



136  
274  
THS

2002

This is to certify that the

thesis entitled

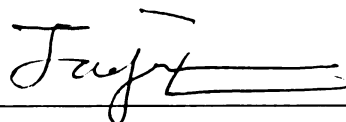
**ADAPTIVELY INCREASING PERFORMANCE AND SCALABILITY OF  
AUTOMATICALLY PARALLELIZED PROGRAMS**

presented by

**H. D. K. MOONESINGHE**

has been accepted towards fulfillment  
of the requirements for

      M.S.       degree in Computer Science



Major professor

Date 7/23/02

**LIBRARY**  
**Michigan State**  
**University**

**PLACE IN RETURN BOX** to remove this checkout from your record.  
**TO AVOID FINES** return on or before date due.  
**MAY BE RECALLED** with earlier due date if requested.

DATE DUE	DATE DUE	DATE DUE
<div style="text-align: center;"> <div>511403</div> <div>APR 31 2004</div> </div>		

ADAPTIVELY INCREASING PERFORMANCE AND SCALABILITY OF  
AUTOMATICALLY PARALLELIZED PROGRAMS

By

H. D. K. MOONESINGHE

A THESIS

Submitted to  
Michigan State University  
in partial fulfillment of the requirements  
for the degree of

MASTER OF SCIENCE

Department of Computer Science and Engineering

2002

## **ABSTRACT**

### **ADAPTIVELY INCREASING PERFORMANCE AND SCALABILITY OF AUTOMATICALLY PARALLELIZED PROGRAMS**

By

H. D. K. MOONESINGHE

Compilers use various transformation techniques to obtain a higher performance in programs. But the optimal implementation depends not only on the information available at compile-time but also on the information available at run-time, where complete knowledge of the execution environment exists. In this thesis, we present adaptive execution techniques to increase the performance and scalability of automatically parallelized loops by executing them parallelly or serially using the information available at both compile-time and run-time. We have implemented several adaptation and performance estimation algorithms in our compiler preprocessor. The preprocessor inserts code that determines at compile-time or at run-time the way the loops are executed. Using a set of standard numerical applications written in Fortran77 and running them with our techniques on a 32-processor distributed shared memory machine (SGI Origin2000), we obtain the performance of our techniques, on average, 30%, 25%, 20%, and 14% faster than the original parallel programs on 32, 16, 8 and 4 processors respectively. Some programs run more than twice faster than the corresponding original parallel programs. Also, our approach increases the scalability of parallel applications.

*To My Parents*

## ACKNOWLEDGEMENTS

This thesis would not have happened without the help of many people whom I wish to thank here.

I express deep gratitude to my advisor Dr. Jaejin Lee, for allowing me to pursue this research and helping me along every step of the way. Without his generous support and constant guidance, this thesis would simply not be possible. I appreciate the confidence and encouragement extended by him, towards my work.

I sincerely thank Dr. Lionel M. Ni, Dr. Sandeep Kulkarni, and all other professors of the Department of Computer Science & Engineering at Michigan State University for providing me with a sound foundation on all aspects of computer science, which helped me immensely in achieving this goal.

I would also like to thank all the staff members of CSE for their assistance's. Although I cannot name them all here, I appreciate the help they provided to me in many ways throughout the graduate school. Special thanks to Debbie and Linda for providing information and advises about the administrative tasks.

A special word of thanks goes to my friend Niroshena for his support and encouragement's given to me throughout this period. I would also like to thank all other friends for the support given to me during this period.

These acknowledgements wouldn't be complete without an expression of gratitude to my parents, and teachers throughout the years. They have always been a source of support and encouragement.

# TABLE OF CONTENTS

<b>LIST OF TABLES</b>	<b>vi</b>
<b>LIST OF FIGURES</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Related Work . . . . .	3
1.3 Organization of the Thesis . . . . .	4
<b>2 Execution Model and Our Framework</b>	<b>5</b>
2.1 Parallel Execution Model . . . . .	5
2.2 An Overview of the Framework . . . . .	6
<b>3 Adaptive Execution Algorithms</b>	<b>8</b>
3.1 Compile-time Cost Estimation . . . . .	9
3.2 Run-time Cost Estimation . . . . .	11
3.3 Adaptive Execution Algorithms . . . . .	13
3.3.1 First Two invocations with Timing (F2T) . . . . .	13
3.3.2 Most Recent with Timing (MRT) . . . . .	14
3.3.3 Static cost estimation and Most Recent with Timing (SMRT)	15
3.3.4 MRT with Run-time cost estimation (MRTR) . . . . .	15
3.3.5 MRT with Static and Run-time cost estimation (SMRTR) . . .	15
<b>4 Evaluation Environment</b>	<b>17</b>
4.1 Compiler . . . . .	17
4.2 Applications . . . . .	18
4.3 Target Architecture . . . . .	18
<b>5 Evaluation</b>	<b>20</b>
5.1 Characteristics of the Parallel Loops . . . . .	20
5.2 Results . . . . .	24
<b>6 Conclusions</b>	<b>34</b>
<b>APPENDIX</b>	<b>36</b>
<b>BIBLIOGRAPHY</b>	<b>40</b>



## LIST OF TABLES

4.1	Auto-Parallelizing directives used for SGI MIPSpro Fortran77 compiler. .	18
4.2	Applications used for the evaluation. . . . .	19
4.3	Machine specification. . . . .	19
5.1	Characteristics of parallel loops in the applications. . . . .	22
5.2	Characteristics of parallel loops whose amount of work can be estimated by the compile-time cost estimation model. . . . .	25
5.3	Speedup of Base and SMRTR strategies of each application. . . . .	27
1	Execution time of the applications under each strategy. . . . .	37
2	Execution time of the applications in APO mode. . . . .	39

## LIST OF FIGURES

2.1	Parallel execution model. . . . .	6
2.2	Adaptation framework. . . . .	7
3.1	The adaptation scheme. . . . .	9
3.2	Determining the threshold value with compile-time cost estimation .	11
3.3	Determining the threshold value with run-time cost estimation. . . .	13
5.1	Normalized execution time of Applu . . . . .	28
5.2	Normalized execution time of Cg . . . . .	28
5.3	Normalized execution time of Hydro2d . . . . .	29
5.4	Normalized execution time of Lu . . . . .	29
5.5	Normalized execution time of Mg . . . . .	30
5.6	Normalized execution time of Mgrid . . . . .	30
5.7	Normalized execution time of Sp . . . . .	31
5.8	Normalized execution time of Su2cor . . . . .	31
5.9	Normalized execution time of Swim . . . . .	32
5.10	Average normalized execution time of all the applications . . . . .	32
5.11	Speedup of Base and SMRTR for each application. . . . .	33
5.12	SMRTR execution time normalized to APO. . . . .	33

# Chapter 1

## Introduction

### 1.1 Background

High performance optimizing and parallelizing compilers perform various optimizations to improve the performance of sequential and parallel programs [2]. A problem that most of such compilers currently faced is the lack of information about machine parameters and input data at compile-time. The performance of an algorithm that best solves a problem largely depends on the combination of the input and hardware platforms used to execute it. This information is difficult to obtain or unavailable at compile time. Thus, it is difficult or sometimes impossible to statically fine tune the application to a particular execution platform. In order to address this issue, recent work [1, 8, 17, 20, 21] has been started to explore the feasibility of run-time fine-tuning and optimizations when complete knowledge about the input data set and the hardware platform is available.

Automatic parallelizing compilers analyze and transform a sequential program into a parallel program without any intervention of the user. However, in order to achieve maximum performance from the automatically parallelized programs, the user must consider the following with regards to the underlying multiprocessor architec-

ture: amount of parallelism contained in the program, cache locality of the program, hardware cache coherence mechanism, workload distribution among processors, data distribution, false sharing, coordination overhead incurred between processors, synchronization overhead, etc. Since these factors manifest synergistically on the performance of parallel loops and the effects differ from one machine to another, identifying performance bottlenecks of a parallel program is a tedious and difficult job for the programmer. Thus, the cost that a programmer pays in order to obtain a reasonable performance from an automatically parallelized program adds extra difficulties in developing and maintaining parallel programs.

Here in our work, instead of identifying the performance bottlenecks manually from an automatically parallelized program, we avoid executing some parallel loops in parallel if performance degradation of the loop exceeds a predefined threshold value during parallel execution of the program. No knowledge of the program should be assumed from the programmer.

In this thesis, we present adaptive execution techniques that increase the performance and scalability of automatically parallelized programs. The adaptation and performance estimation algorithms are implemented in a compiler preprocessor. The preprocessor inserts code that automatically determines at compile-time or at run-time the way the parallelized loops are executed.

Using a set of standard numerical applications written in Fortran77 and running them with our techniques on a 32-processor distributed shared memory machine (SGI Origin2000), we obtain the performance, on average, 30%, 25%, 20%, and 14% faster than the original parallel programs on 32, 16, 8, and 4 processors respectively. Even, some programs run more than twice faster than the original parallel version with our scheme.

## 1.2 Related Work

Recent work has begun to explore the possibilities of optimizations performed at run-time when complete knowledge of the execution environment exists. Many different types of adaptive optimization techniques have been proposed recently in the literature.

