities.

A model for processor execution time estimation was presented. It counts operations in the high level language code and applies compile-time heuristics to model low-level compiler optimizations. The processor architecture is abstracted by providing variables for groups of basic operations.

The bulk of the chapter discusses the modeling of the memory hierarchy. A precise model of cache behavior based on stack distances is developed. and a compile-time algorithm to compute the stack distances is given. The stack distances compile-time algorithm depends on the ability of the compiler's data dependence test to extract distance vectors information. In Polaris we use the Omega test [53] for this purpose. The Omega test is able to extract the data dependence distance information for more than 75% of the loops in SPECfp95.

For the cases where dependence information is not available. such as sparse computations with indirect array accesses. a simpler model is presented. This model estimates the number of cache lines accessed in the loop using very simple heuristics. Experimental results with both models are presented in Chapter 6.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4

Stack Distance and Stack Algorithms

In this chapter we present our experience with the stack processing algorithm to quantify program locality. We start by presenting a new metric for data locality, the stack histogram. We then discuss several ways to improve the performance of the LRU stack processing algorithms, when used to process memory traces.

4.1 The Stack Distance as a Metric for Locality

> "There are three most important factors in writing programs, either sequential or parallel: locality, locality, locality." [Michael Wolfe, personal communication]

Programs with good data locality take better advantage of the caches, have low communication costs and low interconnection network traffic. There are many companies that will hire highly skilled programmers just to have them tune their most important codes to run well on a specific architecture. But, in today's rapidly changing landscape, machines become obsolete very soon, and the programmers keep changing applications to suit the new evolving architectures.

There is one most important characteristic we are looking for in a model for data locality - *architecture independence*. We would like to specify what is the locality of a program on existing machines as well as on future architectures. We consider that a good theoretical model should be *abstract*, to hide the details that would make it too complex), and *general*, to be applicable to a large variety of programs and systems.

The model we propose is based on the "stack processing" method developed by Mattson et al. [43] to evaluate the cost-performance of page replacement algorithms in virtual memory sys-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Variable	Program 1	Program 2
a	2	2
b	2	2
c	9	15
Average	4.33	6.33

Table 4.1: Inter-reference distances and averages for memory references in Programs 1 and 2

tems. Their technique, for a particular page replacement algorithm (such as Least Recently Used), computes a *success function*, based on the frequencies of accesses at different stack distances, in a single pass through the memory trace. The stack distances are computed by maintaining a list of pages in an LRU stack, and measuring a distance on this stack for every page reference.

Our model for data locality also maintains an LRU stack, but the resolution is either at memory location level for temporal locality, or at cache line level for spatial locality. The model can be easily extended to handle multiprocessor codes by maintaining a separate stack for each processor.

Lilja et al. [42] propose the inter-reference distance as a model for locality. They define the inter-reference distance as the number of memory references that occur between two references to the same memory location. They claim that the inter-reference distance can be used as a measure for the temporal locality of the variable, and that the average of all the inter-reference distances for all the variables in the program can be used as a measure of temporal locality that exist within a program.

Consider the following example of memory traces generated by two different programs:

```
Program 1: c a b a b a b a b c
Program 2: c a b a b a b a b a b a b a b c
```

Inter-reference distances and their averages are shown in Table 4.1. We note that the inter-reference distance averages differ for the two programs, leading us to believe that the first program has better temporal locality than the first one.

This is actually not true, since both programs have the same working set, and in fact there is more reuse in the second program, as proven by the stack distances algorithm (see Table 4.2), which gives the same distances for both programs, but more references to distance 2.

Although one can reason about locality using the stack histogram, if a single number that defines the locality of the program is desired, we can compute the average locality based on the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Program 1		Program 2	
Stack Distances	# References	Stack Distances	# References
∞	3	∞	3
2	6	2	12
3	1	3	1

Table 4.2: Stack distances and number of references in Programs 1 and 2

histogram as follows:

$$AvgLoc = \frac{\sum_{\delta} (\delta \times s(\delta))}{\sum_{\delta} s(\delta)}.$$

where $s(\delta)$ is the number of accesses at stack distance δ, and δ are the stack distances for which $s(\delta) > 0$. The lower the value of $AvgLoc$, the better the locality of the program.

As an example, consider matrix multiplication. It is well known that tiling improves locality in matrix multiplication [75, 17]. In Figure 4.2 we present the stack histograms for a 100x100 matrix multiplication, and two versions of tiled matrix multiplication loops: the one in Figure 4.2(b) has the two outermost loops tiled with tile size 25, and the one in Figure 4.2(c) has all three loops tiled, tile size being also 25. The codes for these loops are shown in Figures 4.1(a-c).

In Table 4.3, we show how the two metrics for locality, the inter-reference distance and the stack distance compare for matrix multiplication and its tiled versions. For each metric, we consider the three versions of the matrix multiplication loop, the classical `ijk` loop, the outermost (`ij`) loops tiled (2-tiled) and all three loops tiled (3-tiled). We consider both temporal and spatial locality. For spatial locality, two values are given, for cache line sizes (CLS) of 32 bytes and 128 bytes. These correspond to 4 and 16 array elements per cache line and are among the most commonly used values for the cache lines in L1 and L2 caches, respectively.

We note that, for temporal locality, there is no difference between the 2-tiled loops and the classical loop when using the inter-reference distance metric. Another anomaly of the inter-reference distance metric can be observed for spatial locality between the 2-tiled and 3-tiled loops. The locality metric increases when the locality improves. These anomalies are a consequence of the fact that the inter-reference distance metric considers **all** the references between two accesses to the same memory location, not just references to **new** memory locations. Moreover, when averaged

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
do i = 1, n
   do j = 1, n
      do k = 1, n
         c(i, j) = c(i, j) + a(i, k) * b(k, j)
      enddo
   enddo
enddo
```

(a) Matrix multiplication

```
do ii = 1, n, TILE
   do jj = 1, n, TILE
      do i = ii, MIN(ii+TILE-1, n)
         do j = jj, MIN(jj+TILE-1, n)
            do k = 1, n
               c(i, j) = c(i, j) + a(i, k) * b(k, j)
            enddo
         enddo
      enddo
   enddo
enddo
```

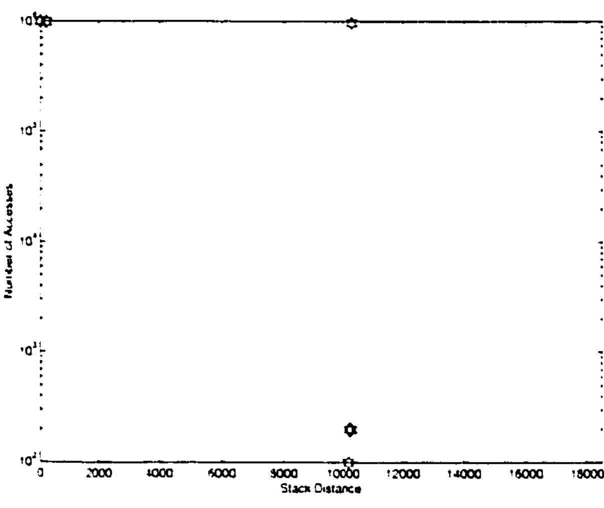
(b) 2-tiled matrix multiplication