Multiple versioning of a loop for run-time optimization was first proposed by Byler *et al.* [4], and modern compilers still use this technique. Diniz *et al.* [8] used multiple versioning with dynamic feedback to automatically choose the best synchronization optimization policy for object-based parallel programs. The main problem of multiple versioning is that it has a significant code growth, since each version is a replication of a single code section with a particular optimization. In our work, we generate at most two versions of a parallel loop. Moreover, versioning of the parallel loop can be avoided using conditional parallelization directives.

Some approaches [10, 19, 7, 18] are based on parameterization of the code at compile-time to restructure it at run-time. Gupta and Bodik [10] dealt with the complexity of loop transformations, such as loop fusion, loop fission, loop interchange, and loop reversal, done at run-time. Saavedra *et al.* [19] proposed adaptive prefetching algorithm that can change the prefetching distance of individual prefetching instructions. Their adaptive algorithm uses simple performance data collected from hardware monitors at run-time. Another adaptive optimization technique, which is based on replication of objects in object oriented programs, is proposed by Rinard *et al.* [17]. In order to avoid synchronization overhead occurred in updating a shared object, they replicate the object adaptively at run-time. The replication policy adapts to the dynamic characteristic of each program execution.

Holzle *et al.* [11] proposed a dynamic type feedback technique for improving the performance of object oriented programs. These approaches are similar to our work in that the program dynamically adapts to the environment during its execution

using run-time information. However, we neither restructure the code at run-time nor deal with object-oriented programs. We focus on shared memory parallel programs with loop level parallelism, and use different adaptation strategies in order to improve performance and scalability.

A generic compiler-support framework called ADAPT was proposed by Voss and Eigenman [20, 21] for adaptive program optimization. Users can specify types of optimizations and heuristics for applying the optimizations at run-time using ADAPT language. The ADAPT compiler generates a complete run-time system by reading these heuristics and applying them on to the target application. Lee [14] proposed a serialization technique of small parallel loops using static performance prediction and some heuristics.

A sophisticated static performance estimation model based on the stack distance [16] was proposed by Cascaval *et al.* [5]. Even though we used a fairly simple static performance estimation model here in our work, our run-time cost estimation models compensate for the inaccuracy caused by the static model. Our work is also related to adaptive compilers for heterogeneous processing in memory systems [15], where heterogeneity of the system is exploited adaptively.

## 1.3 Organization of the Thesis

The rest of the thesis is organized as follows: Chapter 2 describes the parallel execution model and an overview of our framework; Chapter 3 presents adaptive execution algorithms including the compile-time and run-time cost estimation techniques; Chapter 4 describes the environment used to test our algorithms, such as the compiler, benchmarks and the execution platform; Chapter 5 presents the results obtained by evaluating the algorithms on the target architecture. Chapter 6 concludes the thesis.

# Chapter 2

## Execution Model and Our Framework

### 2.1 Parallel Execution Model

We use the OpenMP parallel programming model as our model of parallel execution. It is a master-slave thread model [3, 6, 12]. When a thread encounters a parallel region, it creates a team of threads with an execution context for each thread, and it becomes the master of the team. A parallel region is a block of code, where in our case a parallel loop, which is executed by multiple threads in parallel.

Once a parallel loop is executed, the master thread creates slave threads and divides the iterations of the loop among the slaves as shown by Figure 2.1. Each thread performs a chunk of the iterations concurrently. There is an implicit barrier at the end of each parallel region. If one thread finishes its portion of the iterations, it has to wait until other threads finish executing their portion of the iterations. After all the threads finished their portion of iterations the barrier condition is complete and all the threads are released from the barrier. At this point of execution of the program, slave threads get disappeared and master thread resumes execution of the

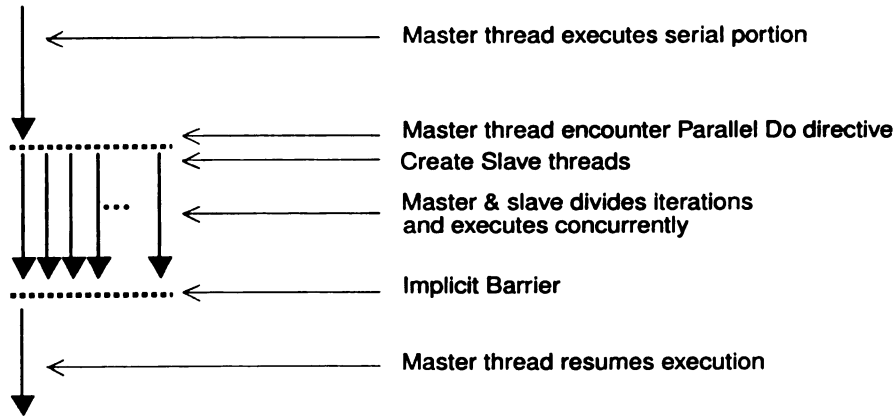


Figure 2.1: Parallel execution model.

code after the parallel loop [6, 3, 12].

Creating slave threads for a parallel loop, distributing the iterations of the loop among threads, cache affinity of each thread executing different iterations, and the synchronization between threads at the end of the loop incur an overhead for running the loop in parallel. We call it *parallel loop overhead*.

## 2.2 An Overview of the Framework

Our framework is shown by Figure 2.2. An automatically parallelized program or manually parallelized program is fed into the compiler preprocessor. The preprocessor inserts performance estimation and adaptive execution code in to the parallel program. It restructures these codes using the target machine specific parameters that are also fed into it. The output program from the preprocessor is compiled by a compiler that generates code for the target multiprocessor.

We are particularly interested in adaptive optimization strategies that select code at run-time from a set of statically generated code variants based on the execution environment. We generate two versions of the same parallel loop, one is a sequential version and the other is a parallel version. Since we are interested in running a parallel loop sequentially or parallelly, we focus on the parallel programs that contain large



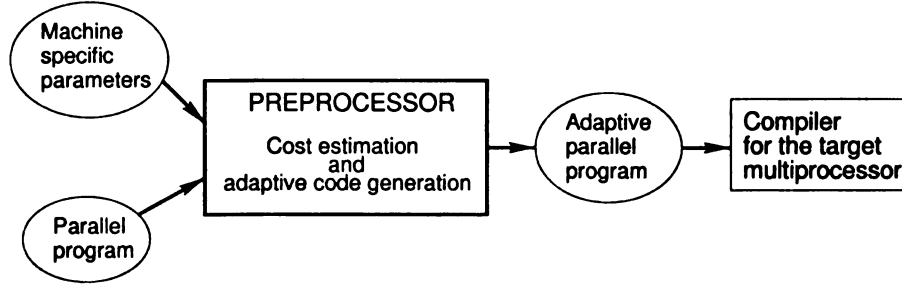


Figure 2.2: Adaptation framework.

amount of loop level parallelism. A variety of selection algorithms are compared and evaluated with these highly parallel programs: compile-time cost estimation, run-time cost estimation, selection based on execution time, selection based on performance counters, such as the number of graduated instructions, and combinations of those strategies. The key observation in this work is that most of the parallel loops are invoked many times so that adaptive execution of the parallel loops is feasible and effective.

The preprocessor inserts instrumentation code in to the parallel loop in order to measure, at run-time, the execution time and the number of instructions graduated in a single invocation. These invocations which measure execution time, the number of graduated instructions or both are called *decision runs*. Based on the measurements in the decision runs of the parallel program, the adaptation code determines the way of executing the loop, i.e., sequentially or parallelly in the next or remaining invocations.

A drawback of our adaptive execution techniques with decision runs is that we have to run a parallel loop at least once sequentially and at least once in parallel in order to obtain some useful information for adapting the loop to the run-time environment. To compensate this penalty, we add compile-time cost estimation model to the adaptation model. Also, we introduce the notion of *adaptation window* for the adaptive execution techniques.

# Chapter 3

## Adaptive Execution Algorithms

We have implemented compile-time and run-time algorithms that adaptively execute automatically parallelized programs. The same techniques can be applied to the parallel programs generated by hand as long as they use standard parallelization directives such as the ones used in automatic parallelizing compilers.

Our adaptation scheme has basically three different parts. A compile-time cost estimation model, a run-time cost estimation model, and adaptation strategies using the two models.

First, the compile-time cost estimation model (Figure 3.1) filters parallel loops that contain smaller amount of work than the parallel loop overhead (i.e. small loops). Second, the run-time cost estimation model filters highly efficient parallel loops due to large amount of work in the loop. An *efficient parallel loop* is the parallel loop whose speedup is greater than 1. Otherwise, it is an *inefficient parallel loop*. The model counts the number of instructions executed in an invocation of the loop at run-time. Also, it identifies small loops that cannot be handled by the compile-time cost estimation model due to run-time parameters in the loop. Finally, several adaptive execution strategies (including execution time based strategies) determine the way the remaining loops are executed.

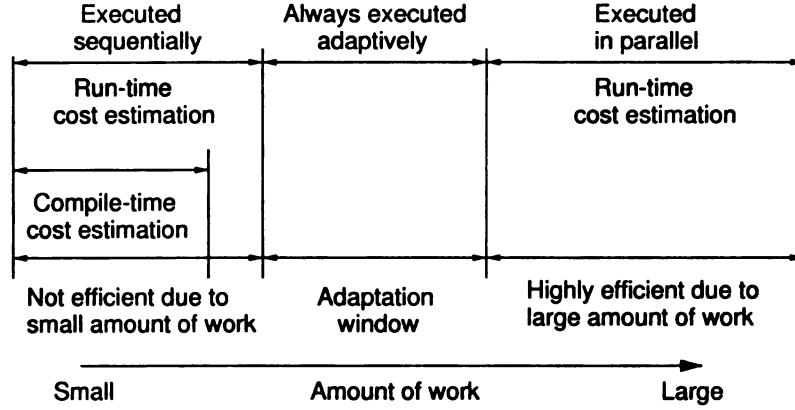


Figure 3.1: The adaptation scheme.

### 3.1 Compile-time Cost Estimation

The compile-time cost estimation model identifies parallel loops that are not efficient due to insufficient amount of work in the loop. It is not beneficial to run this type of loops in parallel because the amount of work in the loop is fairly small compared to the parallel loop overhead. We define a threshold value for the amount of work contained in the loop. If the amount of work is smaller than the threshold value, we run the loop sequentially.

We use a fairly simple cost estimation model. The amount of work ( $W$ ) in a loop can be estimated as a function of the number of iterations of the loop ( $n_i$ ), the number of assignments ( $n_a$ ), the number of floating point operations for addition ( $n_{f_{ADD}}$ ), multiplication ( $n_{f_{MUL}}$ ), subtraction ( $n_{f_{SUB}}$ ), division ( $n_{f_{DIV}}$ ), the number of intrinsic function and system procedure calls ( $n_f$  and  $n_p$ ), and the number of user defined function and procedure calls ( $n_u$ ). Consequently, the estimated amount of work ( $W_i$ ) in an iteration of the loop is given by the following formula:

$$\begin{aligned}
 W_i = & n_a \cdot c_a + n_{f_{ADD}} \cdot c_{f_{ADD}} + n_{f_{MUL}} \cdot c_{f_{MUL}} \\
 & + n_{f_{SUB}} \cdot c_{f_{SUB}} + n_{f_{DIV}} \cdot c_{f_{DIV}} + n_{fi} \cdot c_{fi} \\
 & + n_{fs} \cdot c_{fs} + n_{fu} \cdot c_{fu}
 \end{aligned}$$

Where  $c_{op}$  is the cost of performing a single operation with type  $op$  on the target machine. Thus, the total amount of work in an invocation of the loop is,

$$W(n_i) = n_i \cdot W_i$$

Since we cannot determine at compile-time the actual number of iterations in a loop in general, this formula is parameterized by  $n_i$ . When we estimate the amount of work in a loop that has branches, we give equal weights to each branch.

We determine the threshold value heuristically. First, we run several representative micro benchmark programs that contain many different type of parallel loops. After measuring sequential execution time and parallel execution time of each loop contained in the program on  $p$  processors, we plot the speedup of each loop on the Y-axis and the estimated cost (the estimated amount of work) on the X-axis. Then, draw a line from the point that has the lowest estimated cost ( $p_a$  in Figure 3.2) to the point that has the lowest estimated cost among those whose speedup is greater than 0.8 ( $p_b$ ). We choose the cost of the intersecting point with the horizontal line of speedup 1.0 ( $p_T$ ) as our threshold value.

When the loop is multiply nested, it is hard to determine the number of iterations of an inner loop before we run the outer loop. It is because the upper bound, lower bound, and step of the inner loop may change in the outer loop. In other words, it is hard to obtain the cost function parameterized by the numbers of iterations of both the outermost loop and the inner loop. In this case, we simply pass the loop to the run-time cost estimation model.

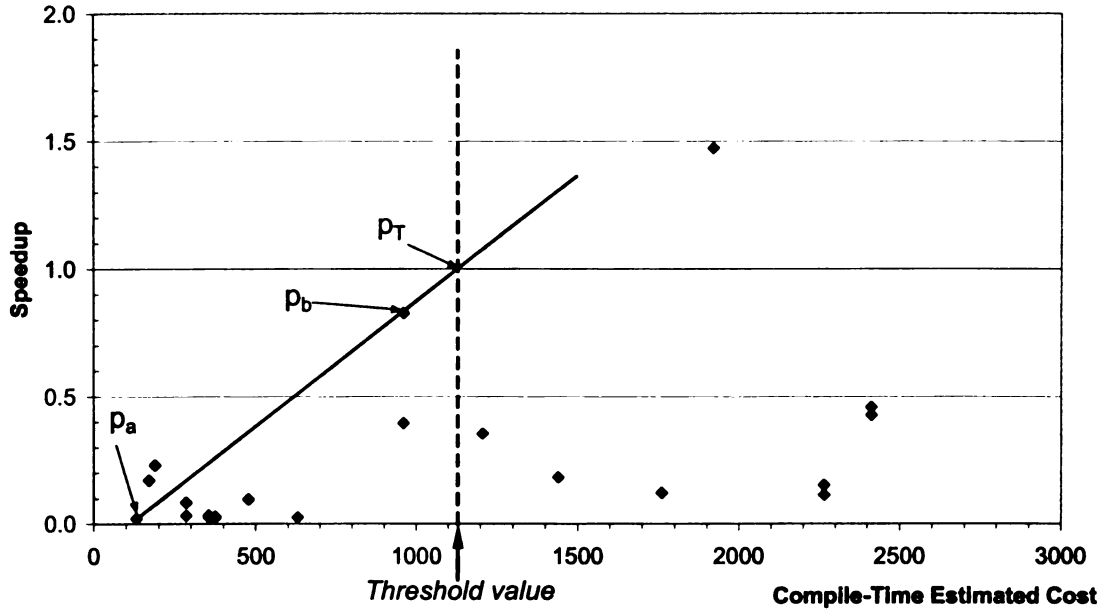


Figure 3.2: Determining the threshold value with compile-time cost estimation

## 3.2 Run-time Cost Estimation

While the compile-time cost estimation model estimates the performance of parallel loops that contain small amount of work and that are highly inefficient, the run-time cost estimation model identifies the loops that contain enough amount of computation that can overcome the parallel loop overhead and other overheads incurred during its execution. It also handles small loops that cannot be handled by the compile-time cost model due to some run-time parameters.

Since the number of instructions executed in a loop is proportional to its amount of work contained in it, we use the number of instructions executed (graduated) in an invocation of a parallel loop as the cost estimation. This measurement is much more accurate for estimating the cost than the execution time. For the run-time cost estimation model to be effective, the parallel loop must be invoked at least once in the program.

There are two threshold values to be determined in the run-time cost estimation

model. One (the lower threshold value) is for inefficient loops that contain small amount of work and that cannot be handled by the compile-time cost estimation model. The other (the higher threshold value) is for filtering out highly efficient loops. Heuristically determining the threshold values for the run-time cost estimation model is similar to the compile-time cost estimation model. We run several representative benchmark programs that contain many different types of parallel loops. We measure sequential execution time, parallel execution time, and the number of graduated instructions in the parallel execution of each loop in the benchmark programs on  $p$  processors. Then, we plot the speedup of each loop on the Y-axis and the number of graduated instructions (run-time estimated cost) on the X-axis.

For the higher threshold value, we draw a line from the point that has the lowest number of instructions executed ( $p_a$  in Figure 3.3) to the point that has the lowest number of instructions executed among those whose speedup is greater than the *average speedup* of all the loops plotted ( $p_b$  in Figure 3.3). We choose as our higher threshold value ( $T_H$ ) the cost of the point ( $p_H$ ) whose number of instructions executed is in the middle of the intersecting point with the speedup 1.0 ( $p_c$ ) and the intersecting point with the average speedup ( $p_d$ ).

For the lower threshold value, the method is the same as the compile-time cost estimation model. Draw a line from the point with the lowest cost ( $p_a$ ) to the point with the lowest cost among those whose speedup is greater than 0.8 ( $p_e$ ). We choose the cost of the intersecting point ( $p_L$ ) with the horizontal line of speedup 1.0 as our lower threshold value ( $T_L$  in Figure 3.3).

The region in between the lower threshold value and the higher threshold value is called the *adaptation window*.