```
do ii = 1, n, TILE
   do kk = 1, n, TILE
      do jj = 1, n, TILE
         do j = jj, MIN(jj+TILE-1, n)
            do k = kk, min(kk+TILE-1, n)
               do i = ii, MIN(ii+TILE-1, n), 1
                  c(i, j) = c(i, j) + a(i, k) * b(k, j)
               enddo
            enddo
         enddo
      enddo
   enddo
enddo
```

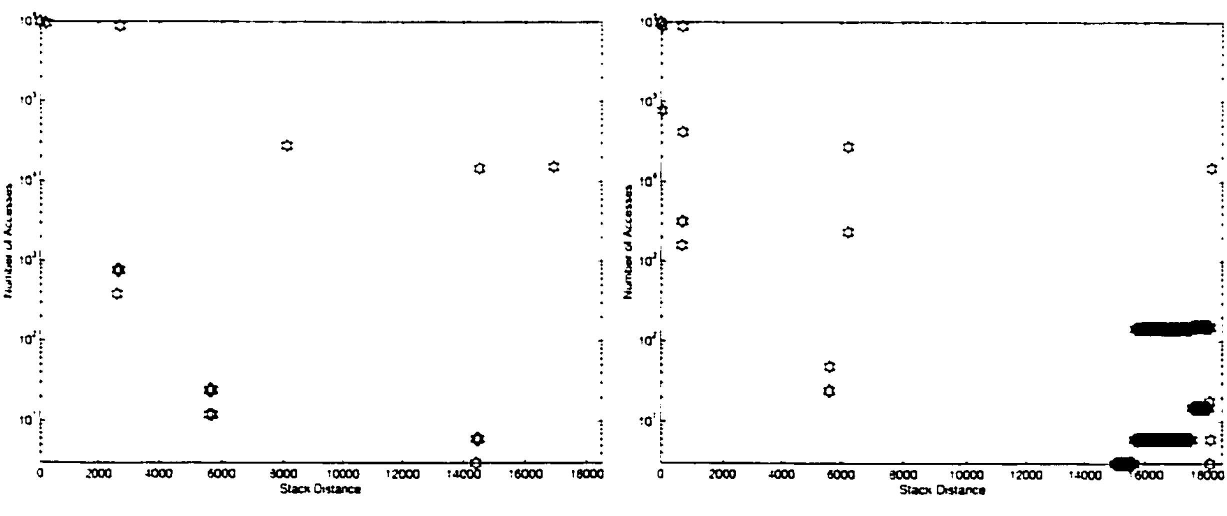
(c) 3-tiled matrix multiplication

Figure 4.1: Fortran code for tiled matrix multiplication

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a) Matrix multiplication 100x100



(b) Matrix multiplication 100x100 - 2 tiled 25x25

(c) Matrix multiplication 100x100 - 3 tiled 25x25

Figure 4.2: Stack histograms for matrix multiplication and tiled matrix multiplication

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Metric	Locality		Classical MM	2-tiled	3-tiled
Inter-reference Distance	Temporal		101.005	101.005	91.462
	Spatial	(CLS = 32)	2650.007	2650.007	2668.749
	Spatial	(CLS = 128)	886.009	859.009	868.798
Stack Distances	Temporal		2619.890	886.451	367.794
	Spatial	(CLS = 32)	221.146	93.087	30.219
	Spatial	(CLS = 128)	46.971	36.014	6.026

Table 4.3: Temporal locality for matrix multiplication

over all variables. the variables with *bad* locality. even if accessed only a few times. contribute the same weight as *good* locality variables.

On the other hand. the stack distances metric does not suffer from these anomalies. The stack processing algorithm makes sure that only accesses to new memory locations or in a position below a memory location in the stack modify the stack distance of that location. Thus. the stack distance metric behaves as expected - when program locality increases. the value of the average locality decreases.

4.2 LRU Stack Processing Algorithms

This section describes our experience using the stack processing algorithm [43] for estimating the number of cache misses in scientific programs. By using a new data structure. and various optimization techniques. we obtain instrumented run-times within 50 to 100 times the original optimized run-times of our benchmarks.

The stack algorithm [43] was originally designed for modeling virtual paging. i.e. to operate on a program trace consisting of virtual page references. but in the recent past has been used mainly to model cache behavior. by tracing cache line references [65. 67. 35. 70].

The main advantage of the stack algorithm in simulating cache behavior is that it allows the estimation of the number of misses for caches of any size in a single pass through the trace. Variants of the algorithm have been used to simulate caches of multiple line sizes.

The stack algorithm is however very expensive to run. especially if the stack becomes large enough. It was soon recognized that more efficient data structures were needed to do the job of the stack search. Bennett and Kruskal [6] presented an algorithm which replaces the stack with

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a preallocated hierarchy of partial sums. Hill and Smith [35] used a forest of trees to simulate multiple cache associativities: Sugumar and Abraham [65] used a generalized binomial tree for the same purpose.

Seeking to further improve the performance of the stack algorithm. we introduce two new data structures and corresponding algorithms. each of which is more suitable for a particular kind of application. The *interval tree* approach works well for programs with long traces but relatively good locality. whereas the *preallocated tree* approach is more suited to shorter traces with bad locality.

Figure 4.3 gives a formal three-step description of the LRU stack algorithm. as first described by Mattson in [43]. We use this description as the basis for the algorithms we present.

It is assumed henceforth that the algorithm is operating on a memory trace of length N that contains M distinct memory references (obviously $M \leq N$). For the notations used in this chapter refer to Section 4.2.9.

Repeat the following steps for each memory reference x_τ. $0 \leq \tau < N$:

- **search**: find the location in the stack of the most recent reference to the current location.
- **count**: compute $dist(\tau)$. the stack distance for the current location. by finding the previous reference to the current location and counting the number of elements on the stack above it. If the most recent reference is not found. $dist(\tau)$ is defined as ∞.
- **update**: bring the most recent reference to the top of the stack.

Figure 4.3: Stack algorithm

4.2.1 Naive Implementation

This implementation directly follows the algorithm presented above. The stack is represented as a doubly linked list. For each reference in the trace. the first two operations (**search** and **count**) are executed simultaneously by traversing the stack top to bottom. If the element exists in the stack. its distance from the top of the stack is recorded. Finally the element is moved from its current position to the top of the stack – the **update** stage. If the element is not found. ∞ is recorded as its stack distance and the element is pushed on top of the stack.

Analysis For each reference in the trace the work done is. in the worst case. M (due to the traversal of the linked list). The total complexity is thus $O(NM)$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The worst case doesn't happen very often. In fact, many programs exhibit excellent locality, causing many references to lie close to the top of the stack. Unfortunately the few references that are near the bottom of the stack cause huge slow-downs, resulting in overall bad performance.

4.2.2 Markers Algorithm

The major cause of slowness in the naive algorithm is the linear traversal of the linked list that makes up the stack. The *markers* algorithm attempts to replace linear search wherever possible.

The **search** phase of the markers algorithm is done using a hash table that associates a cache line/memory/page reference with its current place in the linked list. Given enough hash buckets, hashtable access and update are $O(1)$ operations. The number of necessary hash buckets can be approximated with M, the number of distinct references in the trace.

Unfortunately finding an element in the middle of the stack by using the hashtable is not enough. The stack depth of the element needs to be counted. To avoid traversing the stack from top to bottom, a set of *markers* are interspersed in the linear list implementing the stack, one about every D elements. The markers form another doubly linked list, and each marker records its distance from the top. To find out the depth of a memory reference in the stack, one needs to find the nearest marker by traversing the stack (a marker would be at most D steps away) and then look up the marker's distance from the top.

When an element is removed from the stack and inserted at the top, the markers between the top and the element need to be updated. This involves at most M/D steps.

Analysis The cost per memory reference of this algorithm is at most $D + M/D$ (the cost of finding a marker, plus the cost of updating all markers up to the beginning of the stack). D can be varied at runtime by adding or removing markers, in order to minimize the cost: assuming $D = \lceil\sqrt{M}\rceil$, the cost evaluates to $O(\sqrt{M})$ per element, or $O(N * \sqrt{M})$ total.

4.2.3 Alternative Data Structures

The major stumbling block in implementing more efficient versions of the LRU stack algorithm is the implementation of the stack as a linear list. We will present a formulation of the LRU stack algorithm that does not use a stack. We will closely follow Bennett and Kruskal's [6] notation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Definition 1. We formalize the concept of the hashtable **P**, which we already used informally in the markers algorithm. Let us define $\mathcal{J}$ as the set of indices of references to z that occurred *before* an index τ in the trace:

$$\mathcal{J}_\tau = \{i \mid 0 \le i < \tau \wedge x_i = z\}$$

Using $\mathcal{J}$ we define the hashtable $\mathbf{P}_\tau$ as follows:

$$\mathbf{P}_\tau(z) = \begin{cases} max\{i \mid i \in \mathcal{J}\} & \text{if } \mathcal{J} \ne \emptyset \\ undefined & \text{otherwise} \end{cases} \tag{4.1}$$

$\mathbf{P}_\tau(z)$ is undefined when a cold miss occurs, i.e. when there is *no* previous reference to z.

Definition 2. Next we define **B**, a mapping from the trace indices $0 \dots N-1$ to $\{0, 1\}$. Like **P**, **B** changes with time and therefore is subscripted with τ. $\mathbf{B}_\tau(i)$ is defined as follows:

$$\mathbf{B}_\tau(i) = \begin{cases} 1 & \text{if } \mathbf{P}_\tau(x_i) = i \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

$\mathbf{B}_\tau(i)$ is 1 if at time τ there is no reference to x_i in the program trace at any index larger than i.

Definition 3. Given **P** and **B** we can define $dist(\tau)$, the stack distance of the element x_τ in the program trace: it is the number of 1's in **B** between the last reference to x_τ and τ.

$$dist(\tau) = \begin{cases} |\mathcal{H}| & \text{if } \mathbf{P}_\tau(x_\tau) \text{ is defined} \\ \infty & \text{otherwise} \end{cases} \tag{4.3}$$

where $\mathcal{H}$ is the set of trace indices after $\mathbf{P}_\tau(x_i)$ whose **B** values are 1:

$$\mathcal{H} = \{i \mid \mathbf{P}_\tau(x_\tau) \le i < \tau \wedge \mathbf{B}_\tau(i) = 1\}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Repeat for each reference x_τ, $0 \leq \tau < N$:

- **search**: compute $\mathbf{P}_\tau(x_\tau)$;
- **count**: evaluate $dist(\tau)$. If $\mathbf{P}_\tau(x)$ is undefined then $dist(\tau)$ is defined as ∞;
- **update**: change **B** and **P** as follows:

$$\mathbf{B}_{\tau+1}(i) = \begin{cases} 1 & \text{if } i = \tau \\ 0 & \text{if } i = \mathbf{P}_\tau(x_\tau) \\ \mathbf{B}_\tau(i) & \text{otherwise} \end{cases}$$

$$\mathbf{P}_{\tau+1}(z) = \begin{cases} \tau & \text{if } z = x_\tau \\ \mathbf{P}_\tau(z) & \text{otherwise} \end{cases}$$

Figure 4.4: Modified stack algorithm

We can now reformulate the stack algorithm by using **P** and **B** instead of the stack.

4.2.4 Bennett and Kruskal's Algorithm

We present Bennett and Kruskal's algorithm [6] first because it introduces ideas we need later.

The algorithm represents **P** and **B** explicitly. The first step of the algorithm, evaluating $\mathbf{P}(x_\tau)$, is a hash table lookup.