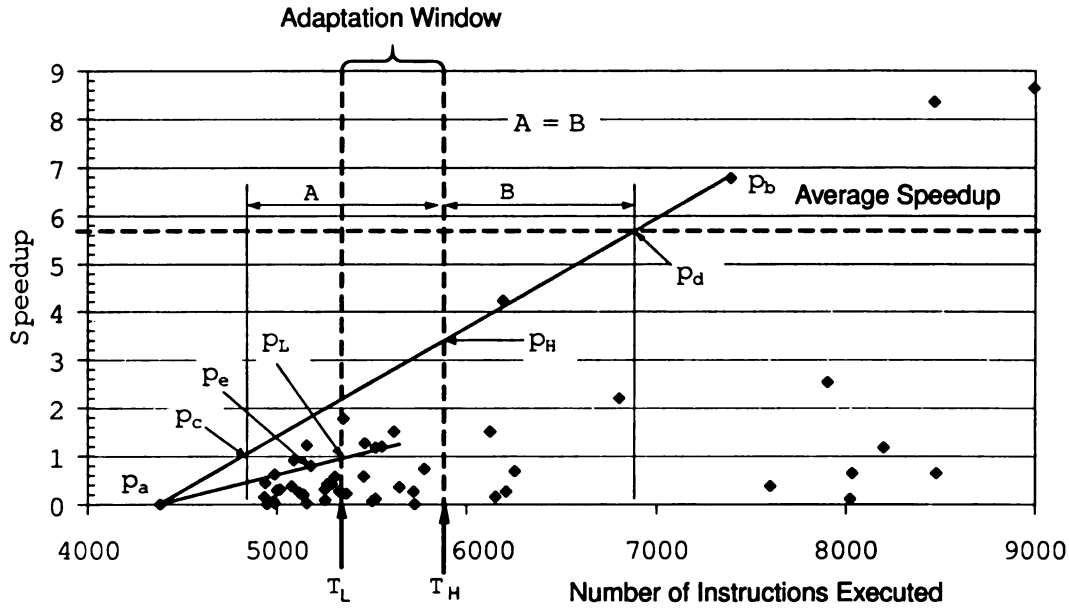


Figure 3.3: Determining the threshold value with run-time cost estimation.

### 3.3 Adaptive Execution Algorithms

Depending on the frequency of decision runs and the cost estimation model used, we propose five different adaptive execution schemes: First 2 Invocations with Timing (F2T), Most Recent with Timing (MRT), Static (compile-time) estimation and Most Recent with Timing (SMRT), MRT with Run-time cost estimation (MRTR), and MRT with Static (compile-time) and Run-time cost estimation (SMRTR). Our adaptive schemes are based on the observation that most of scientific applications have one outermost sequential loop and the parallel loops contained in it are invoked multiple times.

#### 3.3.1 First Two invocations with Timing (F2T)

In F2T, the loop is executed in parallel and timed when it is first invoked in the program. When it is invoked for the second time, it is executed sequentially and timed. Then we determine whether we run this loop parallelly or sequentially by comparing

the two time measurements. i.e. if the time measurement of the parallel execution is lower then we execute the loop in parallel, otherwise in serial. The loop is executed for the remaining invocations in the program in parallel or sequentially depending on the result of the comparison. The drawback of *First Two Invocations with Timing* (F2T) is that a highly efficient parallel loop has to be executed sequentially once in the decision runs. Also, the measured execution time in the decision runs is not the representative for the remaining invocations of the loop.

### 3.3.2 Most Recent with Timing (MRT)

When a loop is invoked for the first time, it is executed in parallel and timed. When it is invoked for the second time, it is executed sequentially and timed again. Then we determine whether to run this loop parallelly or sequentially in the next invocation by comparing the two measurements. However, the way we execute the loop for the remaining invocations is not fixed at this point. Instead, every time the loop is executed, we time it and compare the execution time to its most recent execution time in the other way. If the latter is lower, we change the way it runs. *Most Recent with Timing* can adapt to changes in the workload of the loop across invocations. It uses the recent past of a loop to predict its future behavior. Consequently, if the workload of the loop changes gradually, this strategy works well. However, sudden changes may cause this strategy to work poorly. Similar to F2T, the drawback of MRT is that a highly efficient parallel loop has to be executed sequentially at least once in the first decision run.



### 3.3.3 Static cost estimation and Most Recent with Timing (SMRT)

By combining compile-time cost estimation model with MRT, we can avoid running some loops inefficiently. As shown in Figure 3.1, the loop that contains smaller amount of work than parallel loop overhead can be filtered out by the compile-time cost estimation model. We run these loops sequentially. Then, the remaining loops are executed by MRT.

### 3.3.4 MRT with Run-time cost estimation (MRTR)

MRTR uses the notion of *adaptation window*. The adaptation window consists of two threshold values of the run-time cost estimation model. In this strategy, a loop is executed in parallel when it is first invoked. We measure the execution time and number of instructions executed (graduated) in the loop. If the number of instructions executed is less than the lower threshold value, the loop is executed sequentially for the remaining invocations. If the number of instructions executed is greater than the higher threshold value, it is executed in parallel for the remaining invocations. Otherwise, its execution in the remaining invocations follows MRT, which takes the decision based on the execution time. I.e. the loops with number of instructions falls into the adaptation window follow MRT scheme. A loop is neither highly efficient nor highly inefficient if it falls into the adaptation window.

### 3.3.5 MRT with Static and Run-time cost estimation (SMRTR)

SMRTR is a combination of Static and MRTR. Before it applies to MRTR, it filters out small inefficient loops by applying compile-time cost estimation model (Static), which runs them always sequentially. Then, MRTR is used for the remaining parallel

loops. For those inefficient loops filtered out by **Static**, we do not pay the penalty of executing them in parallel at least once in **MRTR** because their cost is estimated at compile-time. Consequently, we expect that **SMRTR** gives the best performance.

# Chapter 4

## Evaluation Environment

### 4.1 Compiler

We have implemented cost estimation and adaptive execution algorithms, described in chapter 3, in a compiler preprocessor. The preprocessor is written in Perl. In order to obtain parallelization information for the program, we use SGI MIPSpro Fortran77 compiler [13] with the automatic parallelization option (APO). When we consider the automatic parallelization, APO parallelizes the loop conditionally, and generate code for both parallel and sequential versions. So, it can avoid running a loop in parallel if the loop has a smaller trip count. APO select the code version based on the trip count, the code inside the loop's body, the number of processors available and an estimate of the cost to invoke a parallel loop in that runtime environment [13].

The parallelization information together with the original program is fed into our compiler preprocessor. The preprocessor inserts instrumentation and adaptive execution code with appropriate directives in to the original program to direct the compiler of the target multiprocessor machine (SGI Origin2000 in our case). The directives inserted are summarized in Table 4.1. The output program from the preprocessor is compiled by the SGI Fortran77 compiler with the APO option to generate

Directive	Meaning
<b>C** ASSERT DO (SERIAL)</b>	Instructs the Auto-Parallelizing Option not to parallelize the loop following the assertion.
<b>C** ASSERT DO PREFER (CONCURRENT)</b>	Instructs the Auto-Parallelizing Option to parallelize the loop following the assertion, if it is safe to do so.

Table 4.1: Auto-Parallelizing directives used for SGI MIPSpro Fortran77 compiler.

an executable.

To measure the execution time of an invocation of each loop, we use an SGI system call from *syssgi* to read the processor cycle counter. To count the number of graduated instructions of each loop in an invocation, we use SGI *perfex* library to access processor event counters. The number of graduated instructions from the master thread is counted.

## 4.2 Applications

We evaluate the effectiveness of our algorithms using scientific applications written in Fortran77. We selected applications that are highly parallel. They are Applu, Mgrid and Swim from SPECfp2000, Hydro2d and Su2cor from SPECfp95 and Cg, Lu, Mg and Sp from NAS benchmarks. Table 4.2 shows the problem sizes and number of iterations used by each of the applications.

## 4.3 Target Architecture

The code generated by our system is targeted to SGI Origin2000 at NCSA [9]. All our experiments are done in the dedicated mode of the SGI Origin2000. Table 4.3 shows the parameters of this architecture.

Application	Source	Number of Lines	Data Size and Number of Iterations
Applu	SPECfp2000	3980	Reference input with 20 iterations
Cg	Nas	1322	4 iterations
Hydro2d	SPECfp95	4303	Reference input with 100 iterations
Lu	Nas	3989	40 iterations
Mg	Nas	1643	6 iterations
Mgrid	SPECfp2000	489	Test input with 40 iterations
Sp	Nas	3302	40 iterations
Su2cor	SPECfp95	2271	Reference input with 100 iterations
Swim	SPECfp2000	435	Reference input with 50 iterations

Table 4.2: Applications used for the evaluation.

Architecture	Distributed Shared Memory
Processor Type (Clock speed)	MIPS R10000 (250MHz)
Number of Processors	128
Total Memory	128 GB
Total Disk	640 GB
Instruction cache size (cache line size)	32 KB (64 B)
Data cache size (cache line size)	32 KB (32 B)
Secondary unified instruction/data cache size (cache line size)	4 MB (128 B)

Table 4.3: Machine specification.

# Chapter 5

## Evaluation

Before we evaluate our adaptive execution strategies, we first examine the characteristics of the parallel loops in each applications (Section 5.1). We then evaluate the performance of our strategies (Section 5.2).

### 5.1 Characteristics of the Parallel Loops

Table 5.1 shows the characteristics of the parallel loops in each application. The table gives us the rationale of our adaptive execution strategies. It has one section for all the parallel loops and another section for inefficient parallel loops in each application.

The first row in the first section shows the total number of parallel loops in each application and their % sequential execution time relative to the sequential execution time of the application. The second row in the first section shows the average number of invocations for each individual parallel loop in the application. The last row in the first section shows the average loop size measured in number of processor cycles that it takes to execute one invocation of the loop.

The first row in the second section shows the total number of inefficient loops in each application for different number of processors. It also shows the % sequential execution time relative to the sequential execution time of the application and %

parallel execution time of inefficient loops relative to the parallel execution time of the application for different number of processors. The second row in the second section shows the average number of invocations for each individual inefficient parallel loop in the application for different number of processors. The last row in the second section shows the average loop size measured in number of processor cycles that it takes to execute one invocation of the loop in parallel on different number of processors.