The **count** step of the algorithm counts the number of true values in **B** between the indices $\mathbf{P}(x_\tau)$ and τ. To make the counting step efficient, Bennett and Kruskal use a hierarchy of partial sums $\mathbf{B}^1, \mathbf{B}^2 \ldots \mathbf{B}^L$, where $L = \lceil log(N) \rceil$. Renaming **B** to $\mathbf{B}^0$, the partial sum hierarchy is set up such that for some chosen interval m, at any time τ,

$$\mathbf{B}^s{}_\tau(j) = \sum_{i=j\cdot m}^{i=(j+1)\cdot m-1} \mathbf{B}_\tau^{s-1}(i)$$

This formula describes an m-ary tree of nodes having the value of each node being equal to the sum of the values of its children.

Calculating the number of 1's between the indices $\mathbf{P}(x_\tau)$ and τ is now a matter of traversing the partial sum hierarchy, as shown in Figure 4.5. The figure presents the first 31 elements of a trace. We trace the 31st access, and x_31 last occurred in position 4.

The third step, updating **B**, also becomes a matter of tree traversal, since all partial sums on

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

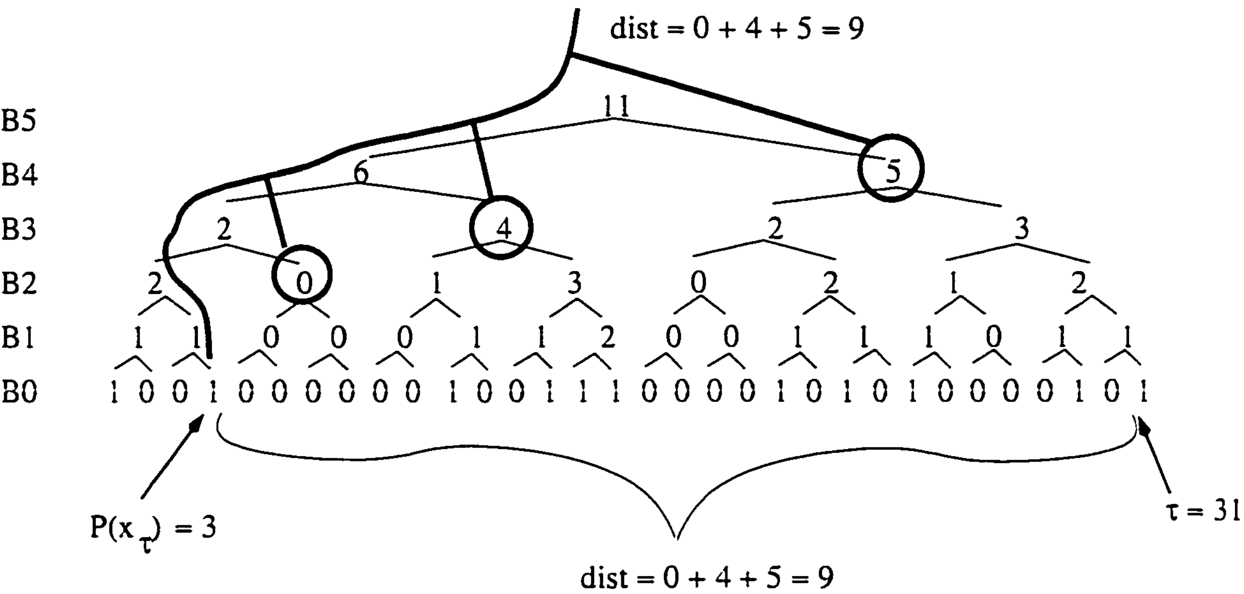


Figure 4.5: A partial sum hierarchy

the path from the root of the hierarchy to the leaf node are affected.

The algorithm needs two traversals of the tree. The first traversal, from the root to index $\mathbf{P}(x_\tau)$, deletes $\mathbf{P}(x_\tau)$ as the last reference to x by setting $B(\mathbf{P}(x_\tau))$ value to 0 and adjusting all partial sums along the path. The second traversal is from the root to index τ and sets $B(\tau)$ to 1, again adjusting partial sums on the path. For reasons of brevity we are not going to fully explain the algorithm, except to mention that our major improvement, to be presented in the next sections, replaces the two traversals with a single traversal of the tree.

Analysis Since the tree traversal is an $O(log(N))$ operation and the location finder works in constant time (hashtable lookup is $O(1)$), the total execution time is $O(Nlog(N))$. The memory requirements of the algorithms are very large because **B** and its partial sums are represented explicitly in memory.

4.2.5 Hole-based Algorithms

We define a *hole* as a memory reference in the program trace that is *not* the latest reference to a particular location at time τ. Holes are thus elements in $\mathbf{B}_\tau$ that have been set to 0.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Whereas values of 1 in **B** (and corresponding latest references) are newly created and then destroyed all the time. holes have the property of being created and never destroyed.

Using the concept of holes. the stack distance at index τ can be expressed as

$$dist(\tau) = \tau - \mathbf{P}_\tau(x_\tau) - holes_\tau(\mathbf{P}_\tau(x_\tau))$$

where $holes_\tau(i)$ is the number of holes in the program trace between indices i and τ. Here we are in effect counting the 0's in **B** instead of counting the 1's. and adjusting Equation (4.3) to reflect this.

Holes can be represented more efficiently than latest references. We will present two kinds of algorithms based on holes. a variant in which holes are held by an interval tree and another which is a faster version of Bennett and Kruskal's algorithm.

4.2.6 Interval Tree of Holes

An interval tree is used to efficiently represent an ordered set of mutually disjunct intervals $I = \{[i_{11}, i_{12}], [i_{21}, i_{22}], \ldots, [i_{n1}, i_{n2}]\}$. In our case the intervals in I are all bounded by natural numbers (indices in the program trace). The intervals represent contiguous sets of indices that are holes in the trace.

Interval trees (Figure 4.6) are represented as a quasi-balanced binary trees **BT** (such as red-black trees [18] or AVL trees [39]) in which each node **n** represents the closed interval $[k_1(\mathbf{n}), k_2(\mathbf{n})]$. The tree ordering corresponds to the order of the intervals in I: thus $k_1(\mathbf{n}) > k_2(left(\mathbf{n}))$ and $k_2(\mathbf{n}) < k_1(right(\mathbf{n}))$. where $left(\mathbf{n})$ and $right(\mathbf{n})$ are respectively the left and right children of **n**.

The Partial Sum Hierarchy

We use the interval tree to evaluate the number of holes between $P(x_\tau)$ and τ. There are no holes beyond the current index τ (a logical impossibility considering the definition of a hole). Thus we are left with counting the number of holes at indices larger than $P(x_\tau)$. To do this. we follow Bennett and Kruskal's method and associate a value $sum(\mathbf{n})$ with each interval node **n**. to hold the sum of holes contained in the children of **n**. Our hole tree now becomes equivalent to Bennett and Kruskal's partial sum hierarchy.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

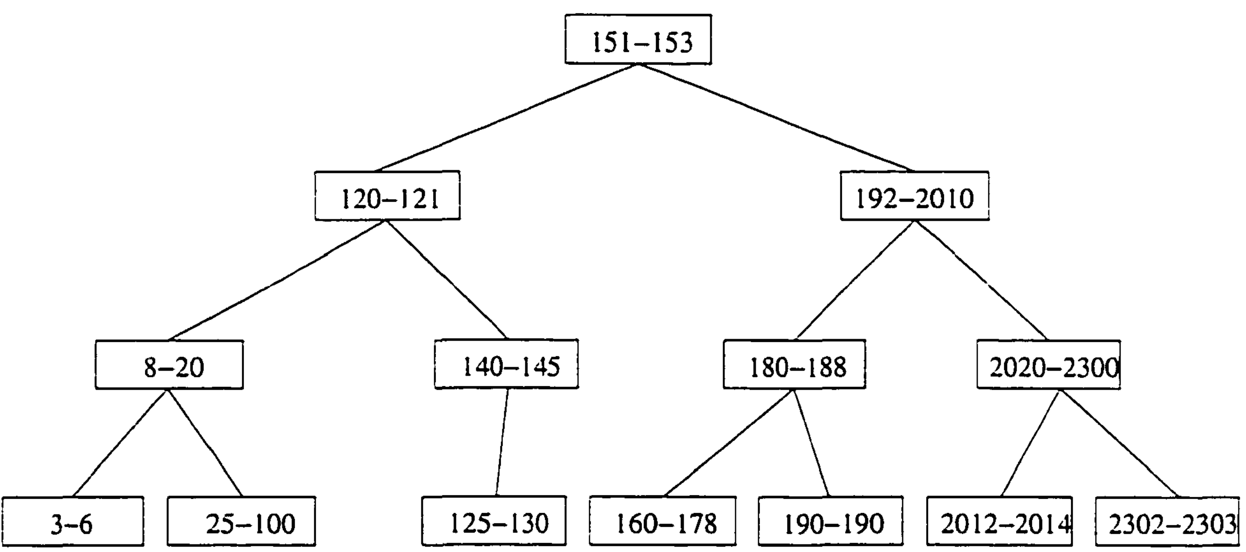


Figure 4.6: An interval tree

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

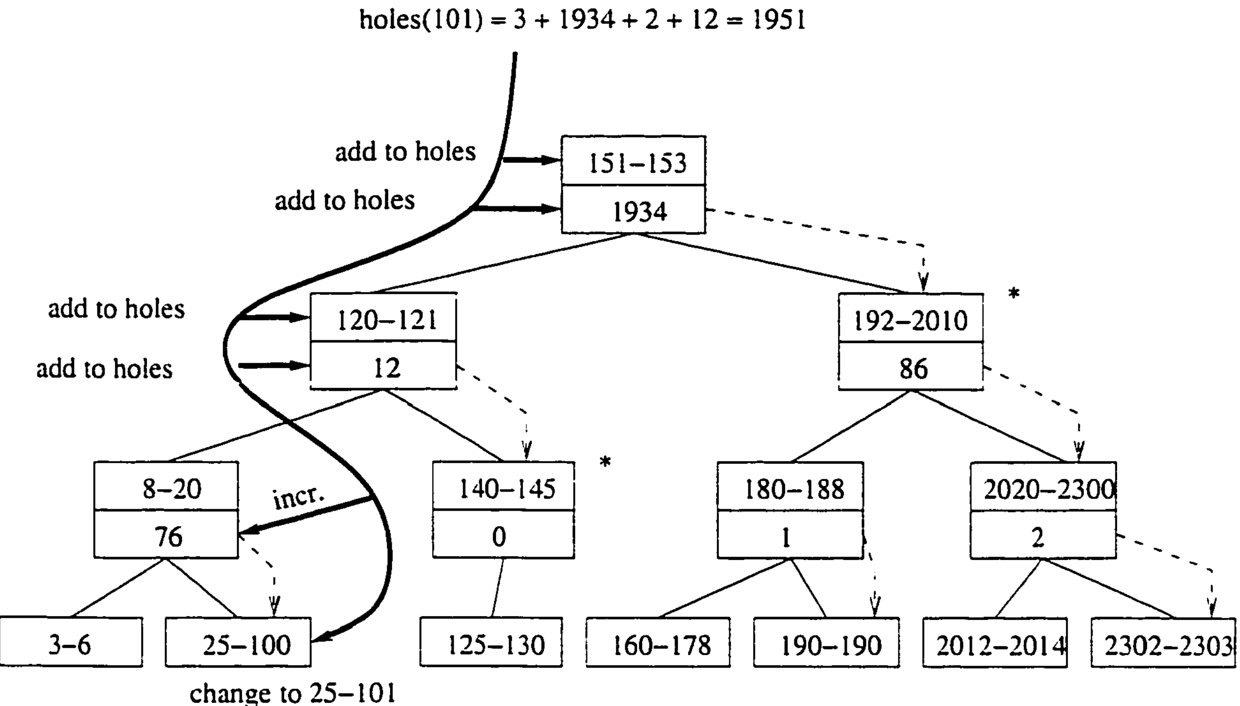


Figure 4.7: Updating the tree of holes

A slight optimization is to make $sum(\mathbf{n})$ hold the sum of holes in the *right subtree* of $\mathbf{n}$ instead of $\mathbf{n}$ itself. In Figure 4.7 the shaded boxes contain partial sums of the right subtree of their nodes, as indicated by the dashed arrows. This optimization reduces the number of right subtree dereferentiations when the next target of the tree traversal is the left subtree (in Figure 4.7 the nodes marked with (*) will not need to be dereferenced).

The counting algorithm works like this: we traverse the interval tree from the root towards the leaf node closest to index $i = P(x_\tau)$. We carry a partial sum along the path, and add to it the sum of holes in all subtrees encountered *to the right* of the path (i.e. having indices larger than i).

Updating the Interval Tree

We now extend the counting algorithm to include the third component of the LRU stack algorithm: **update**. We need to update the tree structure as well as the partial sum hierarchy residing in it.

With regard to inserting a new hole p into the interval tree, there are several cases to consider:

- p may be adjacent to a single existing interval $[k_1, k_2]$ in the tree, i.e. $p = k_1 - 1$ or $p = k_2 + 1$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In this case the interval is adjusted to include p.

- p may be adjacent to two intervals $[k_1.p-1]$ and $[p+1.k_3]$. In this case the intervals are fused into a single interval $[k_1.k_3]$ prompting the deletion of one of the nodes and the potential re-balancing of the whole tree.
- p may not be adjacent to any intervals in **BT**. In this case a new node is created. to hold the interval $[p.p]$. Again. the tree may need to be rebalanced.

The partial sum hierarchy is updated by changing the *sum* values of the nodes on the path from the root to the affected interval. Figure 4.7 illustrates the operation of counting holes and inserting a new hole at location 101 in an example tree. Figure 4.8 lists the algorithm that performs this operation.

Analysis The algorithm presented in Figure 4.7 is based on a quasi-balanced binary tree. $dist(p.\mathbf{n})$ is a variant of the insertion operation for quasi-balanced binary trees. which makes it an $O(log(M))$ operation (the number of disjunct hole intervals. and thus nodes in the tree. is always less than M+1). Thus the total execution time of the Binary Tree Hole Algorithm is $O(Nlog(M))$.

4.2.7 Preallocated Tree of Holes

A tree of holes can also be implemented as a preallocated fixed tree $\{\mathbf{B}^0.\mathbf{B}^1....\mathbf{B}^{log_2(N)}\}$. like the one of Bennett and Kruskal. Unfortunately the memory requirements for the whole tree get quickly out of hand: for a program trace of length 2^{31} (a realistic number for todays programs) we need to allocate $1+2+2^2+...+2^{30}=2^{31}-1$ locations.

The silver lining is that not all locations need to be of the same size. Elements of $\mathbf{B}_0$. for instance. need to hold only one bit: elements of $\mathbf{B}_1$ need to be two bits each. and so on: the total memory requirement is $\frac{1}{8}\times(2^{30}+2\times2^{29}+3\times2^{28}+\ldots+31\times2^0)\cong 536.87$MB. which fits into the virtual memory of most modern workstations. Also. the algorithm does not use all of this memory at once. but rather progresses slowly through it as the trace is analyzed. This allows for huge portions of the preallocated tree to reside in virtual storage.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
function dist (n. p)
begin
      if (p ¡ key1(n) - 1)                  /* continue search left */
         if (left(n) ≠ nil)
            return sum(n)+dist(left(n))
         else                               /* can't continue left - no nodes left */
            left(n) := new interval (p.p)
            sum(left(n)) := 0
            return sum(n)
         end if
      else if (p > key2(n) + 1)             /* continue search right */
         if (right(n) ≠ nil)
            sum(n) := sum(n) + 1
            return dist(right(n))
         else                               /* can't continue right - no nodes left */
            right(n) := new interval (p.p)
            sum(right(n)) := 0
            return 0
         end if
      else if (p = key1(n) - 1 AND p = key2(left(n)) + 1) /* merge left node */
         key1(n) := key1(left(n))
         remove_node(left(n))
         rebalance(n)
         return key2(n) - p + sum(n)
      else if (p = key1(n) - 1)             /* add to node */
         key1(n) := p
         return key2(n) - p + sum(n)
      else if (p = key2(n) + 1 AND p = key1(right(n)) - 1) /* merge right node */
         key2(n) := key2(right(n))
         remove_node(left(n))
         rebalance(n)
         return sum(n)
      else if (p = key2(n) + 1)             /* add to node */
         key2(n) := p
         return sum(n)
      end if
end
```

Figure 4.8: Interval tree update

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Analysis Since the tree is preallocated. and has N leaf nodes. tree traversal is now an $O(log(N))$ operation rather than $O(log(M))$. which would seem to make this algorithm impractical. Also. the tree needs to be allocated before the program is run. which means that the user has to guess N.

However. once N is calibrated. the algorithm becomes the fastest we tried so far. outperforming Bennett and Kruskal's by a factor of up to 2:1. The reason for this is that only one tree traversal is needed per element. as opposed to two for Bennett and Kruskal's algorithm.

4.2.8 Experimental Evaluation

We selected the Perfect Benchmarks [7] as our experimental base and instrumented them with a source-to-source translator to generate a program trace. Rather than storing the program trace we hooked up the analyzer to the instrumented benchmark directly. and generated the trace and the histogram on the fly.

At first we used the naive implementation of the LRU stack algorithm. and experienced a drastic slowdown. In an effort to find better implementations of the LRU algorithm we experimented with all algorithms described in this chapter.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Parameters			Overhead		Algorithms				
Benchmark	N=#refs	M	orig	nul	pre	B&K	avl	rb	mrk
adm	460405734	16520	11.81	180.87	703.21	1010.35	1036.01	1128.84	5652.68
arc2d	1961059165	899122	81.13	2680.86	5372.14	6382.12	12391.1	14986.1	>24h
bdna	436191438	191344	17.59	278.16	823.83	1092.76	1137.56	1492.08	61693.60
dyfesm	410077680	16075	19.35	175.87	579.36	998.65	858.14	969.78	>24h
flo52	740622750	207857	10.18	513.1	1427.88	1912.36	2994.88	3744.76	>24h
mdg	2390578661	29053	94.70	836.22	3557.01	6127.57	4576.22	5363.06	4897.80
ocean	1705808702	149990	39.08	1443.31	3625.56	4561.16	4981.03	7343.2	>40h
qcd	150442837	459275	4.69	86.59	268.91	487.7	367.82	500.64	>24h
spec77	2489578773	216430	170.96	1505.91	5008.94	7001.32	7640.15	10653.4	>24h
spice	8228372	63559	2.90	5.24	20.58	34.58	17.03	19.24	34.04s
track	40569358	72548	3.21	23.9	80.51	113.71	87.01	99.88	4570.19
trfd	666159025	675681	9.63	238.33	1139.39	1800.28	1735.31	2474.74	>40h

Table 4.4: Perfect-Club Benchmarks run times (seconds)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We ran our experiments on a 270MHz Ultrasparc Solaris machine. Table 4.4 summarizes the results we obtained. Benchmarks are listed by name: the total number of references and the maximum stack depth are included.

The algorithms we measured are the following:

- `orig` is the runtime of the original non-instrumented benchmark.
- `nul` measures the trace generation overhead. but the stack processing part is not implemented. We measured "nul" to find out how much the benchmarks are affected by just generating the trace.
- `B&K` and `pre` are preallocated implementations of Bennett and Kruskal's. and the preallocated tree hole based algorithm's. respectively.
- `avl` and `rb` are interval tree implementations using AVL and red-black trees respectively.
- `mrk` is the markers algorithm. Many of the numbers are missing because we had to abort runs that were taking too long.

We also show (Figure 4.9) the increase in execution time for all these benchmarks with respect to the optimized execution time of the program. The three bars for each benchmark in Figure 4.9 depict. from left to right. the increase in execution time by adding instrumentation to collect the program trace on the fly. the increase in execution time of our preallocated tree algorithm. and the increase in execution time of our implementation of Bennett and Kruskal's algorithm.

We measured the relative overhead of the stack computation. Figure 4.10 breaks down the total runtime of each benchmark into the time spent in the original benchmark. instrumentation overhead (i.e. time spent generating the program trace). hash table lookup overhead and stack computation overhead.

The interval tree based algorithms have better theoretical bounds than the preallocated tree algorithms. $O(Nlog(M))$ versus $O(Nlog(N))$. There are several reasons why the preallocated algorithms tend to yield better execution times in practice:

- The interval tree implementation severely stresses the memory bandwidth of the host processor. For each element in the program trace the interval tree algorithm generates about

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

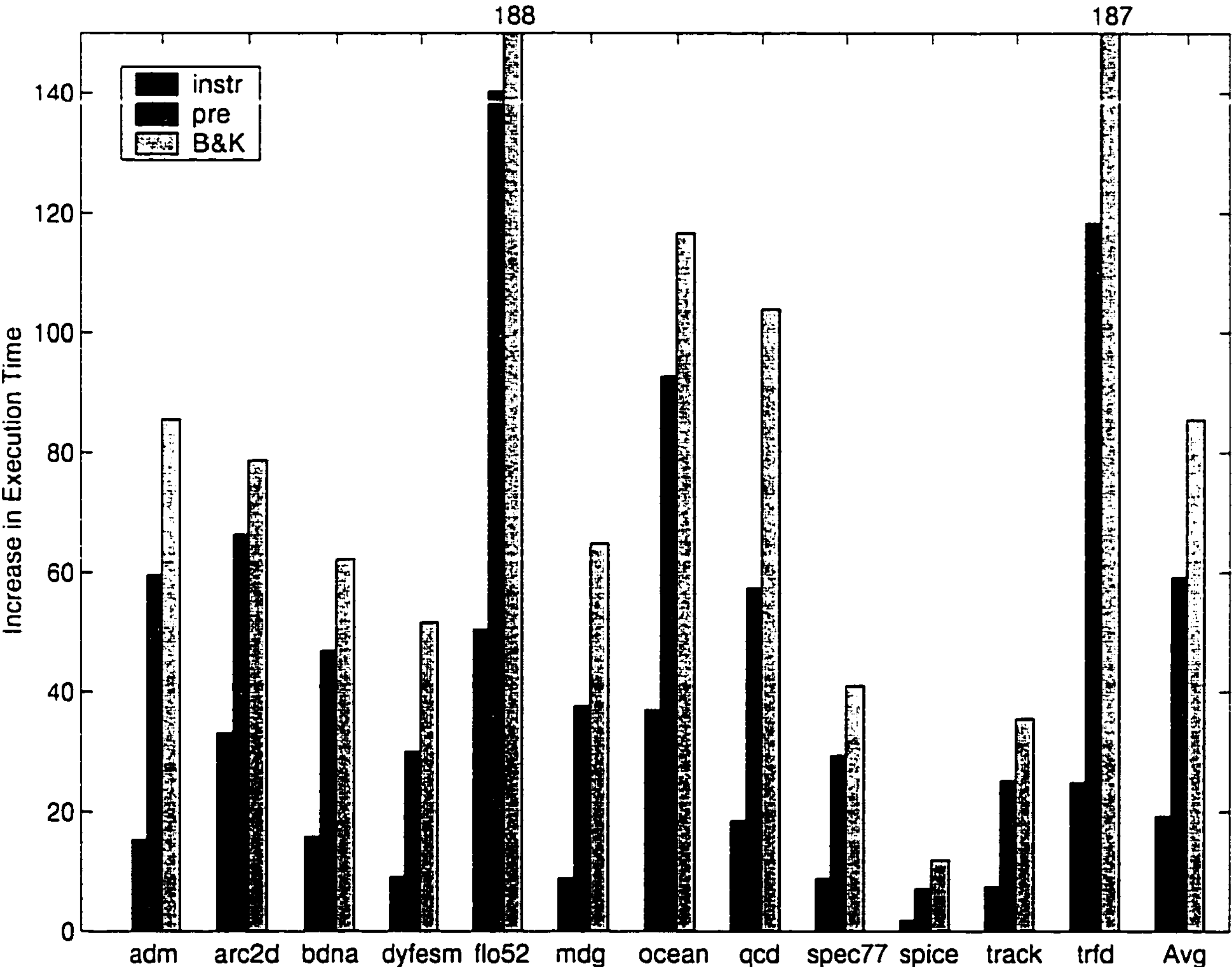


Figure 4.9: Increase in execution time with respect to the optimized program of instrumented code. preallocated hole tree algorithm and Bennett and Kruskal's algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

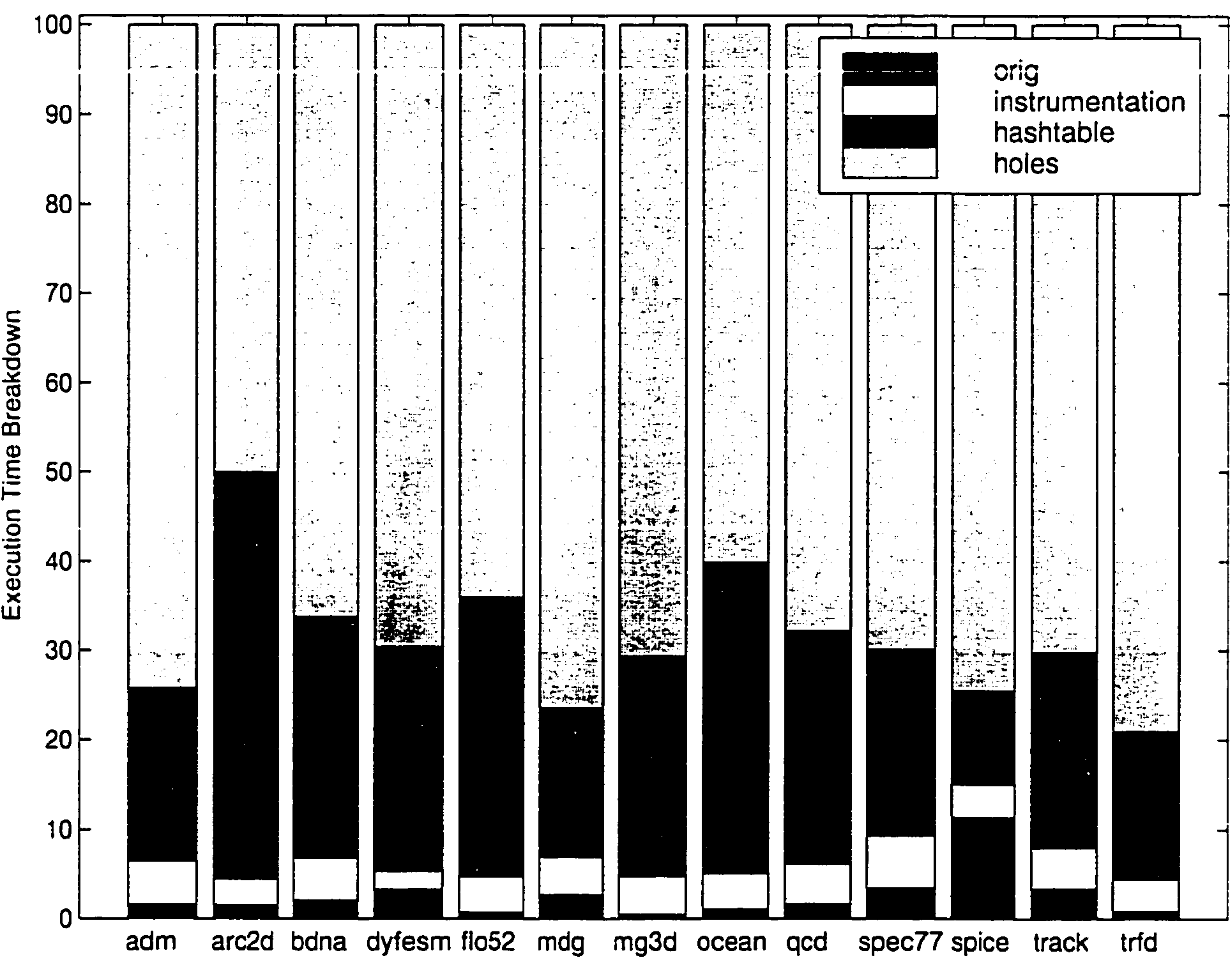


Figure 4.10: Execution time breakdown for the preallocated hole tree algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$3 \cdot log(M)$ additional references while traversing the interval tree: in each tree node at least one node key is accessed; in addition the node's partial sum is accessed and one of the leaves is dereferenced.

The value of M can be approximated with the measured maximum stack depth, which for most of our algorithms yields an AVL tree height of around 20 to 25, resulting in up to 75 extra memory accesses per element in the memory trace. In the case of red-black trees the number of references is even higher.

By comparison the preallocated tree implementation generates only $log(N)$ (or $2 \cdot log(N)$, in the case of Kruskal's algorithm) references. In practice we limited N to 2^{31}, which means 31 memory references for each tree. In addition the preallocated tree is built such that adjacent nodes at lower levels tend to be clustered into the same cache line, resulting in good spatial locality.

- The interval tree implementation relies on dynamic memory allocation as the interval tree shrinks and expands in the course of the process. We were able to gain up to 33% in execution speed by writing our own memory allocators (this gain is included in the performance figures).

The better speed of the preallocated strategy comes, however, at the cost of extremely high memory usage (about 600 MBytes of virtual memory for the preallocated tree) and a hard limit of 2^{31} references in the memory trace. For a few of the benchmarks this limit is exceeded.

The interval tree implementation, if slower, has no inherent limitation with respect to the trace size and delivers reasonable performance. We see it as a more useful tool on the whole. The AVL tree is preferable to the red-black tree, since the higher reordering cost is clearly amortized by the lower average tree height.

In conclusion, the preallocated implementation works better for programs with short traces, bad locality and large cores (that is, large M and relatively small N values), whereas the interval tree implementation works better on long traces with good locality and small cores (larger N, smaller M values).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2.9 Notations

This section enumerates and explains some of the symbols we used throughout this chapter.

- N: number of references in the program trace under consideration.
- M: number of distinct memory references in the program trace. In effect M is equal to the size of the memory used by the program we are analyzing.
- τ: used as the *current index* in the trace. As such, $0 \le \tau < N$. The stack algorithm processes the program trace sequentially: τ always denotes the current position processed by the algorithm.
- x_τ: denotes the memory reference at index τ in the program trace.
- **P**: a mapping from memory references to trace indices. Since **P** changes in time, it is normally indexed with τ, the current index in the program trace.
- **B**: a mapping from trace indices to booleans. $\mathbf{B}_\tau(i)$ is set if at moment τ the location referenced at position i in the trace is the latest reference to its location.
- $dist(\tau)$: the stack distance corresponding to the location referenced in position τ in the trace. This is the number we compute for each element in the trace.
- $holes_\tau(i)$: the number of holes in the program trace between indices i and τ at moment τ. $i < \tau$ by definition.

4.3 Summary

In this chapter we have presented a new metric for data locality based on the stack distances. We have shown that the stack distance metric captures locality more precisely than the inter-reference distance metric [42]. It is also applicable to any program granularity, as opposed to other metrics proposed, such as the number of accesses in the inner-most loop [73, 72].

The second part of the chapter discussed a new algorithm for stack processing. The new algorithm, using preallocated trees, improves over the best known stack algorithm [6] (which also

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

uses preallocated data structures) by more than 30%. If the size of the trace is not know before running the algorithm. the preallocated data structures are not the best choice. Therefore we propose another scheme. based on the same algorithm. that uses AVL trees. The performance of this scheme is only marginally worse than the previously best known algorithm using preallocated trees. while using less memory for short traces. and giving the possibility to grow the data structures as needed for long traces.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5

Polaris Performance Prediction Framework

In this chapter we present the details of our implementation of the performance prediction framework inside the Polaris source-to-source restructurer (Section 5.1). We also provide a description of the interface between the performance prediction framework in Polaris and the SvPablo performance visualization tool (Section 5.2).

5.1 The Polaris Framework

The Polaris performance prediction framework consists of a collection of classes that allow easy implementation of models for loop based compile-time performance prediction. Its main use is as a compiler pass that can be called whenever the need for performance prediction data occurs. It also provides support for profiling information collection and performance data reporting and visualization.

The main design goals for the performance prediction framework are:

- **modularity** – we want to provide a basis for developing performance estimation modules. A module can focus on a specific part of a computer system, such as CPU, memory, or I/O system, and model that part at the desired level of detail. In previous chapters we have presented algorithms that are implemented as three such modules: the CPU model, and two memory models, the stack distances and the indirect accesses models.

- **consistency** – the framework contains abstract base classes for performance estimators, thus

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

guaranteeing that whenever a new module is developed it provides the necessary basic functionality, consistent with the rest of the system. Since our prediction model for a computing system consists of several models that estimate the behavior of different subsystems symbolically, the symbolic expressions that represent the performance must be compatible with each other. The base classes ensure this property.

- **ease of use and maintenance** – we designed the framework in such a way that one can "unplug" a performance estimator module and "plug" another one in place very easily, while maintaining the code readability.

In the following discussion we assume that the reader is familiar with the Polaris internal representation [26]. Polaris is a source-to-source restructurer that parses Fortran 77 and outputs Fortran with parallel directives for a large set of platforms. For more details on Polaris, see [8].

In Figure 5.1 we present the UML diagram of the classes contained in the framework, and Table 5.1 details the functionality implemented by the methods. The `PerformanceEstimator` object is the interface to the performance prediction module. This object can be instantiated either for a Fortran program unit (subroutine or function), or for a particular statement in the program, and to access it, we have added the `get_perf_estimator` method to both `ProgramUnit` and `Statement` Polaris objects. A `PerformanceEstimator` object contains a collection of `CostEstimator` objects, that is called to estimate the performance of a block of code. Each `CostEstimator` implements a particular performance prediction model, as discussed in Chapter 3. The user registers the cost estimators corresponding to the performance data desired before calling the `estimateCost` member function on the `PerformanceEstimator` object.