We see that the applications are highly parallel and that the parallel loops account for an average of 94.5% of the sequential execution time. Applu, Hydro2d, Lu, Sp and Su2cor contains many inefficient parallel loops. More than 30% of their parallel loops are inefficient. Inefficient parallel loops do not affect the sequential execution time because they account for at most 4.3% of the sequential execution time on average. However, inefficient parallel loops dominate in parallel execution for all applications except Cg, Mg and Mgrid. They account for average 13.9% of the parallel execution time of the applications on 2 processors and their execution time covers up to average 58.8% of the parallel execution time on 32 processors. The % parallel execution time of inefficient parallel loops increases as the number of processors increase. This is due to the parallel loop overhead incurred by the inefficient parallel loops. Consequently, we see that inefficient parallel loops are a significant target for optimizations.

Most of the parallel loops are invoked many times (15527.8 times on average). Inefficient parallel loops are invoked much more times than the other parallel loops (for example, 35420.6 times on 32 processors). The size of the inefficient parallel loops is much less than the size of other parallel loops in all the applications except in Cg, Mgrid, Sp and Swim. Therefore, the overhead involved in executing small parallel loops in parallel is likely to be large compared to their execution time. This means that our serialization technique is an effective optimization technique for parallel programs.

			Applu	Cg	Hydro2d	Lu	Mg
Parallel Loops	Number of Parallel Loops (% sequential time)		55	16	86	37	20
			(95.2%)	(86.0%)	(97.3%)	(99.2%)	(85.7%)
		Average Number of Invocations	8779.7	40.7	373.0	82811.0	137.6
	Average Loop Size (processor cycles)	127.3K	138.2K	4.2K	119.0K	127.9K	
Inefficient Loops	Number of Inefficient Loops  (% sequential time)  (% parallel time)	#procs	Applu	Cg	Hydro2d	Lu	Mg
		2	28 (0.9%) (16.5%)	5 (0.5%) (5.0%)	34 (0.6%) (13.3%)	19 (3.1%) (25.8%)	10 (1.1%) (10.5%)
		4	29 (0.9%) (28.1%)	6 (0.5%) (4.6%)	28 (0.2%) (20.8%)	18 (4.3%) (59.4%)	7 (0.2%) (10.9%)
		8	28 (0.9%) (44.3%)	7 (0.5%) (5.7%)	28 (0.2%) (28.9%)	17 (2.8%) (77.4%)	5 (0.0%) (13.7%)
		16	29 (0.9%) (57.2%)	3 (0.4%) (6.8%)	28 (0.2%) (37.1%)	18 (2.8%) (82.3%)	5 (0.0%) (14.4%)
		32	32 (1.4%) (79.6%)	3 (0.3%) (8.2%)	32 (0.3%) (44.8%)	18 (2.8%) (98.7%)	5 (0.0%) (16.9%)
	Average Number of Invocations	2	16871.3	26.4	357.7	160330.3	83.4
		4	16289.6	22.2	309.4	169243.0	48.4
		8	16871.3	19.1	337.9	179196.1	1.8
		16	16289.6	1.0	312.9	169240.8	1.8
		32	15077.8	1.0	351.9	169240.8	1.8
	Average Loop Size (processor cycles)	2	1.4K	90.6K	0.6K	36.6K	0.9K
		4	3.7K	76.7K	0.6K	5.7K	0.2K
		8	1.4K	66.8K	0.5K	0.2K	0.0K
		16	1.4K	152.4K	0.5K	5.2K	0.0K
		32	1.3K	151.9K	0.4K	5.2K	0.0K

Table 5.1: Characteristics of parallel loops in the applications.



		Mgrid	Sp	Su2cor	Swim	Average	
Parallel Loops	Number of Parallel Loops (% sequential time)	11 (99.7%)	76 (98.9%)	41 (88.8%)	16 (99.8%)	39.8 (94.5%)	
	Average Number of Invocations	422.7	12937.3	34220.1	28.4	15527.8	
	Average Loop Size (processor cycles)	44.3K	46.0K	0.6K	364.6K	108.0K	
Inefficient Loops	Number of Inefficient Loops  (% sequential time)  (% parallel time)	#procs	Mgrid	Sp	Su2cor	Swim	Average
		2	1 (0.5%) (0.9%)	21 (4.9%) (17.3%)	15 (2.8%) (25.2%)	3 (0.9%) (10.2%)	15.1 (1.7%) (13.9%)
		4	–	17 (4.5%) (23.0%)	15 (2.8%) (34.5%)	3 (4.2%) (21.6%)	15.4 (2.2%) (25.4%)
		8	–	28 (22.8%) (81.9%)	16 (2.8%) (48.6%)	3 (4.2%) (33.4%)	16.5 (4.3%) (41.7%)
		16	1 (0.2%) (1.2%)	21 (22.5%) (68.9%)	18 (2.8%) (62.5%)	3 (4.2%) (51.8%)	14.0 (3.8%) (42.5%)
		32	–	26 (22.8%) (85.4%)	18 (2.8%) (70.0%)	6 (4.2%) (67.1%)	17.5 (4.3%) (58.8%)
	Average Number of Invocations	2	1042.0	31138.4	80892.9	17.3	32306.6
		4	–	38455.0	80892.9	2.3	38157.8
		8	–	28989.7	75837.2	2.3	37656.9
		16	1.0	38640.8	67458.8	2.3	32438.8
		32	–	31214.9	67458.8	18.0	35420.6
	Average Loop Size (processor cycles)	2	0.4K	8.1K	0.2K	368.7K	56.4K
		4	–	7.2K	0.2K	633.5K	91.0K
		8	–	31.1K	0.5K	633.5K	91.7K
		16	179.2K	40.3K	0.4K	633.5K	112.5K
		32	–	46.3K	0.4K	317.0K	65.3K

Table 5.1 (Cont'd)

## 5.2 Results

In this section, we analyze the results obtained from the execution of the applications. Execution time for each application under each strategy for different number of processors (1, 2, 4, 8, 16, 32) is shown in the Appendix (Table 1). Figure 5.1 to Figure 5.10 compare the execution times of the applications under several strategies. For each application, there are 5 groups of bars. Each group corresponds to the number of processors 2, 4, 8, 16, and 32. The leftmost bar (**Base**) in each group corresponds to the execution time of the original parallel program. All parallel loops that are identified by the SGI parallelizing compiler are contained in the original parallel program, and they run in parallel in **Base**. The remaining bars correspond to the our strategies described in Chapter 3: First Two invocations with Timing (**F2T**), Most Recent with Timing (**MRT**), Static cost estimation (**Static**), Run-time cost estimation (**Runtime**), Static cost estimation and **MRT** (**SMRT**), **MRT** with Run-time cost estimation (**MRTR**), and **MRT** with Static and Run-time cost estimation (**SMRTR**). In each application, all the bars are normalized to **Base** (the smaller, the better).

The First Two invocations with Timing (**F2T**) is the worst. This is because it uses only the first two invocations of a loop in the decision runs to determine the way it runs for the remaining invocations. Unless the amount of computation of the parallel loop is constant across invocations, the decision is likely to be inaccurate. Also, it runs each parallel loop sub-optimally at least once.

The Most Recent with Timing (**MRT**) is better than **F2T** and some times better than **Base**. The reason why it can do better than **F2T** is that it chooses the way to run each individual parallel loop in an adaptive manner. However, in the process of doing so, it runs each parallel loop sub-optimally at least once. Under some conditions, changes in the amount of work in a parallel loop across invocations may confuse **MRT** and make it slow.

**Static** runs the parallel loops according to the result of compile-time cost esti-

mation model. From the figures, we can see that **Static** is better than **Base** for most of the applications (**Applu**, **Cg**, **Hydro2d**, **Lu**, **Sp** and **Su2cor**). This is because it can estimate the cost of most of the parallel loops in these applications, and these loops tend to be small (note that it does not estimate the cost of doubly nested parallel loops) and invoked many times in the applications. In addition, it estimates the cost of a parallel loop before it runs. Thus, it does not pay the penalty caused by the decision runs. Table 5.2 shows, number of parallel loops whose amount of work is estimated by the compile-time cost estimation model. It also shows their % sequential time and average number of invocations. From the Table 5.2 we can see that the average number of invocations for most of the applications are high. Overall, **Static** is attractive because of its simplicity.

Application	Number of Parallel Loops	% Sequential Time	Average Number of Invocations
Applu	18	0.51%	25122.4
Cg	13	1.50%	40.0
Hydro2d	31	0.36%	389.0
Lu	14	2.72%	234328.4
Mg	6	5.10%	248.3
Mgrid	1	0.00%	1.0
Sp	9	16.17%	106350.7
Su2cor	32	88.50%	43801.6
Swim	8	0.44%	37.3
Average	14.7	12.81%	45591.0

Table 5.2: Characteristics of parallel loops whose amount of work can be estimated by the compile-time cost estimation model.

The Run-time cost estimation **Runtime** is better than **F2T**, **MRT** and **Base**, but slightly worse than **Static**. This is because it estimates the cost by counting the number of instructions executed in the loop. Thus, the estimation is accurate. However, in the process of doing so, it runs each parallel loop sub-optimally at least once.

The Static cost estimation with **MRT** (**SMRT**) is better than **F2T** and **MRT**, but not better than **Base**. The reason is that the penalty from the decision runs of **MRT** is much bigger than the benefits obtained from **Static**. Even though some parallel loops

are filtered by **Static**, running the remaining parallel loops sub-optimally at least once still affects the performance a lot.

The **MRT** with run-time cost estimation (**MRTR**) is much better than **Base** and **Static** in **Cg**, **Mg**, **Sp** and **Su2cor**. It is comparable to **Static** in **Applu**, **Hydro2d**, **Mgrid** and **Swim**. Since it estimates the cost of a parallel loop more accurately than **Static** by using the run-time information, more inefficient loops that tend to be small are filtered by the run-time cost estimation model and are executed sequentially in **MRT**. Consequently, there is no penalty of running efficient (big) parallel loops sequentially at least once.

As expected, **MRT** with static and run-time cost estimation (**SMRTR**) is the fastest. This is because it runs only the parallel loops which fall in the adaptation window at least once sub-optimally. Even though we see a slight performance degradation in **Mgrid** and **Swim** due to the penalty caused by decision runs, it significantly reduces the penalty caused by the decision runs of **MRT**. Moreover, it uses both compile-time and run-time cost estimation model. Consequently, it selects inefficient loops more accurately. The parallel program with **SMRTR** runs 30%, 25%, 20% and 14% faster on average than the original parallel program on 32, 16, 8 and 4 processors respectively. It shows comparable results (1% improvement) on 2 processors.

The speedup obtained by **SMRTR** for each application is shown by Table 5.3 and graphed in Figure 5.11. **Lu** with **SMRTR** runs more than twice faster than **Base** on 16 and 32 processors. Similarly, **Su2cor** shows such performance for 32 processors. Figure 5.11 further reveals that we have improved the scalability of most of the applications (**Applu**, **Cg**, **Hydro2d**, **Lu**, **Mg**, **Mgrid** and **Swim**) except **SP** and **Su2cor**. Overall, **SMRTR** is the best strategy for adaptively executing the applications.

We have also compared our best strategy (**SMRTR**) with automatic parallelization option (**APO**) of the SGI MIPSpro Fortran77 compiler. The execution time of the auto-parallelized programs (**APO**) for each application is shown in Appendix

Application	Scheme	Number of Processors				
		2	4	8	16	32
Applu	Base	1.48	2.01	2.92	3.55	3.47
	SMRTR	1.53	2.55	3.47	4.55	5.55
Cg	Base	1.34	1.26	1.41	1.91	2.20
	SMRTR	1.40	1.38	1.86	2.08	2.52
Hydro2d	Base	1.56	2.53	3.49	4.44	4.95
	SMRTR	1.75	2.84	4.95	7.02	7.74
Lu	Base	1.41	1.75	2.06	1.71	1.71
	SMRTR	1.81	2.85	4.10	4.89	4.87
Mg	Base	1.45	1.77	2.32	2.41	2.71
	SMRTR	1.31	2.03	3.04	3.17	3.18
Mgrid	Base	1.56	2.62	4.14	5.31	5.46
	SMRTR	1.11	2.56	3.91	5.25	5.42
Sp	Base	1.30	1.29	1.54	1.18	0.92
	SMRTR	1.31	1.57	1.75	1.68	1.65
Su2cor	Base	1.17	1.43	1.38	1.13	0.73
	SMRTR	1.34	1.66	2.02	1.88	1.83
Swim	Base	1.49	2.12	3.21	4.81	5.91
	SMRTR	1.50	2.11	3.26	4.76	5.75

Table 5.3: Speedup of Base and SMRTR strategies of each application.

(Table 2). Figure 5.12 shows the normalized execution time of SMRTR for each application. Here, all the bars are normalized to APO and therefore APO time is 1.0. From this figure, it is clear that SMRTR strategy of Hydro2d, Mg and Sp shows better performance (all less than 1.0 line) than the corresponding APO version. Furthermore, Mg runs 50% faster than APO on 32 processors. Also, Cg and Su2cor shows good performance for some cases. Execution time for Applu and Lu is somewhat comparable to APO. Only Mgrid and Swim shows slightly worse results. On average, parallel programs with SMRTR runs 1%, 2% and 5% faster than the APO programs on 32, 16 and 8 processors respectively. SMRTR shows comparable results to APO on 4 processors and runs slightly slower than APO on 2 processors. We see such slight performance degradation in SMRTR due to the penalty caused by decision runs in Mgrid and Swim. Without Mgrid and Swim, SMRTR runs on average 12%, 9%, 14%, 5% and 1% faster than the auto-parallelized programs on 32, 16, 8, 4 and 2 processors respectively.