For example, if performance data for the CPU execution time is needed for a loop, the following code estimates it:

```
PerformanceEstimator *pe = loop->get_perf_estimator();
pe->registerCostEstimator(new OpsCostEstimator(loop, pgm));
pe->estimateCost();
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

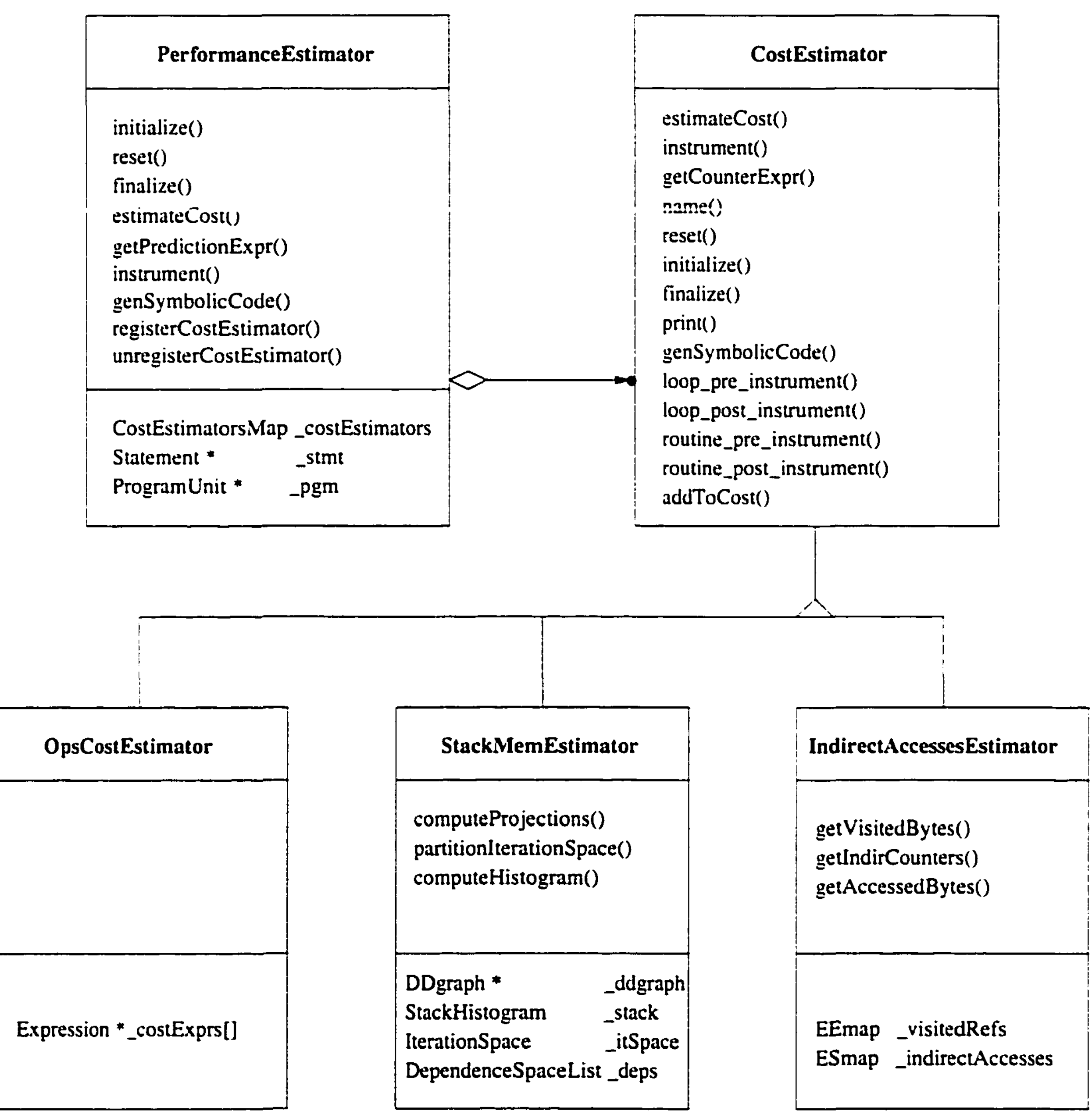


Figure 5.1: Polaris performance prediction framework

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Object	Method	Comment
PerformanceEstimator	initialize	initializes the data structures for the PerformanceEstimator and its cost estimators
	reset	resets all the cost estimators
	finalize	frees all the allocated data structures in the PerformanceEstimator and its cost estimators
	estimateCost	estimates the cost by calling the estimateCost method on every registered cost estimator
	getPredictionExpr	returns the symbolic expressions denoting the prediction
	instrument	places instrumentation in the Fortran code to evaluate the prediction expressions
	genSymbolicCode	generates C++ code to evaluate the symbolic expressions as described in Section 5.2
	un/registerCostEstimator	registers and unregisters a cost estimator with this PerformanceEstimator object
CostEstimator	initialize	initializes the data structures for this CostEstimator
	reset	resets the cost estimator
	finalize	frees all the allocated data structures
	name	returns a unique name for the cost estimator
	estimateCost	traverses the AST and generates cost expressions denoting performance
	instrument	places instrumentation in the Fortran code to evaluate the prediction expressions
	getCounterExpr	returns the symbolic expressions for performance
	genSymbolicCode	generates C++ code to evaluate the symbolic expressions
	loop_*_instrument	used by the instrument method to place instrumentation before and after estimated loops
	routine_*_instrument	used by the instrument method to place instrumentation before and after estimated routines
	addToCost	adds the cost expressions of the argument to this estimator. Used to aggregate costs for block statements

Table 5.1: Method functionality in the Polaris performance prediction framework

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Object	Method	Comment
StackMemEstimator	partitionIterationSpace	partitions the iteration space of the loop as described in Section 3.3.3
	computeProjections	computes the dependence spans and array sections (see Sections 3.3.4 and 3.3.5)
	computeHistogram	computes the stack histogram. as described in Section 3.3.6
IndirMemEstimator	getVisitedBytes	returns an expression denoting the number of bytes accessed by array references
	getIndirCounters	returns the list of symbols that correspond to indirect array accesses
	getAccessedBytes	returns the sum of all the bytes accessed by arrays in a loop

Table 5.2: Method functionality for memory cost estimators

After the cost estimation is performed. the user can perform several tasks. depending where and in what format the performance data is needed:

- *inside the compiler* - if performance data is needed to guide optimizations (or for any other task at compile-time). the user can retrieve the performance symbolic expressions using the `getPredictionExpr()` method on the `PerformanceEstimator` object. This method will aggregate the cost expressions of all its registered cost estimators into one expression.

- *run-time estimation and profiling information collection* - the user can call the `instrument()` method to obtained an instrumented version of the code. The instrumentation consists of statements to evaluate all the cost expressions in each cost estimator plus statements to collect the performance data. The performance data is collected using calls to functions implemented as a separate library [13].

- *performance visualization and/or scalability analysis* – in case the user wants to store the symbolic expressions and evaluate them at a later time using different data sets. the `genSymbolicCode()` method can be used. The method will generate C++ code that contains the symbolic expressions. An example of how this code can be used for performance visualization is presented in Section 5.2.

As we mentioned before. the `PerformanceEstimator` object can be created for a `ProgramUnit` (an object that represents a function or a subroutine in Polaris) or for a statement. including block

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

statements. such as loop nests or if statements. The `PerformanceEstimator` object takes care of aggregating the cost expressions for all the statements contained in the block.

Three cost estimators are provided. that implement the performance prediction models discussed in Chapter 3. The `OpsCostEstimator` implements the processor model. The `StackMemEstimator` implements the Stack Distances model. and the `IndirectMemEstimator` implements the Indirect Accesses Model.

5.2 Integration with SvPablo

SvPablo [21] is a language independent performance analysis and visualization system. We have used the SvPablo system together with Polaris as an example of the integration between performance visualization tools and compilers. SvPablo is capable of instrumenting code. either interactively or automatically. compile. run. collect and summarize performance data for the instrumented statements. as well as displaying the correlation between performance data and the source code in an easy and intuitive user interface. On the other hand. the Polaris performance prediction framework can analyze and extract performance information at compile-time and represent this information using symbolic expressions.

The "marriage" between these two systems provides a very powerful integrated system for performance tuning. One of the major drawbacks of running instrumented code to collect performance information is that the instrumentation code perturbs compiler optimizations and cache behavior. Having the performance information computed at compile-time and stored as symbolic expressions makes the instrumentation code no longer necessary. therefore. there will be no perturbations. Also. the performance data is no longer collected for a unique data set. and thus. we enable other analyses. such as scalability analysis and "what if" questions and answers. to be performed on the code.

The key ideas that enable the integration are:

1. the architectural independence of the performance prediction model implemented in Polaris

2. the symbolic representation of the performance data

3. the language independence and extensibility of SvPablo due to its SDDF meta-format [2]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

representation of performance information

Currently there are two modes in which Polaris and SvPablo interact. In the first mode. Polaris analyzes the code and generates an instrumented program. such that. by running the instrumented program the symbolic expressions that represent performance data are evaluated. The performance data is summarized in an SDDF file. and SvPablo uses this file to relate the performance data to the source code of the program.

The second mode of integration is more involved. In this scenario. Polaris generates code. separate from the analyzed program. to store the symbolic expressions representing performance prediction data. The code can be compiled in a separate library for later use. We have chosen C++ as the language in which to store the expressions. SvPablo will implement the user interface part that makes calls to the library in order to estimate the performance for different machine parameters and/or input data sets.

The class diagram shown in UML notation in Figure 5.2 is the interface to the symbolic expressions library. Because of the separation of program data and machine parameters. there are two class hierarchies in the diagram.

The first hierarchy. based on the class `DelphiMachineDescription` encapsulates architectural specifications. For each processor/system configuration. an SDDF file with the machine specification is provided. This file describes details such as processor clock frequency. number of functional units. repeat rates and latencies of operations. issue width. etc. It also specifies the memory hierarchy: how many levels of cache. the parameters for each cache level. such as: latency of a hit and a miss. associativity. line size and total size. Being written in SDDF. it is extensible. to allow for adding I/O specifications and interprocessor communication. A customized parser extracts the specifications from the SDDF file and generates the machine description classes.

Two of the methods in the machine description class are `getOpsCosts()` and `getCacheCosts()`. An example of their usage is presented below. The method `getOpsCosts()` takes the estimated number of operations. and combines the operations with their latencies and repeat rates to give an estimated execution time. `getCacheCosts()` takes the number of estimated cache misses for each level in the memory hierarchy, and returns the estimated time spent accessing the memory hierarchy. Both these methods are used by the second class hierarchy to extract information about

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

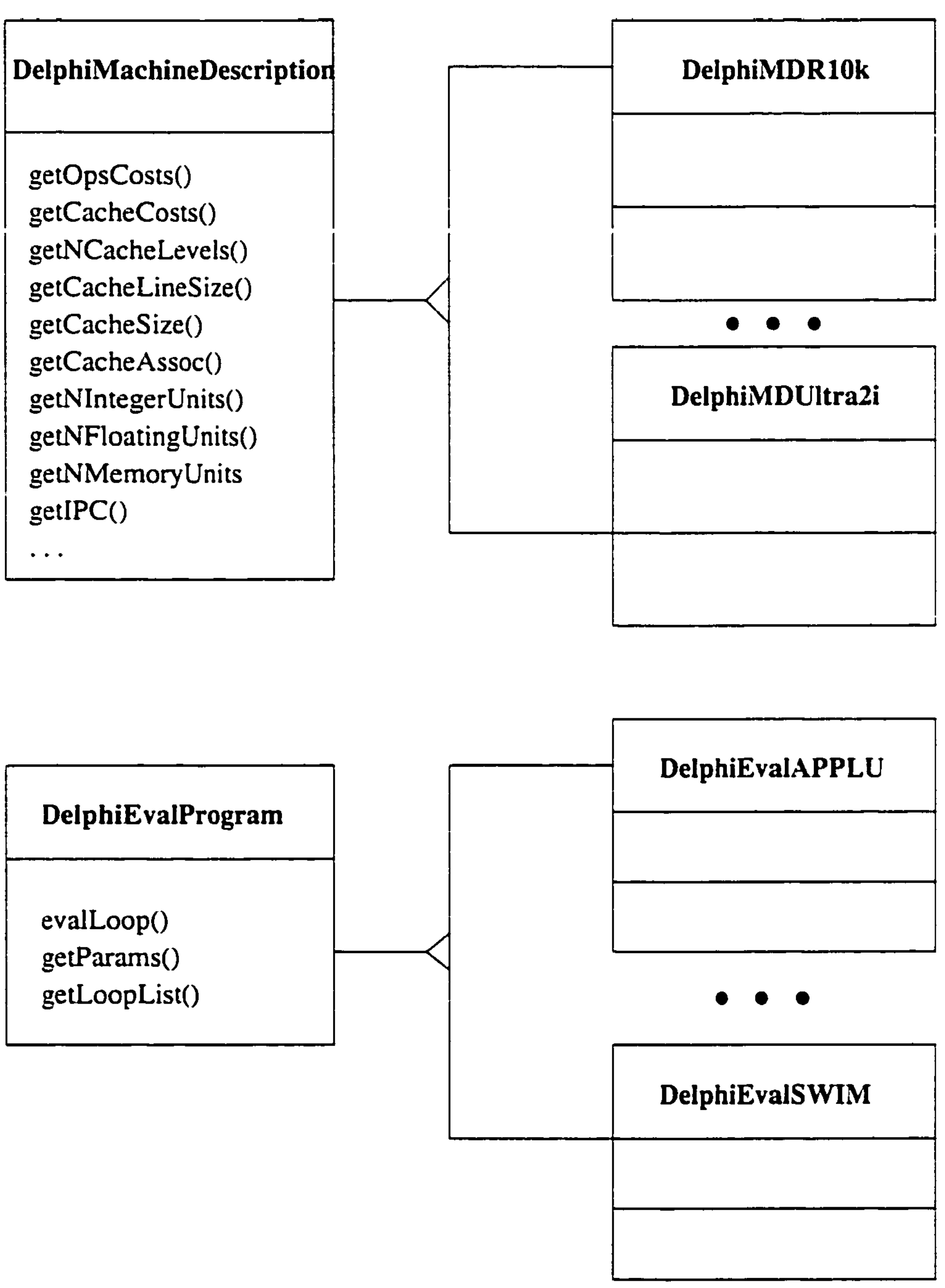


Figure 5.2: Polaris performance prediction interface to SvPablo

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the machines.

```
double eval_CALC1_do100(DelphiParamList &params, DelphiMachineDescription &md,
   DelphiEvalProgram &evalPgm)
{
  double result = 0.0;
  DelphiParamList::const_iterator pIter;
  pIter = params.find("N");  double n = (*pIter).second;
  pIter = params.find("M");  double m = (*pIter).second;

  double opCosts[29];
  for(int i = 0; i < 29; i++)
     opCosts[i] = 0.0;

  opCosts[1] = 1+n;
  opCosts[2] = 22*m*n;
  opCosts[7] = 23*m*n;
  opCosts[8] = m*n;
  opCosts[18] = 14*m*n;
  result += md.getOpsCosts(opCosts);

  int nlabels = 27;
  double *labels = new double[nlabels];
  double *refs   = new double[nlabels];
  double cacheMisses[MAX_CACHE_LEVELS];
  for(int i = 0; i < md.getNCacheLevels(); i++) {
     cacheMisses[i] = 0.0;
     int _delphi_cls = md.getCacheLineSize(i);
     labels[0] = 2; refs[0] = n/(_delphi_cls/8);
     labels[1] = 0; refs[1] = 8+m+m/(_delphi_cls/8)+6/(_delphi_cls/8)+9*n+
       n/(_delphi_cls/8)+7*n*m/(_delphi_cls/8);
     labels[2] = 1; refs[2] = -2-4*n-3/(_delphi_cls/8)+m/(_delphi_cls/8)+
       m-4*n*m/(_delphi_cls/8)-n/(_delphi_cls/8)+4*m*n;
     labels[3] = 3; refs[3] = 1/(_delphi_cls/8);
     labels[4] = 5; refs[4] = -1+m-n-1/(_delphi_cls/8)+m/(_delphi_cls/8)+m*n
       -n*m/(_delphi_cls/8);
     labels[5] = 1+1/(_delphi_cls/8); refs[5] = 4*m*n;
     labels[6] = 7; refs[6] = m*n-(n+n*m/(_delphi_cls/8));
     labels[7] = 4+4/(_delphi_cls/8); refs[7] = 1;
     labels[8] = 6; refs[8] = m*n-(n+n*m/(_delphi_cls/8));
     labels[9] = 5+5/(_delphi_cls/8); refs[9] = 1;
     labels[10] = 9+9/(_delphi_cls/8); refs[10] = m+2*m*n;
     labels[11] = 3+3/(_delphi_cls/8); refs[11] = n;
     labels[12] = 6+6/(_delphi_cls/8); refs[12] = 1+m*n;
     labels[13] = 11+11/(_delphi_cls/8); refs[13] = -1+m;
     labels[14] = 8+8/(_delphi_cls/8); refs[14] = m;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    labels[15] = 13+13/(_delphi_cls/8); refs[15] = -1+m-n+m*n;
    labels[16] = 15+15/(_delphi_cls/8); refs[16] = -n+m*n;
    labels[17] = 16+16/(_delphi_cls/8); refs[17] = -1-n-1/(_delphi_cls/8)+m*n;
    labels[18] = 18+2/(_delphi_cls/8)+17/(_delphi_cls/8); refs[18] = -1+m;
    labels[19] = 35+4*m+m/(_delphi_cls/8); refs[19] = -1+n;
    labels[20] = 17+17/(_delphi_cls/8); refs[20] = -1+m-n+m*n;
    labels[21] = 26+4*m+m/(_delphi_cls/8); refs[21] = -m+m*n;
    labels[22] = 25+m/(_delphi_cls/8)+4*m; refs[22] = m*n-1;
    labels[23] = 28+4*m+m/(_delphi_cls/8); refs[23] = -m+m*n;
    labels[24] = 25+4*m+m/(_delphi_cls/8); refs[24] = -m+m*n;
    labels[25] = 24+m/(_delphi_cls/8)+5*m; refs[25] = -2*m+2*m*n;
    labels[26] = 25+4*m+m/(_delphi_cls/8); refs[26] = -2*m-1+3*m*n;
    for(int j = 0; j < nlabels; j++) {
      if(labels[j] > md.getCacheSize(i) || labels[j] == 0)
        cacheMisses[i] += refs[j];
    }
  }
  delete [] labels; delete [] refs;

  result += md.getCacheCosts(cacheMisses);
  return result;
}
```

The other class hierarchy. based on class `DelphiEvalProgram` represents performance data on the program side. Polaris is used to generate a class for each analyzed program. For each loop nest in the program. Polaris computes the symbolic expressions denoting the predicted performance. and generates a function that can be used to compute the predicted execution time. provided the symbolic parameters are given actual values.

Two methods stand out in the `DelphiEvalProgram` object. **`getParams`** takes a loop name and returns the list of symbolic parameters that make up the prediction expressions. These parameters have to receive values in order evaluate the expressions in a performance figure. The other method is **`evalLoop`**. which. given a loop name. a list of parameters with the values filled in. and a machine description. returns the predicted execution time of the loop with the specified parameters on a given machine.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.3 Summary

In this chapter we have presented the Polaris performance prediction framework. The framework enables access to compile-time performance prediction from inside the compiler. It also allows the generation of instrumentation for collecting profiling information. We have also discussed how the framework helps the integration between the compile-time data prediction module with the SvPablo performance visualization tool.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 6

Experimental Results

6.1 Experimental Setup

To evaluate the accuracy of the performance prediction models described in Chapter 3. we implemented these models within the Polaris performance prediction framework. Since Polaris did not have profiling information support. we have also implemented instrumentation passes to collect profiling data. such as branch frequencies and loop bounds.

To conduct the experiments. two other instrumentation passes were added to Polaris. One is used to collect values for the hardware counters. The other is used to measure execution time. Both these passes can instrument a selected set of loop nests. procedures or entire programs. If a procedure is selected. all the loop nests in the procedure are instrumented. The results are reported for the selected program constructs.

To validate the compile-time estimation of the number of cache misses using the stack distances model. a run-time version of the stack distances algorithm was implemented. Currently. the compile-time version of the algorithm works only intra nest. Therefore. when results are reported for entire procedures. the run-time version of the stack distances algorithm was used.

Two systems. based on two different processors. were used to carry-out the experiments. The first system. an Origin 200. consists of 4 MIPS R10000 processors running at 195 MHz. Each of the processors is a superscalar processor capable of issuing 6 instructions per cycle and executing 4 of those. Instructions are issued out-of-order and retired in-order. Each processor has a 32 KB 2-way set-associative L1 data cache. with a 32 bytes cache block. The L2 cache is 1 MB. 2-way set-associative with a 128 bytes cache block. The L2 cache is unified. i.e.. contains both instructions

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and data. The compiler used on this system is the MIPSpro 7.30 Fortran compiler. This system was also used to collect hardware counters measurements.

The second system is a Sun Enterprise server. It has 4 UltraSparc IIi processors running at 250 MHz. The UltraSparc processor is an in-order superscalar, capable of issuing and executing up to 4 instructions per cycle. The caches on this processor are as follows: the L1 cache is a 16 KB direct mapped data cache, with 32 bytes block size. The L2 cache is a 1 MB direct mapped unified cache with 64 bytes block size. The compiler used on this system is the SparcWorks 4.0 Fortran compiler.

In the following sections, performance prediction of cache misses and execution time are presented. We derived the symbolic expressions representing performance data measured on the UltraSparc, and we substituted the machine parameters for both the R10000 and the Ultrasparc to obtain the performance numbers in these expressions. The machine parameters were taken from the processors' manual [56, 66] and, when not available, determined using micro-benchmarks.

Two types of comparisons are made. First, for each memory hierarchy model, the cache miss estimations are compared against hardware counters values on the R10000 processor. Then, the memory model is combined with the CPU model to predict the execution time for both the R10000 and the UltraSparc.

6.2 Results

6.2.1 Cache Miss Prediction with the Indirect Accesses Model

To quantify the accuracy of the prediction model using the Indirect Accesses model we have chosen SpLib [9], a public domain implementation of several iterative methods for solving sparse linear systems of equations. From this package, we have selected the loops nests that take the most time when solving a sparse linear system using the stabilized bi-conjugate gradient algorithm with an incomplete LU factorization preconditioner. It happens that these loops also satisfy the requirement of having no I/O calls. The routines in which the nests are located are:

- `BMUX` – multiplies a sparse matrix with a vector using the dot product form. The sparse matrix is stored in compressed sparse row (CSR) format;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

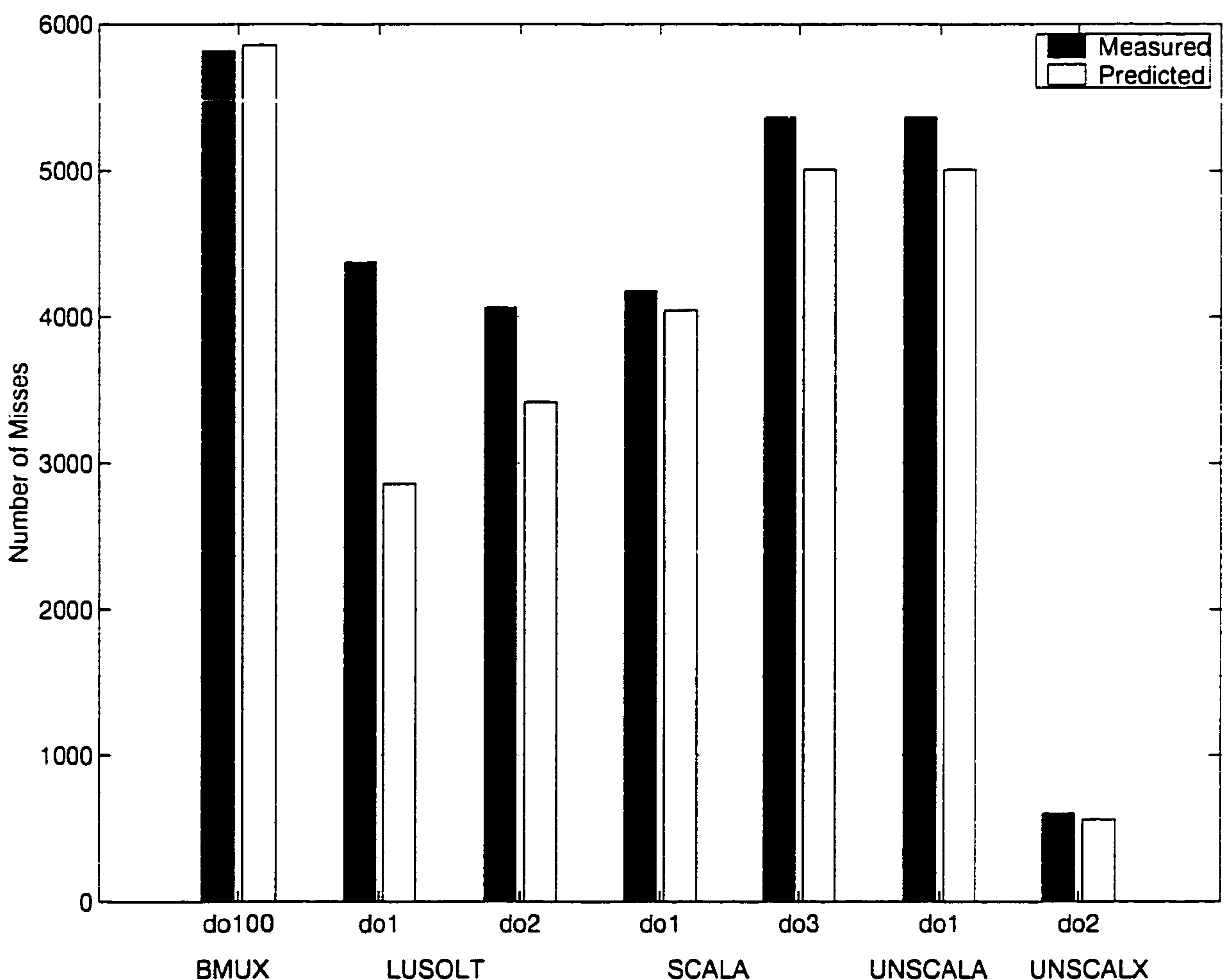


Figure 6.1: SpLib – L1 cache miss prediction for the small data set

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- `SCALA` – scales a sparse matrix stored in CSR to have certain properties. such as scaling each column so that the diagonal entry is 1.0:

- `UNSCALA` – unscales a CSR matrix to reverse effects of SCALA:

- `UNSCALX` – unscales the solution vector at end of computations

- `LUSOLT` – performs a forward then backward solve for a modified sparse row (MSR) matrix containing a unit lower triangular and an upper triangular matrix with inverted diagonal. both stored in a single MSR data structure. The first loop nest (`do1`) performs the forward solve. and the second loop nest (`do2`) performs the backward solve.

We ran the benchmark using two data sets:

- a small data set – a 1128 × 1128 sparse matrix with 13360 non-zero elements

- a large data set – a 20284 × 20284 sparse matrix with 452752 non-zero elements

Figures 6.1 and 6.2 show the predicted number of cache misses. using the indirect accesses model. compared to measured cache misses for the L1 cache on the MIPS R10000 processor. The actual number of misses is obtained using the hardware counters on this processor. Figure 6.1 shows the results for the small data set. while Figure 6.2 shows the results for the large data set. We observe very little variation with the increase in the data set size. which shows that the model handles quite well even large variations in the input data set.

The average prediction error for the cache miss estimation on the L1 cache is 10.60% (standard deviation 11.62%) on the small data set. and 9.41% (standard deviation 11.25%) on the large data set. which is quite good considering that we model a fully associative cache. and the caches for the R10000 processor are two way set-associative. The L2 cache miss estimations are shown in Figure 6.3. For the L2 cache we show only the large data set because the entire small data set of the application fits in the 1 MB cache of the processor. Again. we see a good correlation between the measured and predicted data. The average prediction error for the L2 cache is 12.65%. with an 18.46% standard deviation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

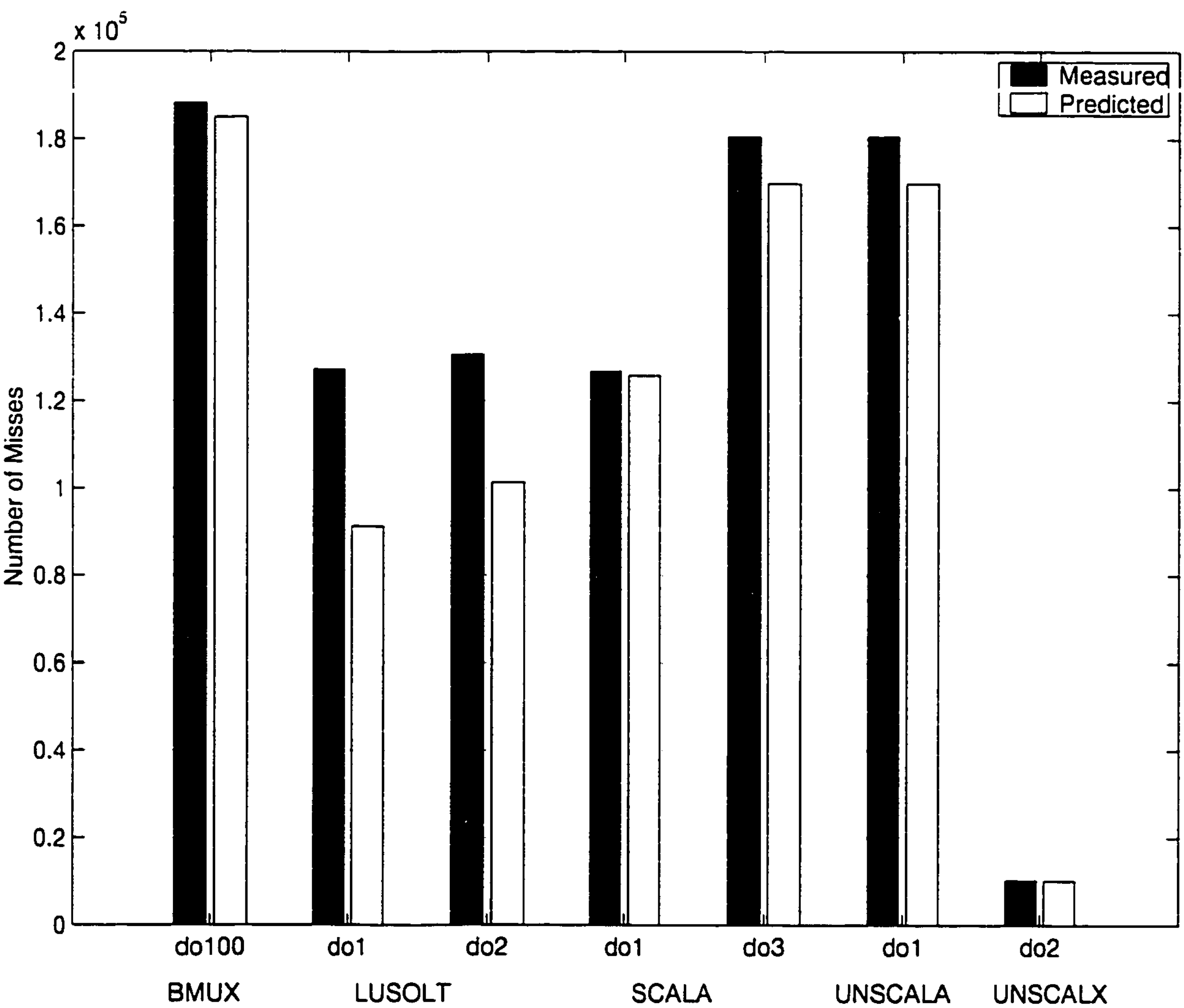


Figure 6.2: SpLib – L1 cache miss prediction for the large data set

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

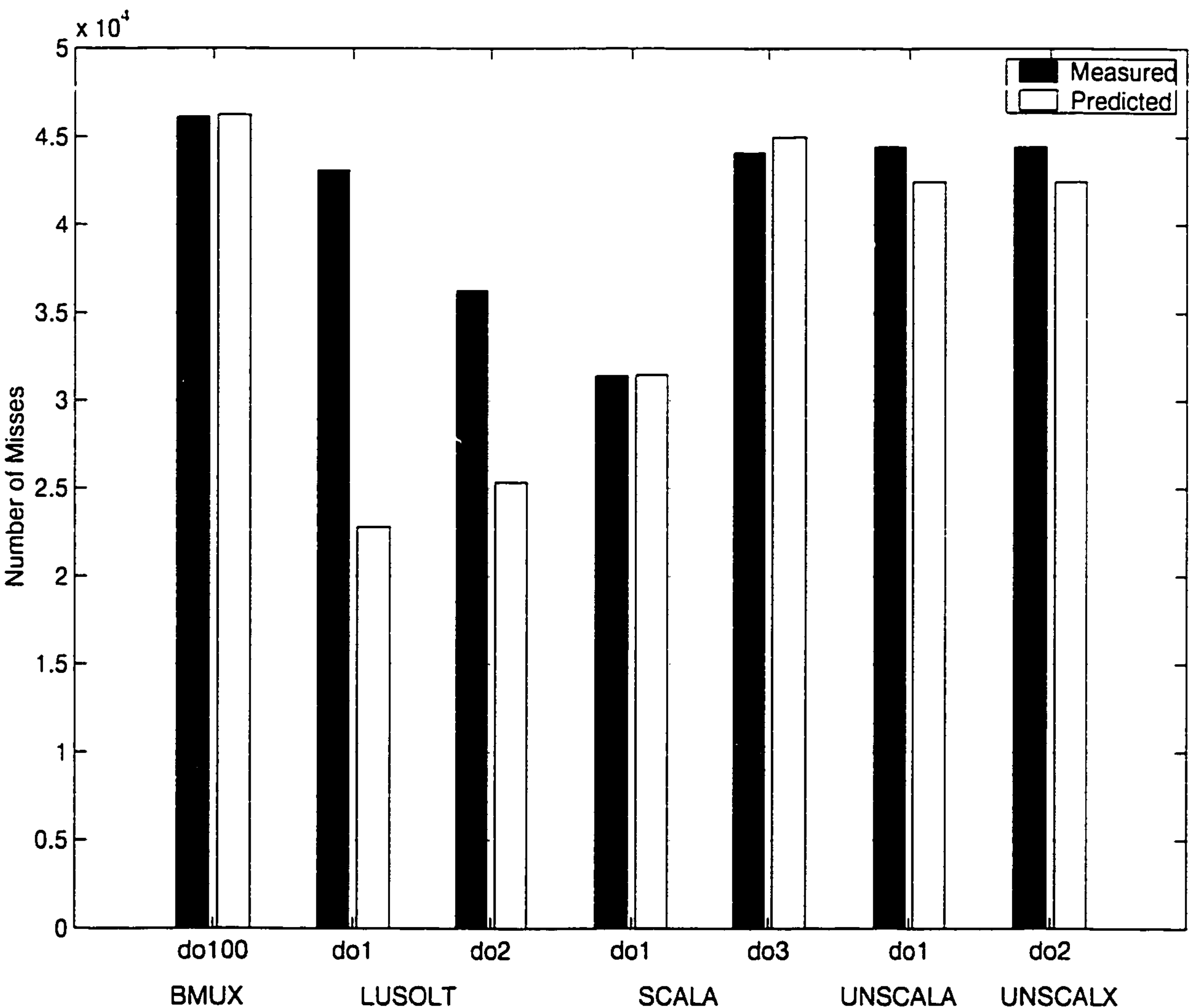


Figure 6.3: SpLib – L2 cache miss prediction for the large data set

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.2.2 Execution Time Prediction with the Indirect Accesses Model

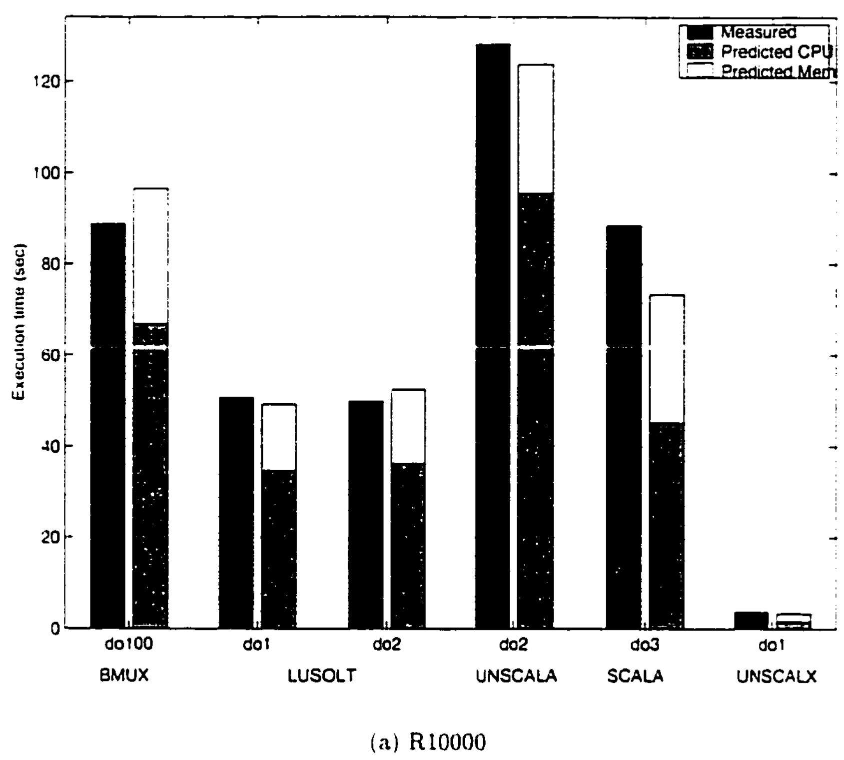
The next set of figures presents the execution time prediction for the same loops in the SpLib package. The figures compare the measured execution time for each loop with the predicted execution time obtained by combining the CPU model with the Indirect Accesses memory model. Each bar for the predicted execution time shows the breakdown into CPU predicted time and memory predicted time. The memory predicted time includes prediction for both levels of cache.

Two sets of results are presented for two processor-compiler combinations. Figures 6.4a and 6.4b show the predicted execution time compared to the measured execution time for unoptimized codes on the MIPS R10000 and UltraSparc II*i* processors. respectively. The compilers used are the F77 MIPSpro 7.30 and SparcWorks 4.0. respectively. with the default levels of optimizations (i.e.. no -O flag was used). We do not apply any of the optimization heuristics described in Section 3.1 in our prediction. The data set is the large data set described above.

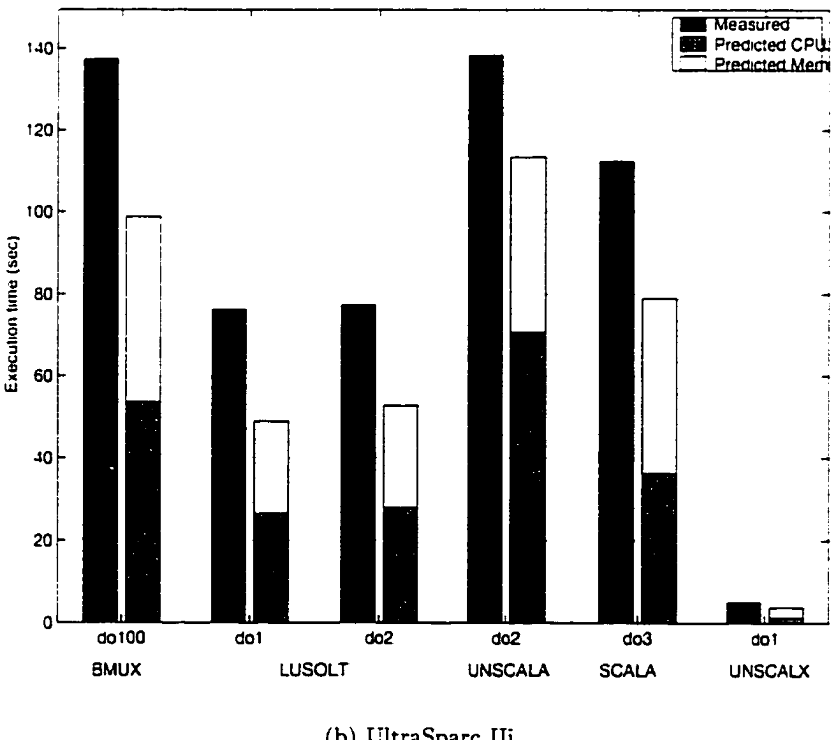
We notice that the prediction is less accurate on the UltraSparc processor than on the R10000 (the average prediction error is 8.10% for the R10000 and 28.04% for the UltraSparc). There are two reasons for underestimating the performance: first. the caches on the UltraSparc are direct mapped. while our model predicts misses for fully associative caches. The second reason. is that the SparcWorks compiler. without optimizations enabled. generates a large amount of redundant code (register spills and redundant conversions from single to double precision) that is not taken into account by our high level language model. The prediction accuracy improves when optimizations are turned on.

Figures 6.5a and 6.5b show the predicted execution time relative to the measured execution time for optimized codes. The same compilers are used. this time with optimizations enabled by the -O2 flag. The prediction model also applies all the optimization heuristics discussed in Section 3.1. The average prediction error is 16.32% for the R10000 and 17.81% for the UltraSparc.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



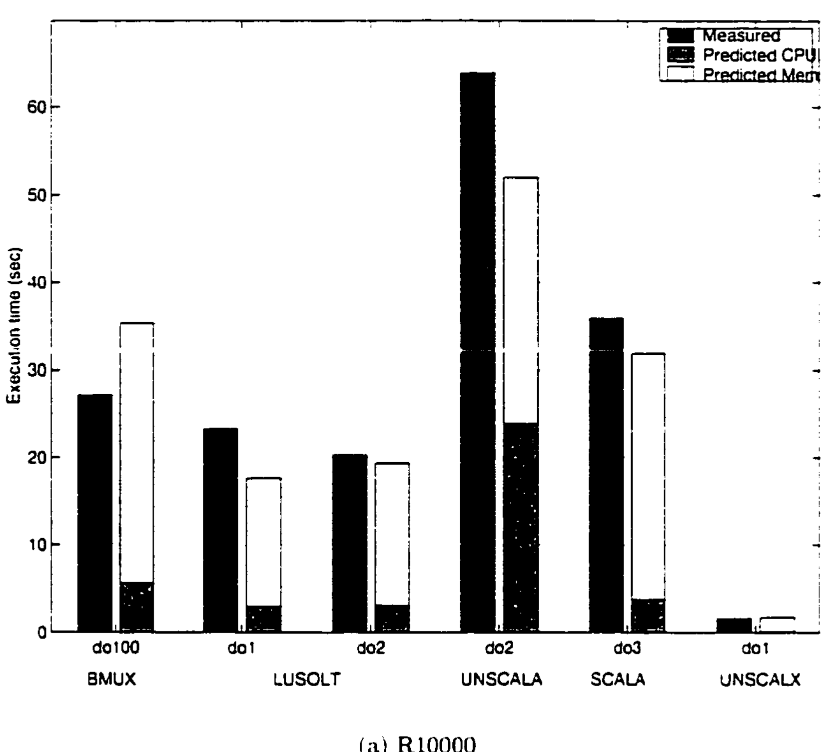
(a) R10000



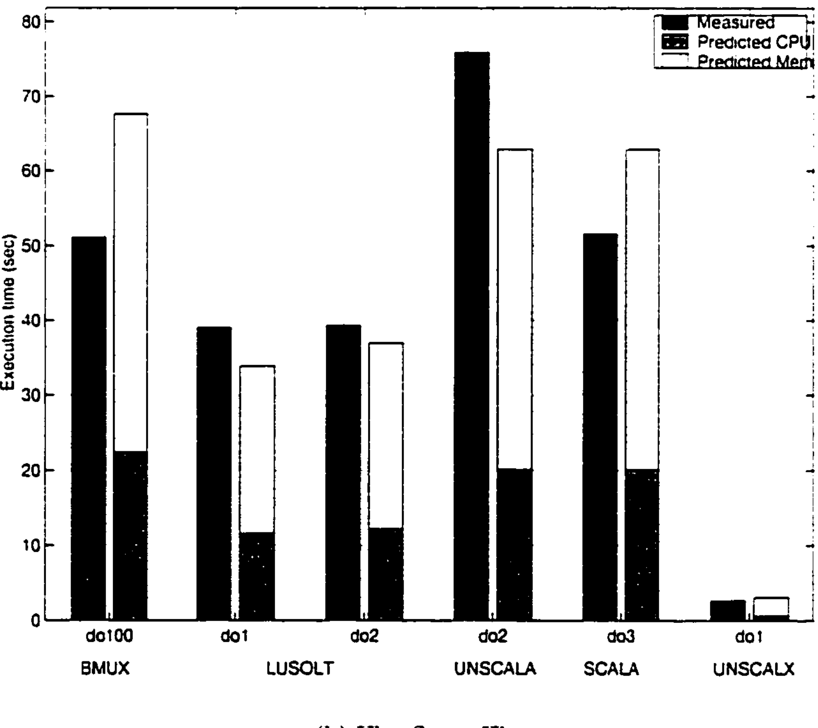
(b) UltraSparc IIi

Figure 6.4: SpLib – unoptimized execution time prediction accuracy

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a) R10000



(b) UltraSparc IIi

Figure 6.5: SpLib – optimized execution time prediction accuracy

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.2.3 Cache Miss Prediction with the Stack Distances Model

In this section we present prediction results using the stack distances models. We compare the cache misses obtained using the stack distances model to the number of actual cache misses obtained using hardware counters on the MIPS R10000 processor.

In Figure 6.6 we look at the predicted versus measured number of cache misses for a Jacobi relaxation code. shown below:

```
do j = 2, n-1
   do i = 2, n-1
      a(i,j) = (a(i-1,j) + a(i+1,j) + a(i,j-1) + a(i,j+1))/4.0
   enddo
enddo
```

On the x-axis is the problem size. increasing from 128×128 to 2048×2048. The prediction is quite accurate. on both levels of cache. except for the largest size. where there are many conflict misses in the L1 cache. The miss-prediction comes from the fact that we model a fully associative caches. and the cache is only 2-way set-associative. On the bigger L2 cache the effect does not occur until the matrix is much larger.

In the remainder of this section we look at loops from the SPECfp95 benchmark suite.

In Table 6.1 we present a summary of the number of loops analyzed and estimated by Polaris for the SPECfp95 benchmarks. For each benchmark. the first two columns are the total number of loops present in the program and the number of loops that are "predictable". i.e. do not contain I/O operations. In parenthesis we show the percentage of the total execution time taken by the measured loops in the column. The next columns show the distribution of the estimated loops based on the amount of compile-time information available. "Full" means that Polaris was able to compute the data dependence distance vectors for all array references in the loop. and all the distances are constant. i.e.. these are loops containing uniformly generated dependences. "Partial". represents those loops for which all the dependence distances were computed. but some

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

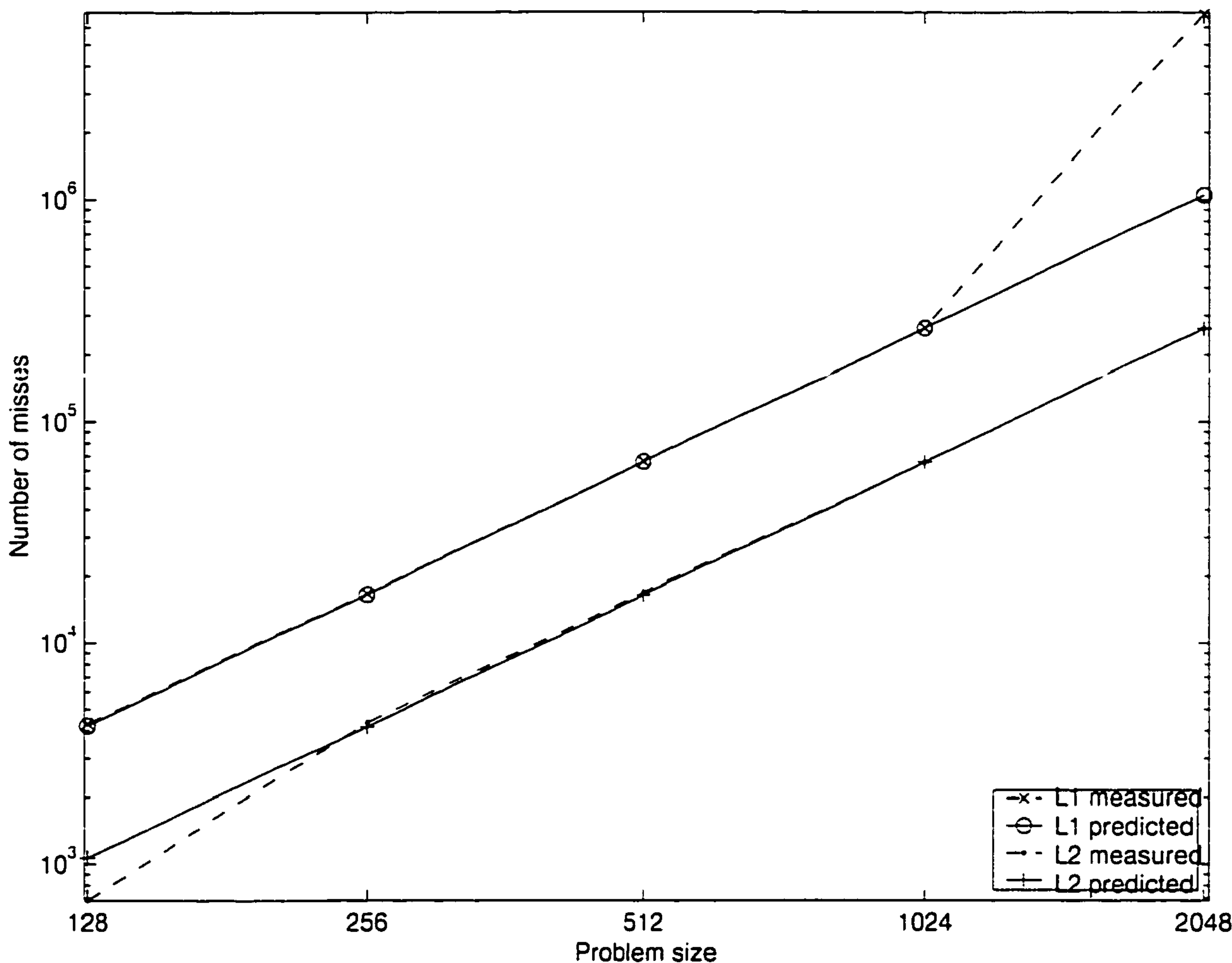


Figure 6.6: Jacobi - cache miss prediction on the R10000

dependences have non-constant distances. For both these cases we can apply the stack distances model to predict the number of cache misses. In the second case we assume that accesses take place at the minimum distance. "Not available" represents the case in which Polaris could not compute the dependence distances for all the array references due to limitations of the Omega test. such as subscripts of subscripts or non-affine subscripts. "Need profiling" is the case in which the compiler needs run-time data due to unknown branch frequencies.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Benchmark	Total Loops	Loops without I/O (% exec)		Compile-time Information (% exec)						Need Profiling	
				Full		Partial		Not available			
APPLU	168	149	(99.67)	137	(46.19)	8	(15.69)	0	(0.00)	4	(37.79)
APSI	316	249	(58.73)	189	(41.68)	33	(10.81)	7	(0.00)	20	(6.25)
HYDRO2D	183	176	(98.41)	142	(27.40)	15	(36.96)	6	(1.99)	13	(32.07)
MGRID	98	88	(97.33)	87	(97.33)	0	(0.00)	1	(0.00)	0	(0.00)
SU2COR	117	82	(90.61)	56	(45.33)	11	(27.77)	1	(0.00)	14	(17.52)
SWIM	24	24	(99.84)	24	(99.84)	0	(0.00)	0	(0.00)	0	(0.00)
TOMCATV	16	12	(96.01)	11	(90.94)	1	(5.07)	0	(0.00)	0	(0.00)
TURB3D	77	67	(48.69)	40	(3.81)	2	(0.00)	20	(44.88)	5	(0.00)
WAVE5	382	354	(75.82)	231	(31.54)	53	(10.69)	21	(12.07)	49	(21.52)
Total (avg)	1381	1201	(95.64)	917	(60.51)	123	(13.37)	56	(7.37)	105	(14.39)

Table 6.1: Compile-time stack distances accuracy

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

There are two important conclusions that can be drawn by looking at the data in Table 6.1. First. most of the loops in this benchmark suite are analyzable by the compiler (86.59%). This percentage includes both cases in which the stack distances algorithm can be applied. These loops make-up. on average. about 74% of the total execution time of the benchmarks. This shows that our method has quite a wide range of applicability. Second. most of the remaining loops. need profiling information due to the presence of if statements within the loop body. The indirect accesses model presented in this work can handle these loops.

One other observation is related to the dependence distances computed by Polaris. It turns out that most accesses happen at very small distances. i.e.. one or two iterations of the loop carrying the dependence. That means that our estimations will not be affected very much when the input data size increases.

In Figures 6.7 and 6.8 we show the predicted cache misses for loops in the SWIM and TOMCATV benchmarks. Each figure presents the predicted misses and the measured misses. again. using hardware counters. Once more. we note that the prediction is very accurate for the L2 cache (average prediction error 3.11% for SWIM and 3.18% for TOMCATV). but not so accurate for the L1 cache (average prediction error 13.73% for SWIM and 18.62% for TOMCATV). This is due to the fact that we model a fully associative cache. and the relatively small L1 cache sees many conflict misses on the bigger loops. such as CALC1 do100 and CALC2 do200 in SWIM. and do60 and do100 in TOMCATV. Confirming this observation is the fact that the model predicts correctly the number of misses in the bigger L2 cache.

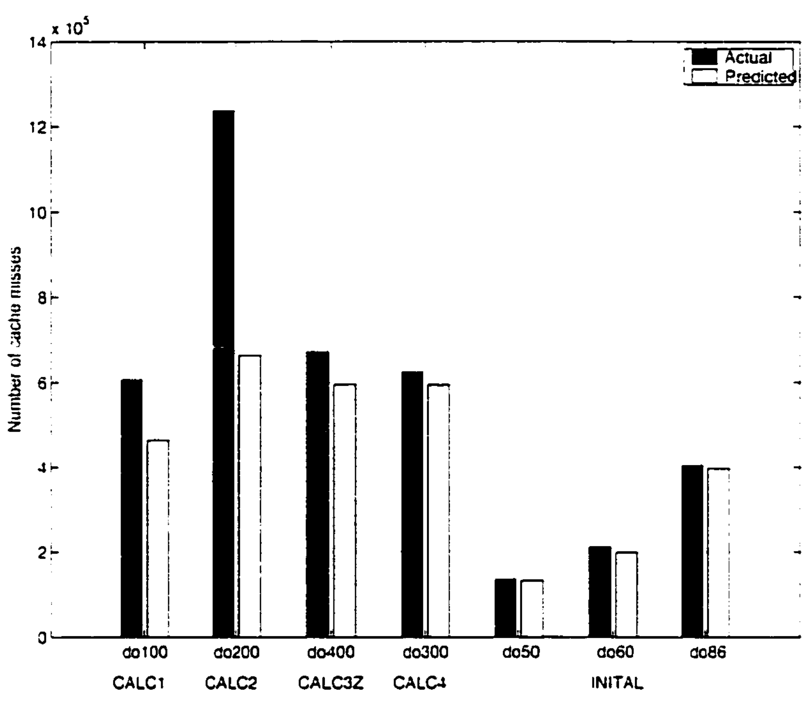
Since the other benchmarks considered have a large number of loops. we summarize the cache prediction for these benchmarks in Figure 6.9. The bar for each benchmark is obtained as follows: for each loop considered. we estimate the number of cache misses for each cache level. and we measure the actual number of cache misses using hardware counters. The misses for each loop are multiplied by the number of executions of the loop in the program. and then added to obtain the total number of predicted misses and the total number of actual misses. Then. the predicted number of misses is divided by the actual number of misses to obtain the prediction accuracy. Thus. a value of 100% represents a perfect prediction.

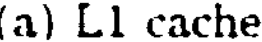
For the benchmarks in Figure 6.9 the average prediction error is 27.41% for the L1 cache. with

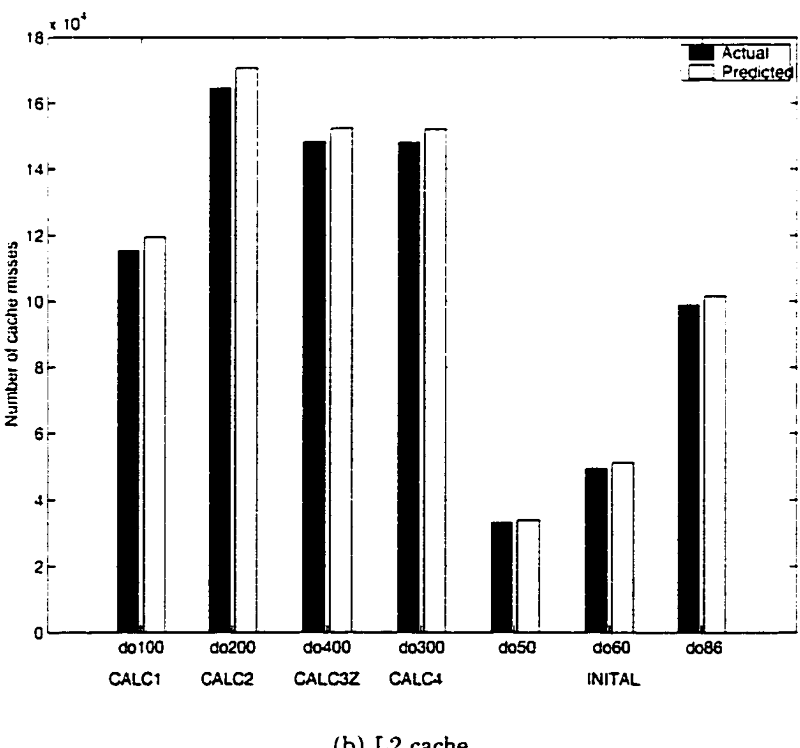
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a standard deviation of 19.62%. The L2 cache prediction has an average error of 17.10%. with a standard deviation of 29.49%. If we eliminate the APSI benchmark. for which the majority of the loops have a very small number of misses (few hundreds). therefore the prediction error is relatively large. the numbers become. 27.33% average prediction error for the L1 cache. with a standard deviation of 21.50%. and 6.13% average prediction error with 5.86% standard deviation for the L2 cache. Again. the main reason behind the relatively high prediction error for the L1 cache is that our model is for fully associative caches. and this cache is a small two-way set associative. therefore there are conflict misses that are not predicted by our model.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



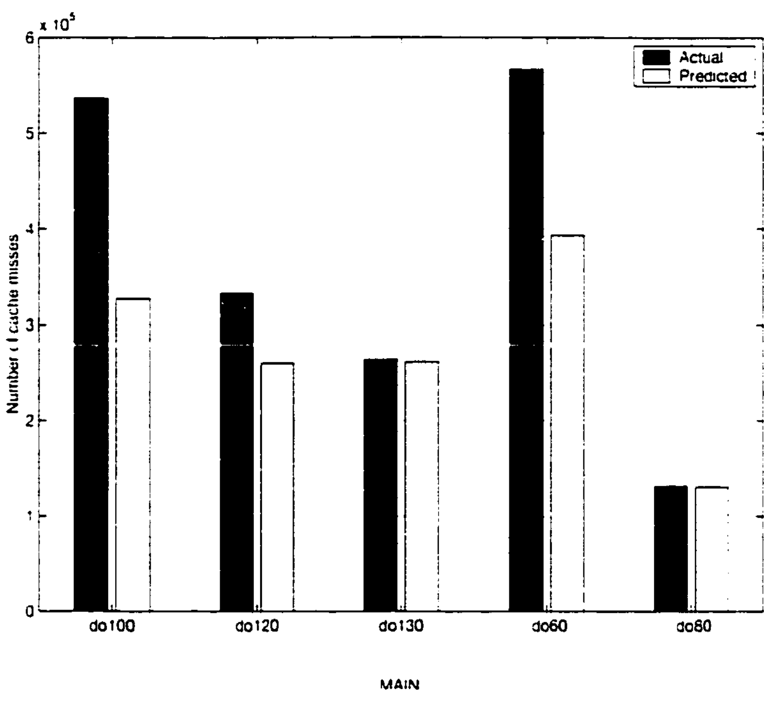

(a) L1 cache



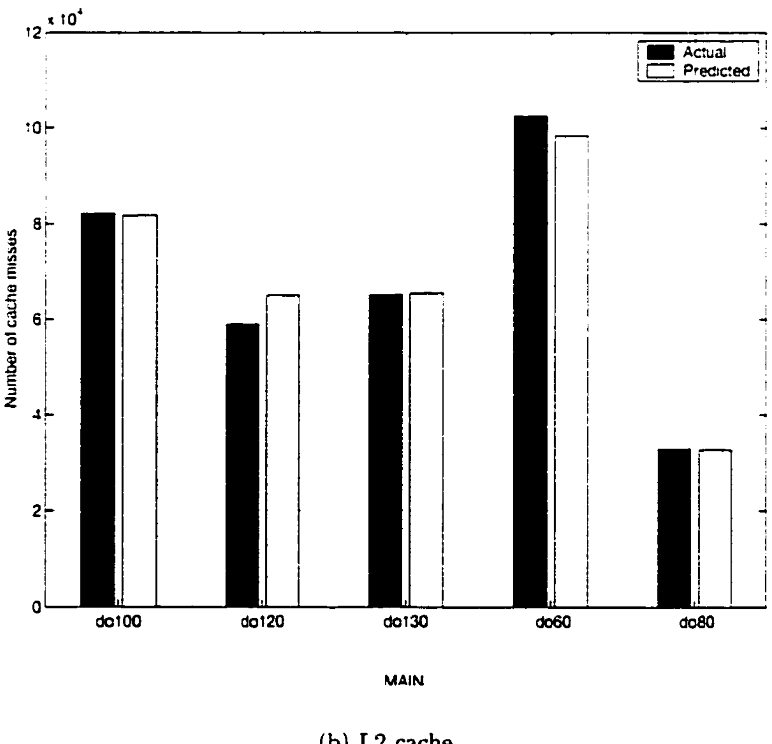
(b) L2 cache

Figure 6.7: SWIM – cache miss prediction on the R10000

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



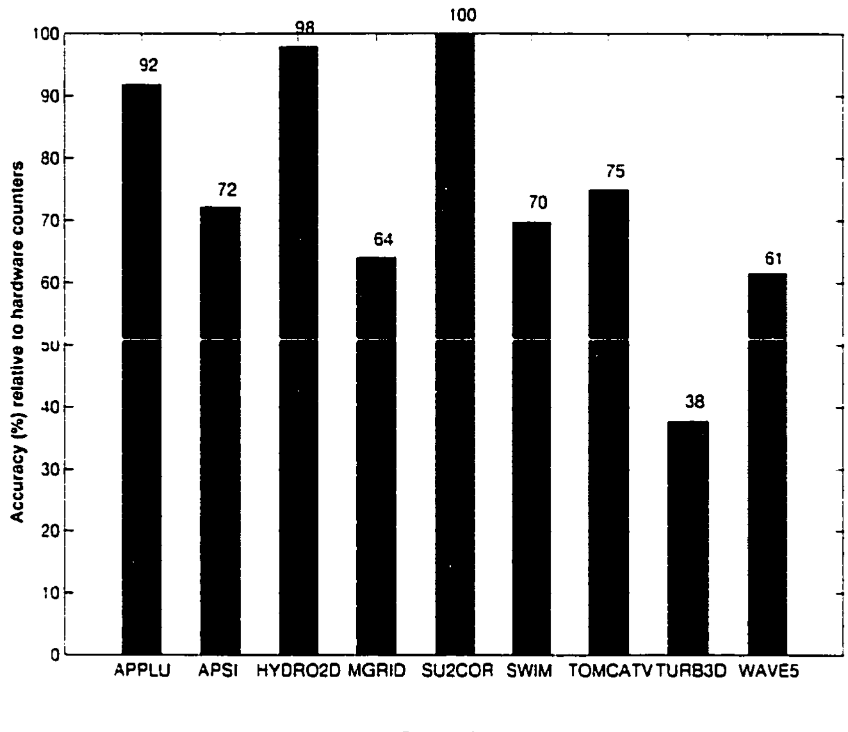
(a) L1 cache

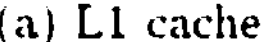


(b) L2 cache

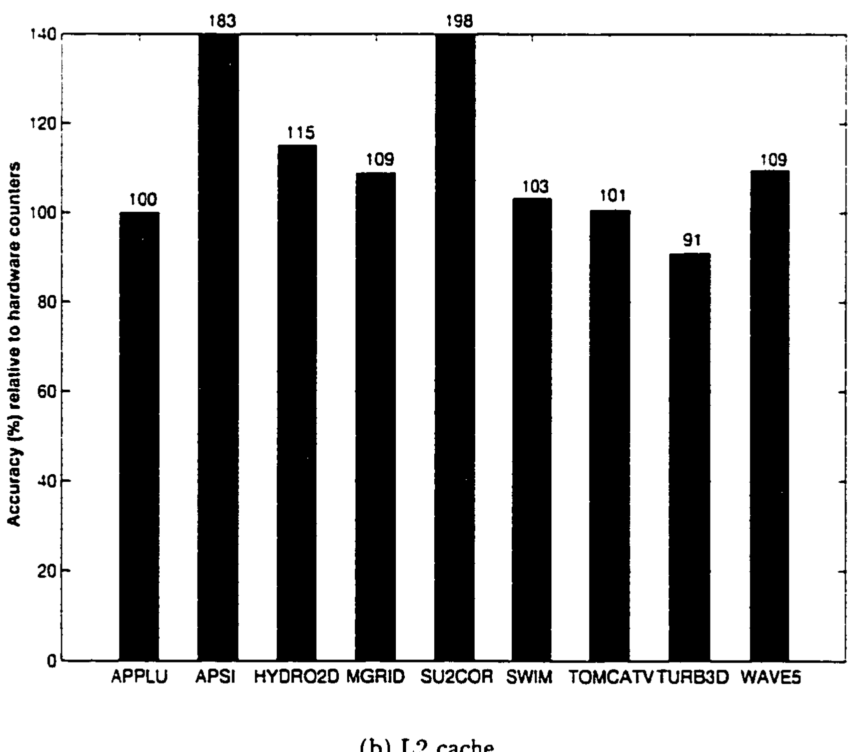
Figure 6.8: TOMCATV – cache miss prediction on the R10000

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.





(a) L1 cache



(b) L2 cache

Figure 6.9: SPECfp95 – cache miss prediction accuracy on the R10000

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

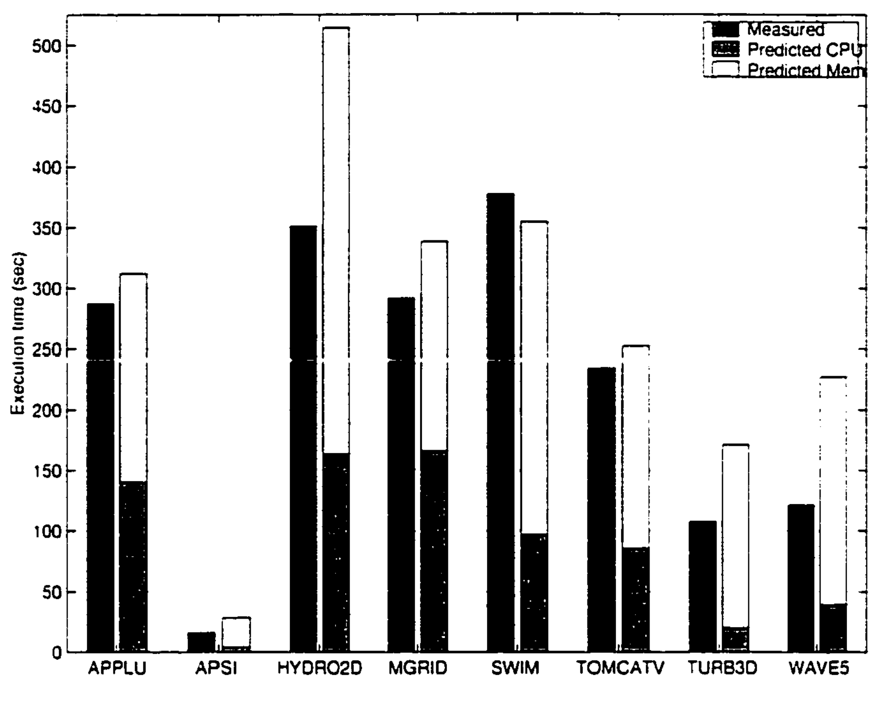
6.2.4 Execution Time Prediction with the Stack Distances Model

In this section we present execution time prediction using a combination of the Stack Distances Model and the CPU model. Since the Stack Distances Model currently handles loop nests only. to obtain the results presented in Figure 6.10 we have used the following procedure: for each loop nest we estimate the number of cache misses for both levels of cache using the Stack Distances Model. Also. for each nest we estimate the CPU execution time using the model presented in Section 3.1. The symbolic prediction expressions are then evaluated using the processor model for both the MIPS R10000 and the UltraSparc II*i*. thus obtaining a predicted execution time for each nest. We multiply the predicted execution time with the number of times that the loop is executed in the benchmark to obtain a predicted execution time for the benchmark. The right bar in each group in Figure 6.10 represents this estimation for each of the SPECfp95 benchmarks. The lower part of the bar (gray) is the CPU estmation and the upper part (white) represents the memory estimation. The left bar in each group is the measured time. Again. to obtain the execution time for the benchmark. we measured each loop nest independently and multiplied its execution time by the number of executions.

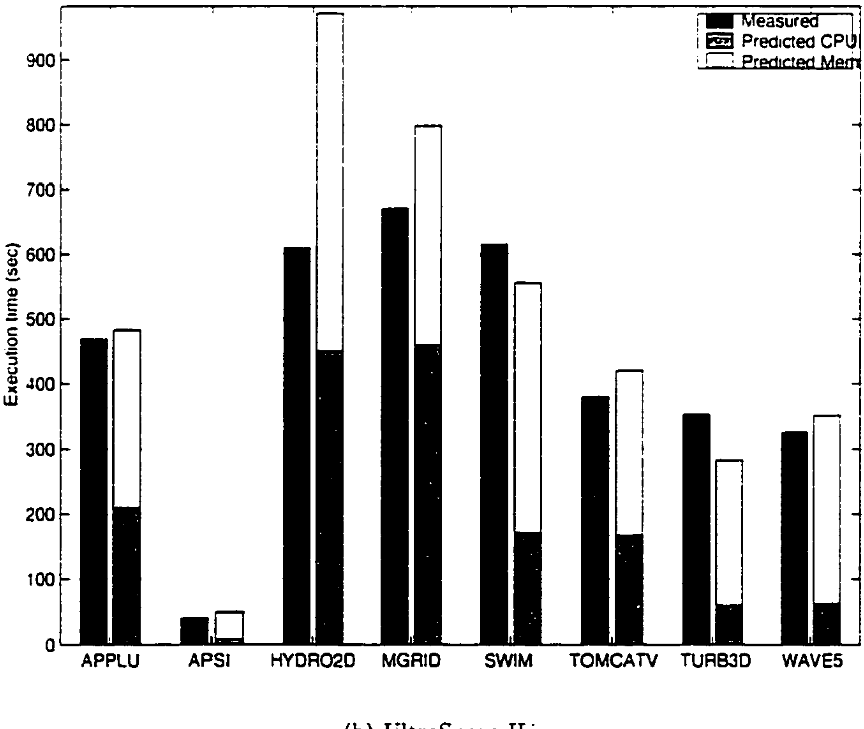
For most of the benchmarks the prediction is quite accurate. Exception makes HYDRO2D. which has a 47% prediction error for the R10000 and 67% prediction error for the UltraSparc. We suspected that the error comes from the fact that we predict the cache behavior for cold caches. and in this benchmark there might be inter-nest reuse. Therefore. we measured the performance using the run-time stack algorithm for the entire program. The prediction error dropped to 14% for the R10000 and to 55% for the UltraSparc. We are currently studying the cause of the highe prediction error on the UltraSparc.

Including HYDRO2D. the average compile-time prediction error is 34.67% (standard deviation 33.49%) for the MIPS R10000. and 18.83% (standard deviation 17.64%) for UltraSparc.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a) MIPS R10000



(b) UltraSparc II*i*

Figure 6.10: SPECfp95 – execution time prediction for selected loops in each benchmark using the -O2 optimization flag

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.3 Summary

In this chapter we have presented experimental results to validate our processor and memory hierarchy models. We have looked at scientific Fortran codes from the SPECfp95 benchmarks and SpLib. a sparse linear algebra package. The Indirect Accesses model for the memory hierarchy works quite well on the SpLib codes. with an average prediction error of less than 15%. When combined with the processor model. the average prediction error over the most significant loops in the program was below 20%. We consider these errors to be very reasonable for a static. architecture independent performance predictor.

The Stack Distances model of the memory hierarchy is even more precise. The average prediction error is about 17% for the small 2-way set-associative L1 cache of the R10000. and around 7% for the larger L2 cache. When combined with the processor model. the average prediction error for the SPECfp95 benchmarks is within 35% for the R10000 and within 20% for the UltraSparc.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 7

Conclusions and Future Work

We started the work presented in this thesis because we thought we do not have enough knowledge about the cache behavior of programs. nor precise enough methods to measure this behavior. much less to predict it. Predicting program performance is a difficult task. Predicting performance at compile-time is inherently more difficult because of all the unknowns. such as loop bounds. branch frequencies. etc.. that have to be taken into consideration. In concluding this work. we will not pretend to have completely solved this problem. However. this work can provide the necessary foundation for performance tuning. from helping a compiler to select the best sequence of optimizations. to helping the user visualize performance data and relate it back to the source code. from enabling system evaluation of not yet available hardware. to comparative system evaluation and scalability analysis.

Of course. the algorithms and tools presented in this thesis are not perfect and there is much space for improvement. The bulk of the work is the compile-time prediction model. We have shown that it is possible to predict performance. in an architecturally independent way. with reasonable accuracy. In fact. when we started this project. we wanted to predict performance within 50% of the actual execution time. It turns out that we do much better for a large fraction of the benchmark programs. including the ones that contain sparse algebra routines. Still it will be interesting to see how well the methods presented here do when they are employed to drive compiler optimization. A flavor of this potential has been already shown when our method was used to automatically map code for intelligent memory architectures [64]. We would like to see how the method can improve compiler optimizations.

Apart from its uses. there are several areas in which the prediction model can be improved.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

especially if we want to use the same model for predicting parallel program performance. A better superscalar processor model that takes into consideration operations dependences can be developed to improve the estimation of small loops. One could even consider using register allocation and instruction scheduling hints, although that would restrict the generality of the model and its compiler and machine independence.

Many other improvements can be performed on the cache model. Currently, the Stack Distances model is applied to a loop if *all* the data dependences in the loop have known data dependence distance vectors. We could relax this restriction and use a combination of the Indirect Accesses model and the Stack Distances model, by estimating the foot-print using the Indirect Accesses model for the references that do not have distance vectors computed, and substituting this estimation for the accessed array section in the Stack Distances model. We will lose some of the accuracy given by the Stack Distances model, but we will be able to analyze more loops. The combined model will become even more important if we want to apply the Stack Distances model to estimate the number of cache misses across loop nests. As McKinley and Temam have shown in [46, 47], inter-loop misses constitute an important fraction of the total cache misses in the SPEC and Perfect Benchmarks. Another limitation of the Stack Distances model is due to the fact that it estimates fully-associative caches. Since there are no fully-associative caches implemented in real hardware, it would be interesting to explore the possibility of adapting the algorithm to model set-associative caches. The stack algorithm has already been used to model set-associative caches, therefore is just a matter of finding an appropriate representation for the set-associativity inside the compiler.