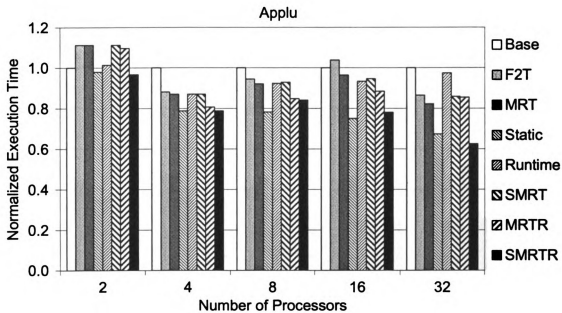


Figure 5.1: Normalized execution time of Applu

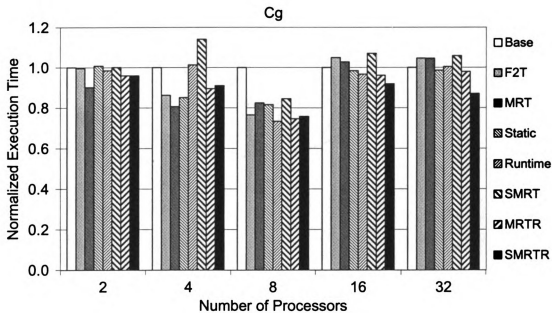


Figure 5.2: Normalized execution time of Cg

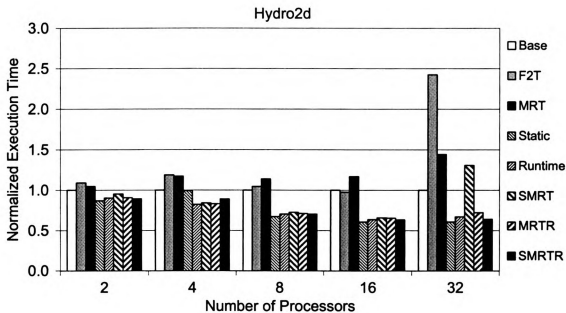


Figure 5.3: Normalized execution time of Hydro2d

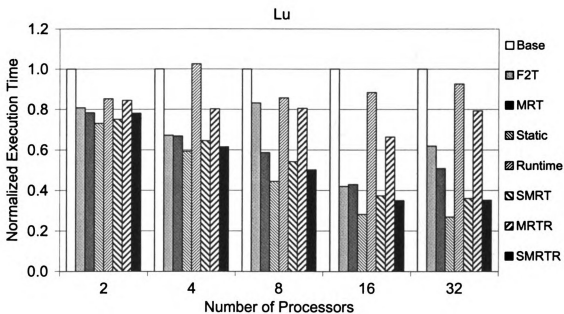


Figure 5.4: Normalized execution time of Lu

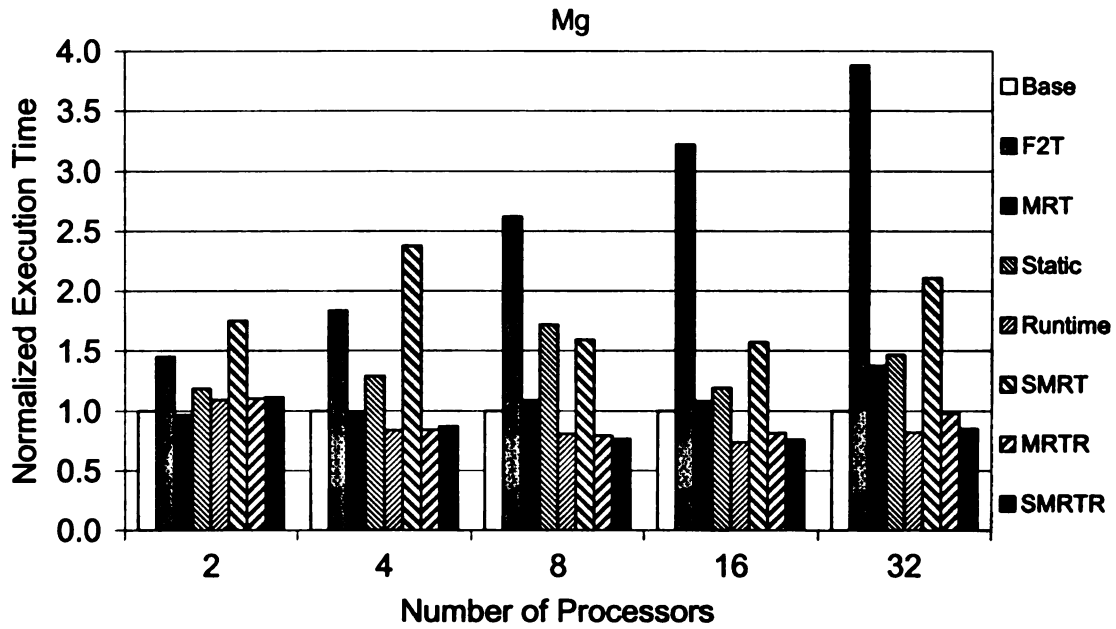


Figure 5.5: Normalized execution time of Mg

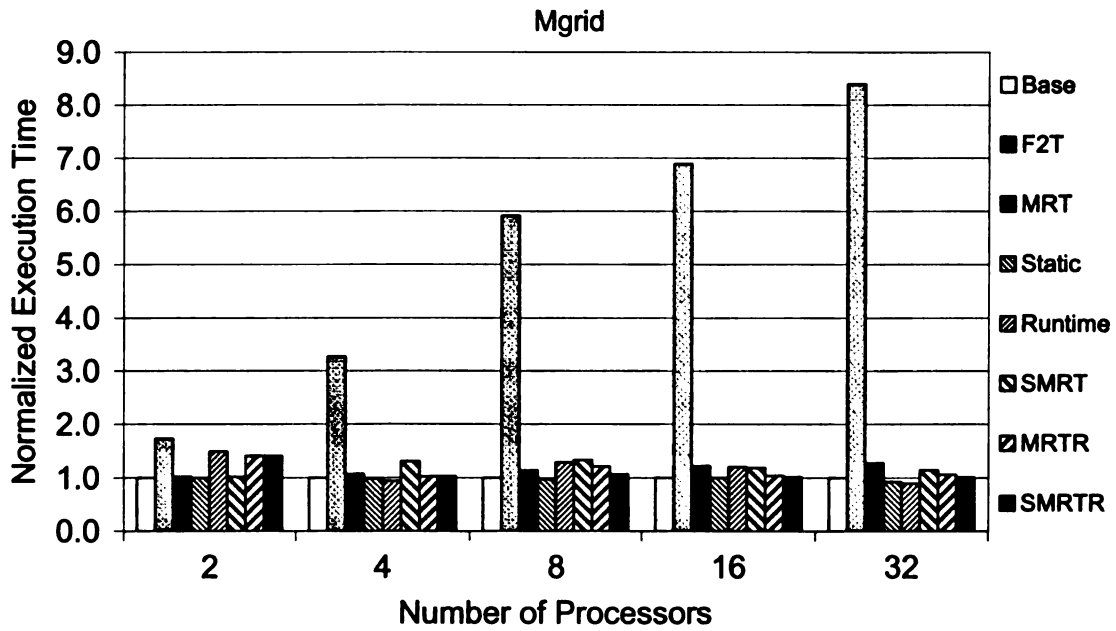


Figure 5.6: Normalized execution time of Mgrid



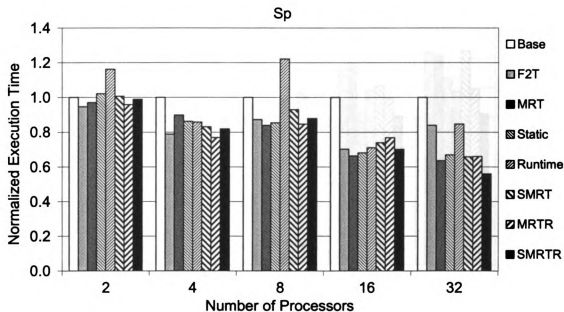


Figure 5.7: Normalized execution time of Sp

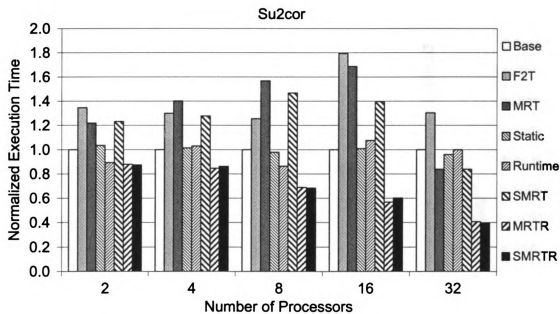


Figure 5.8: Normalized execution time of Su2cor

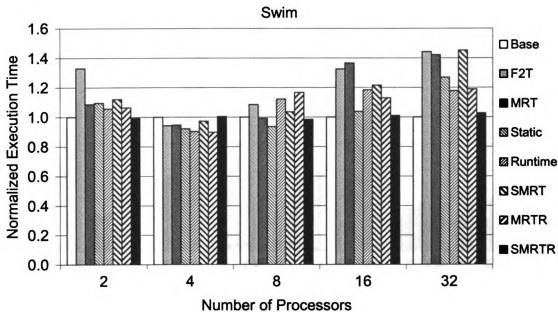


Figure 5.9: Normalized execution time of Swim

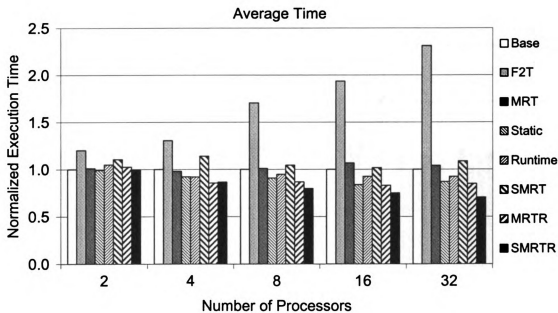


Figure 5.10: Average normalized execution time of all the applications

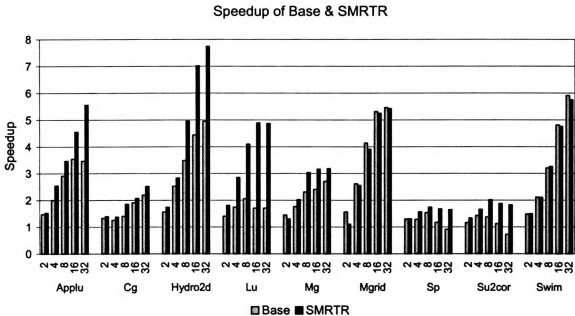


Figure 5.11: Speedup of Base and SMRTR for each application.

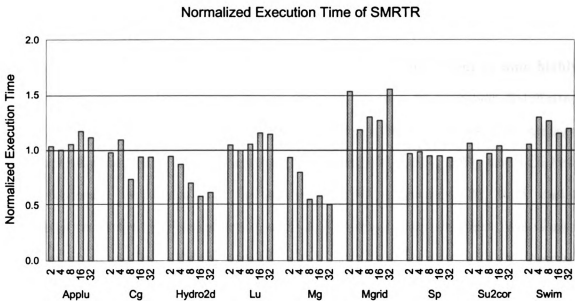


Figure 5.12: SMRTR execution time normalized to APO.

# Chapter 6

## Conclusions

In this thesis, we proposed performance estimation and adaptive execution techniques that determine whether a parallel loop is executed parallelly or sequentially in order to maximize performance and scalability. We have developed compile-time cost estimation model, run-time cost estimation model and a set of algorithms (F2T, MRT, SMRT, MRTR and SMRTR) that utilize them. The adaptation and performance estimation algorithms are in the code that is inserted into the original parallel program by the compiler preprocessor.

We tested our adaptive execution algorithms by applying them to nine highly parallel numerical applications written in Fortran 77 on a 32 processor distributed shared memory machine (SGI Origin2000). We Found **SMRTR** as the best strategy for adaptively executing the applications. It showed 30%, 25%, 20%, and 14% better performance on average than the original parallel programs on 32, 16, 8, and 4 processors respectively. Further more, some applications run more than twice faster than the corresponding original parallel version. Also, we have showed an improvement in the scalability of most of the applications tested. The results indicate that our adaptive execution techniques are promising to speed-up the programs that are already parallel.

We have also compared our results with the automatic parallelization option (APO) available in the SGI MIPSpro Fortran77 compiler. Most of the applications with SMRTR adaptive strategy showed higher performance, when compared with the auto-parallelized programs. This shows that the adaptive program optimization is clearly an important technique that can be applied to modern compilers.

Our work is being extended by designing similar performance estimation and adaptation techniques in multiprogramming environments on multiprocessors to improve performance and scalability of parallel programs. Also, compile time cost estimation model can be further improved by considering some of the multiply nested loops.