Other extensions to be considered are multiprocessor extensions. In fact, if the distribution of the array onto processors is known at compile time, the compile-time algorithm presented in this thesis can be easily extended to multiprocessors by intersecting the array section spanned by a dependence with the array section mapped to the local memory of the processor. All the accesses inside the array section mapped to the processor are local accesses, while the array elements accessed outside the intersection are remote accesses. Using the array sections one can also compute the false sharing, which is considered to be one of the factors making the caches in multiprocessors less effective than in uniprocessors.

Another direction in which the Stack Distances model could be extended is integer codes and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

object-oriented programs. In this type of codes the bulk of the computation is no longer spent in loops accessing arrays. so a different paradigm has to be used. However. the Stack Distances model is not restricted to arrays.

To conclude. we have shown that the stack processing algorithms are a very powerful technique. We used stack distances to quantify locality and we have designed and implemented a compile-time algorithm that computes the stack histogram at compile-time. We have used the stack histogram to predict program performance statically with very good accuracy. The most interesting feature of our stack algorithm is that once the histogram is computed. the number of cache misses can be estimated for any cache size. We do not know of any other method that does not require the complete set of cache parameters to estimate misses. We have also presented a new algorithm for stack processing. that is 30% faster than the best know algorithm on the suite of programs traced.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

References

[1] V. S. Adve. J. Mellor-Crummey. M. Anderson. J.-C. Wang. D. A. Reed. and K. Kennedy. An integrated compilation and performance analysis environment for data parallel programs. In *Proceedings of the 1995 conference on Supercomputing*. San-Diego. CA. December 1995.

[2] R. A. Aydt. *The Pablo Self-Defining Data Format.*. 1992. ftp://vibes.cs.uiuc.edu-/pub/Pablo.Release.5/SDDF/Documentation/SDDF.ps.gz.

[3] V. Balasundaram. G. Fox. K. Kennedy. and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. pages 213–223. Williamsburg. VA. April 1991.

[4] T. Ball and J. R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI '93)*. pages 300–313. 1993.

[5] U. Banerjee. *Dependence analysis.* Kluwer Academic Publishers. 1997.

[6] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal for Research and Development.* pages 353–357. July 1975.

[7] M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications.* 3(3):5–40. 1989.

[8] W. Blume. R. Doallo. R. Eigenmann. J. Grout. J. Hoeflinger. T. Lawrence. J. Lee. D. Padua. Y. Paek. W. Pottenger. L. Rauchwerger. and P. Tu. Parallel Programming with Polaris. *IEEE Computer.* December 1996.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[9] R. Bramley and X. Wang. SPLIB: A library of iterative methods for sparse linear systems. Technical Report TR454. Indiana University. February 1996.

[10] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*. pages 162–175. July 1986.

[11] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Parallel and Distributed Computing*. 5:517–550. 1988.

[12] H. Casanova. M. G. Thomason. and J. J. Dongarra. Stochastic performance prediction for iterative algorithms in distributed environments. *Journal of Parallel and Distributed Computing*. 58:68–91. 1999.

[13] C. Cascaval. *Estimating Execution Time and Cache Misses using Delphi*. Dept. of Computer Science. Univ. of Illinois. 2000.

[14] C. Cascaval. L. DeRose. D. A. Padua. and D. A. Reed. Compile-time based performance prediction. In J. Ferrante and L. Carter. editors. *Lecture Notes in Computer Science*. Springer Verlag. 1999.

[15] P. Clauss. The volume of a lattice polyhedron to enumerate processors and parallelism. Technical Report 95-11. ICPS. 1995. http://icps.u-strasbg.fr/pub-95/pub-95-11.ps.gz.

[16] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *Proceedings of the 10th ACM International Conference on Supercomputing (ICS'96)*. May 1996. Also available as ICPS Technical Report 96-03.

[17] S. Coleman and K. S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI '95)*. La Jolla. CA. June 1995. SIGPLAN.

[18] T. H. Cormen. C. E. Leiserson. and R. L. Rivest. *Introduction to Algorithms*. MIT Press. 1990.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[19] B. Creusillet and F. Irigoin. Interprocedural array region analyses. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*. pages 46–60. 1995.

[20] B. Creusillet and F. Irigoin. Exact versus approximate array region analyses. In *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*. pages 86–100. 1996.

[21] L. DeRose. Y. Zhang. and D. A. Reed. SvPablo: A multi-language performance analysis system. In *10th International Conference on Computer Performance Evaluation - Modelling Techniques and Tools - Performance Tools '98*. pages 352–355. Palma de Mallorca. Spain. September 1998.

[22] T. Fahringer. Evaluation of benchmark performance estimation for parallel Fortran programs on massively parallel SIMD and MIMD computers. In *IEEE Proceedings of the 2nd Euromicro Workshop on Parallel and Distributed Processing*. Malaga. Spain. January 1994.

[23] T. Fahringer. Estimating and optimizing performance for parallel programs. *IEEE Computer*. 28(11):47–56. November 1995.

[24] T. Fahringer. *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic Press. 1996.

[25] T. Fahringer. Estimating cache performance for sequential and data parallel programs. Technical Report TR 97-9. Institute for Software Technology and Parallel Systems. Univ. of Vienna. Vienna. Austria. October 1997.

[26] K. A. Faigin. J. P. Hoeflinger. D. A. Padua. P. M. Petersen. and S. A. Weatherford. The Polaris internal representation. *International Journal of Parallel Programming*. 22(5):553–586. October 1994. Also available as CSRD Technical Report 1317.

[27] J. Ferrante. V. Sarkar. and W. Thrash. On Estimating and Enhancing Cache Effectiveness. In *4th International Workshop on Languages and Compilers for Parallel Computing*. August 1991.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[28] D. Gannon. W. Jalby. and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. *Journal of Parallel and Distributed Computing*. 5(5):587–616. October 1988.

[29] J. D. Gee. M. D. Hill. and A. J. Smith. Cache performance of the SPEC92 benchmark suite. In *Proceedings of the IEEE Micro*. pages 17–27. August 1993.

[30] S. Ghosh. M. Martonosi. and S. Malik. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*. San Jose. CA. October 1998.

[31] M. R. Haghighat. *Symbolic Analysis for Parallelizing Compilers*. PhD thesis. University of Illinois at Urbana-Champaign. 1994.

[32] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*. 18(4):477–518. 1996.

[33] M. Hall et al. Mapping irregular applications to DIVA and a PIM-based data-intensive architecture. In *Proceedings of International Conference on High Performance Computing and Communication (SC'99)*. Portland. Oregon. November 1999.

[34] P. Havlak. *Interprocedural Symbolic Analysis*. PhD thesis. Rice University. May 1994.

[35] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*. 38(12):1612–1630. December 1989.

[36] J. P. Hoeflinger. *Interprocedural Parallelization Using Memory Classification Analysis*. PhD thesis. University of Illinois at Urbana-Champaign. August 1998.

[37] Y. Kang. M. Huang. S.-M. Yoo. Z. Ge. D. Keen. V. Lam. P. Pattnaik. and J. Torrellas. FlexRAM: Toward an advanced intelligent memory system. In *International Conference on Computer Design (ICCD)*. October 1999.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[38] T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff. Iterative compilation in program optimization. In *8th Int'l Workshop on Compilers for Parallel Computers (CPC)*, pages 35–44, 2000.

[39] D. E. Knuth. *The Art of Computer Programming*, volume 3 (Sorting and Searching). Addison Wesley Longman, 2nd edition, 1998.

[40] Z. Li and P.-C. Yew. Efficient interprocedural analysis for program parallelization and restructuring. In *Proceedings of the SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, July 1988.

[41] Z. Li, P.-C. Yew, and C.-Q. Zhu. Data dependence analysis on multii-dimensional array references. In *Proceedings of the third international conference on Supercomputing*, pages 215–224, Crete, Greece, 1989.

[42] D. Lilja, D. Marcovitz, and P. C. Yew. Memory Referencing Behavior and Cache Performance in a Shared Memory Multiprocessor. Technical Report 836, CSRD, 1988.

[43] R. L. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2), 1970.

[44] K. S. McKinley. *Automatic and Interactive Parallelization*. PhD thesis, Rice University, March 1994.

[45] K. S. McKinley. A compiler optimization algorithm for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 9(8):769–787, August 1998.

[46] K. S. McKinley and O. Temam. A Quantitative Analysis of Loop Nest Locality. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, 1996.

[47] K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the Perfect benchmarks. *ACM Transactions on Computer Systems*. To appear.

[48] C. L. Mendes. *Performance Scalability Prediction on Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, May 1997.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[49] M. Oskin, F. Chong, and T. Sherwood. Active Pages: A computation model for intelligent memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA '98)*, pages 192–203, June 1998.

[50] P. M. Petersen and D. A. Padua. Experimental evaluation of some data dependence tests. Technical Report 1080, CSRD, 1991.

[51] V. Phalke and B. Gopinath. An inter-reference gap model for temporal locality in program behavior. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 291–300, May 1995.

[52] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, May 1989.

[53] W. Pugh. A practical algorithm for exact array dependence analysis. In *Communications of the ACM*, volume 35, pages 102–114, August 1992.

[54] W. Pugh. Counting solutions to Presburger formulas: How and why. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI '94)*, volume 29, pages 121–134, June 1994.

[55] C. Pyo, G. Lee, K.-W. Lee, and H.-K. Han. Reference distance as a metric for data. In *Proceedings of the Conference on High Performance Computing on the Information Superhighway, HPC Asia*, pages 151–156, 1997.

[56] *MIPS R10000 Microprocessor User's Manual*, 2.0 edition, January 1997.

[57] D. A. Reed, D. A. Padua, I. T. Foster, D. B. Gannon, and B. P. Miller. Delphi: An integrated, language-directed performance prediction, measurement, and analysis environment. In *Frontiers '99: The 9th Symposium on the Frontiers of Massively Parallel Computation*, Annapolis, MD, February 1999.

[58] R. Saavedra and A. Smith. Measuring cache and TLB performance and their effect on benchmark run times. *IEEE Transactions on Computers*, 44(10):1223–1235, October 1995.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[59] R. H. Saavedra-Barrera and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. Technical Report CSD 92-715. Computer Science Division. UC Berkeley. 1992.

[60] R. H. Saavedra-Barrera. A. J. Smith. and E. Miya. Machine characterization based on an abstract high-level language machine. *IEEE Transactions on Computers*. 38(12):1659–1679. December 1989.

[61] V. Sarkar. Determining average program execution times and their variance. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI '89)*. pages 298–312. Portland. Oregon. July 1989.

[62] Z. Shen. Z. Li. and P.-C. Yew. An empirical study of Fortran programs for parallelizing compilers. Technical Report 983. CSRD. 1990.

[63] A. J. Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Transactions on Software Engineering*. SE-4(2):121–130. March 1978.

[64] Y. Solihin. J. Lee. and J. Torrellas. Automatically mapping code in an intelligent memory architecture. Submitted for publication.

[65] R. A. Sugumar and S. G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Trans. Comp. Sys.*. 13(1). 1995.

[66] Sun Microsystems. *UltraSparc-IIi User's Manual*. 1997.

[67] J. G. Thompson and A. J. Smith. Efficient stack algorithms for analysis of write-back and sector memories. *ACM Trans. Comp. Sys.*. 7(1):78–117. February 1989.

[68] X. Vera. J. Llosa. A. Gonzales. and C. Ciuraneta. A fast implementation of Cache Miss Equations. In *8th International Workshop on Compilers for Parallel Computers (CPC 2000)*. pages 319–325. Aussois. France. January 2000.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[69] K.-Y. Wang. Precise compile-time performance prediction for superscalar-based computers. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI '94)*. pages 73–84. 1994.

[70] W.-H. Wang and J.-L. Baer. Efficient trace-driven simulation methods for cache performance analysis. *ACM Transactions on Computer Systems*. 9(3). 1991.

[71] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98*. 1998.

[72] M. E. Wolf. *Improving locality and parallelism in nested loops*. PhD thesis. Stanford University. August 1992.

[73] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI '91)*. June 1991.

[74] M. E. Wolf. D. E. Maydan. and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture*. pages 274–286. Paris. France. December 1996.

[75] M. Wolfe. More Iteration Space Tiling. In *Proceedings of Supercomputing '89*. pages 655–664. Reno. NV. November 1989. ACM.

[76] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wessley. 1996.

[77] Z. Xu. J. R. Larus. and B. P. Miller. Shared-memory performance profiling. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Las Vegas. NV. June 1997.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Curriculum Vitae

Gheorghe Călin Caşcaval was born in Cluj-Napoca, România. After graduating from the Technical University in his hometown with a Master of Science degree in Computer Engineering, he worked for two years for the Institude for Design in Automation, participating in the software design and implementation of digital circuit testers. In 1993 he was accepted as a student at West Virginia University, Morgantown, WV, where he completed a Master of Science degree in Computer Science. Following his employement at CyberMarché Inc., where he designed software for project management and engineering knowledge bases, he started his PhD at University of Illinois at Urbana-Champaign in 1996, under the guidance of Prof. David Padua.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.