## APPENDIX

Scheme		Number of Processors					
		1	2	4	8	16	32
A P P L U	Base	141037822	95055216	70109805	48243972	39775875	40606595
	F2T	133060696	100236123	58406268	43123428	38897834	33199368
	MRT	132929650	100225680	57619748	41958134	36165534	31529565
	Static	131968472	87129226	51742269	35342355	27875716	25619552
	Runtime	132335268	90749971	57396604	41832604	34845518	37218574
	SMRT	132133508	99692692	57252840	42109395	35261800	32706988
	MRTR	145902502	107893522	58543080	42458062	36340637	35913536
	SMRTR	133770209	87560236	52554184	38509361	29370683	24115588
C G	Base	130516723	97436829	103962670	92355857	68336157	59371900
	F2T	135537610	97068583	89718721	70729699	71750298	62134078
	MRT	127513285	87895829	83870787	76189826	70271741	62069405
	Static	138870333	98225424	88524665	75428179	67311238	58592767
	Runtime	135199382	95992273	105276277	67801417	66128944	59676819
	SMRT	136896546	97547513	118628166	78138484	73141256	62847659
	MRTR	134241753	93595425	93111421	68971703	65739947	58217851
	SMRTR	136266308	93541708	94734212	70071562	62849109	51728663
H Y D R O 2 D	Base	115572090	74197234	45603815	33152255	26005942	23347087
	F2T	117787470	82482370	55178770	35299516	25833238	57721674
	MRT	114345404	76627496	52937494	37233379	30076668	33350942
	Static	118193014	65788415	46308443	22770266	16175384	14487994
	Runtime	114330086	66267860	37133684	23114273	16358286	15445534
	SMRT	114427572	69840092	38072170	23781388	16979204	30212010
	MRTR	114307553	66423212	37558006	23269156	16814547	16651186
	SMRTR	116301082	66460173	40967082	23472516	16566958	15026017
L U	Base	249926377	176888789	142840215	121502039	145874913	145871570
	F2T	249996694	142964154	95974369	101083039	61270502	90253768
	MRT	249528590	138675519	95379641	71172921	62507932	74115385
	Static	260365093	129291737	84697765	53930573	41090058	39255364
	Runtime	262493838	150937720	146486356	104158100	129094109	135100286
	SMRT	250247760	132922405	92136389	65772580	54525046	52428090
	MRTR	268506697	149412467	114645016	97936374	96820804	115633320
	SMRTR	254669214	138239144	87829360	60952992	51080216	51344012
M G	Base	128551302	88370778	72671522	55349588	53420001	47493004
	F2T	122692607	128233551	133346977	144874590	171862884	184231826
	MRT	121400853	85337447	72113323	60167220	57804521	65432964
	Static	120268753	104743514	93725840	95047949	63546111	69635492
	Runtime	120128822	96473584	60914326	44778026	39340931	39035808
	SMRT	121190217	154816484	172690116	88212580	83905529	100136443
	MRTR	121605112	97637614	61080338	43844971	43343292	46543088
	SMRTR	120140633	98469888	63278757	42331938	40528504	40488198

Table 1: Execution time of the applications under each strategy.

Scheme		Number of Processors					
		1	2	4	8	16	32
M G R I D	Base	83342998	53524121	31836110	20121215	15683608	15260138
	F2T	83427058	92418438	104838472	119100409	108949933	128520750
	MRT	83953498	55031009	34233662	23123144	19365509	19648656
	Static	82964364	53068570	31160510	19495931	15602500	13977418
	Runtime	87503949	83231181	31134072	27238244	19795888	14320246
	SMRT	85807142	55842300	42605697	27588882	19126863	17973480
	MRTR	83188994	75100843	32264556	24237362	16263939	15999220
	SMRTR	83066275	75006166	32396479	21247781	15831643	15316101
S P	Base	208220397	160600344	161402984	135487643	176367700	225755289
	F2T	206731830	152150546	127225191	118279953	123842527	189326715
	MRT	208917997	155818038	144986839	113716975	117092110	143526663
	Static	209650034	164038513	139152091	115665170	120221014	151077104
	Runtime	211294060	186531273	138446481	165502112	125434382	191232853
	SMRT	206916996	161536415	134111325	126006789	130399261	148543542
	MRTR	209273414	153959336	124150936	114686469	135476639	149128839
	SMRTR	207547804	158901288	132245736	119185488	123711832	126404914
S U 2 C O R	Base	131562075	112462242	92124610	95424942	116254845	180445997
	F2T	127949430	146915778	116244447	116010795	202076926	227051060
	MRT	127492328	132890306	125354626	144837758	188982933	146863784
	Static	133836950	118606729	94612508	95045768	118969602	176597342
	Runtime	129566180	98643224	93210060	80754810	122956680	176726210
	SMRT	127614107	133701768	113481521	135136585	157442976	147392979
	MRTR	132546492	99291038	78586261	66136348	66459926	74001530
	SMRTR	132645182	99160151	80137615	65769406	70497452	72534136
S W I M	Base	120961904	80933976	56944548	37656852	25159114	20467103
	F2T	115554595	103107841	51325614	39026338	31876250	28202467
	MRT	112500451	82068230	50136694	34713754	31918626	27038870
	Static	109443276	80748684	47664805	31887379	23622868	23493588
	Runtime	116448276	82702884	49497261	40727331	28684924	23207348
	SMRT	108785582	81979186	49802146	35141460	27477331	26722726
	MRTR	114230371	81752623	48466658	41465885	26825685	22982571
	SMRTR	120910798	80648662	57309883	37079126	25412580	21021418
A V E R A G E	Base	1309691688	939469529	777496279	639294363	666878155	758618683
	F2T	1292737990	1045577384	832258829	787527767	836360392	1000641706
	MRT	1278582056	914569554	716632814	603113111	614185574	603576234
	Static	1305560289	901640812	677588896	544613570	494414491	572736621
	Runtime	1309299861	951529970	719495121	595906917	582639662	691963678
	SMRT	1284019430	987878855	818780370	621888143	598259266	618963917
	MRTR	1323802888	925066080	648406272	523006330	504085416	535071141
	SMRTR	1305317505	897987416	641453308	478620170	435848977	417979047

Table 1 (Cont'd)



Application	Number of Processors					
	1	2	4	8	16	32
Applu	131386241	84925197	52543173	36617852	25159283	21686166
Cg	131251445	95993210	86829839	95602531	67055970	55291834
Hydro2d	114426479	70616486	47191256	33549170	28578289	24465276
Lu	248091760	132415820	88281872	57977870	44290736	45008928
Mg	126954842	105623641	79410158	76586385	69646999	80179550
Mgrid	82869449	49028855	27420646	16361963	12496983	9888567
Sp	208289503	164654253	134768739	126113796	130848292	136003232
Su2cor	127316396	93708067	88620495	68158988	68087209	78285611
Swim	107714483	76804007	44294308	29383733	22118263	17615819

Table 2: Execution time of the applications in APO mode.

## BIBLIOGRAPHY

## BIBLIOGRAPHY

- [1] Bowen Alpern et al. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000.
- [2] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.
- [3] OpenMP Standard Board. *OpenMP Fortran Interpretations*, April 1999. Version 1.0.
- [4] Mark Byler, James Davies, Christopher Huson, Bruce Leasure, and Michael Wolfe. Multiple Version Loops. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 312–318, August 1987.
- [5] Calin Cascaval, Luise DeRose, David A. Padua, and Daniel Reed. Compile-Time Based Performance Prediction. In *Proceedings of the 12th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 365–379, August 1999.
- [6] Rohit Chandra, Leo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Manon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publisher, 2001.
- [7] Alan L. Cox and Robert. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th International Symposium on Computer Architectur*, pages 98–108, May 1993.
- [8] Pedro Diniz and Martin Rinard. Dynamic Feedback: An Effective Technique for Adaptive Computing. In *Proceedings of the ACM SIGPLAN Conference on Program Language Design and Implementation*, pages 71–84, June 1997.
- [9] The National Center for Supercomputing Applications. <http://www.ncsa.uiuc.edu>.
- [10] Rajiv Gupta and Rastislav Bodik. Adaptive Loop Transformations for Scientific Programs. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, pages 368–375, October 1995.

- [11] Urs Holzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 326–336, June 1994.
- [12] Silicon Graphics Inc. *MIPSpro Auto-Parallelization Option Programmer's Guide*.
- [13] Silicon Graphics Inc. *MIPSpro Fortran 77 programmer's Guide*.
- [14] Jaejin Lee. *Compilation Techniques for Explicitly Parallel Programs*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1999. Department of Computer Science Technical Report UIUCDCS-R-99-2112.
- [15] Jaejin Lee, Yan Solihin, and Josep Torrellas. Automatically Mapping Code in an Intelligent Memory Architecture. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA)*, pages 121–132, January 2001.
- [16] R. L. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, December 1970.
- [17] Martin Rinard and Pedro Diniz. Eliminating Synchronization Bottlenecks in Object Based Programs Using Adaptive Replication. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 83–92, June 1999.
- [18] Theodore H. Romer, Dennis Lee, Brian N. Bershad, and Bradley Chen. Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 255–266, November 1994.
- [19] Rafael H. Saavedra and Daeyeon Park. Improving the Effectiveness of Software Prefetching with Adaptive Execution. In *Proceedings of the Conference on Parallel Algorithms and Compilation Techniques*, October 1996.
- [20] Michael J. Voss and Rudolf Eigenmann. ADAPT: Automated De-Coupled Adaptive Program Transformation. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, page 163, August 2000.
- [21] Michael J. Voss and Rudolf Eigenmann. High-level Adaptive Program Optimization with ADAPT. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 93–102, June 2001.

MICHIGAN STATE UNIVERSITY LIBRARIES



3 1293 02372 6726