UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of a doctoral thesis by

Iffat Hoque Kazi

and have found that it is complete and satisfactory in all respects,

and that any and all revisions required by the final

examining committee have been made.

David J. Lilja

____

Name of Faculty Adviser(s)

____

Signature of Faculty Adviser(s)

____

Date

GRADUATE SCHOOL

A Dynamically Adaptive Parallelization Model Based on Speculative Multithreading

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Iffat Hoque Kazi

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

David J. Lilja, Adviser

September, 2000

© Iffat Hoque Kazi 2000

Acknowledgments

I would like to thank my advisor, Professor David J. Lilja, for his guidance, support, and encouragement during the last five years. I would also like to thank the members of my Ph.D. examination committee, Professor Pen-Chung Yew, Professor Larry Kinney, and Professor George Karypis, for their valuable comments and feedbacks on my research work.

Thanks to my fellow graduate students in the ARCTic research group, especially, Robert Glamm, Christian Hescott, Jian Huang, Jun-Seong Kim, Qing Zhao and all the others for their friendship, and their help whenever I needed.

Finally, I would like to thank my parents, Justice Kazi Ebadul Hoque and Dr. Sharifa Khatun, and my sisters, Shamim, Sabera, and Zinat, for all their love, support, and encouragement that kept me going during the last five years. And, last, but not the least, thanks to my husband, Badrul Munir Sarwar, for being a wonderful, understanding, supportive, and loving companion throughout this effort.

Abstract

A wide variety of application programs have inherent loop-level parallelism that can be exploited on shared-memory multiprocessor systems for performance improvement. Parallelizing compilers are available that can automatically identify and extract some of the loop-level parallelism in sequential programs. Traditional parallelizing compilers, however, often cannot extract much of the available parallelism due to the inherent limitations of compile-time information. Loops with potential data dependences that are caused by run-time values, or loops with sequential constructs (e.g. *do-while* loops), in particular are impossible to parallelize using these compilers. The parallelism in such loops can be exploited only by using run-time parallelization techniques since the information required to parallelize these loops is available only at run-time. Once a program has been parallelized, it is necessary to efficiently execute it in parallel to improve its performance. Compiler parallelized code cannot predetermine the number of processors a parallel code region can most efficiently use, however, especially, if the program behavior changes dynamically or the system load in a multiprogrammed environment changes as the program executes. With a static mapping assigned by the compiler, the parallel execution performance may suffer due to an insufficient workload in each parallel task, or due to resource contention when too many processes are running in the system.

This thesis proposes a dynamically adaptive parallelization model based on speculative multithreading for multiprogrammed shared-memory multiprocessor systems to address the problems associated with traditional parallelizing compilers. This work first addresses the problem of extracting loop-level parallelism from programs with run-time data-dependences and traditionally sequential loop constructs. A speculative multithreading parallelization model is proposed that uses a pipelined execution model with run-time data-dependence checking and control speculation to parallelize such loops. Next, a comprehensive dynamic processor allocation scheme is proposed to address the problem of executing parallel applications efficiently in a multiprogrammed environment. This processor allocation mechanism allows parallel programs to dynamically adapt to the

current execution environment by dynamically allocating an appropriate number of processors based on current program behavior and the system load. The efficiency of the proposed mechanisms is evaluated using experiments on several off-the-shelf shared-memory multiprocessor systems.

Contents

1 Introduction **1**

1.1 Loop-Level Parallelism 2

1.1.1 Data Dependence in Loop Iterations 2

1.1.2 Loops with Unknown Iteration Space 5

1.2 Efficient Parallel Execution 7

1.2.1 Parallel Execution Overhead 7

1.2.2 Parallel Execution in Multiprogrammed Systems 9

1.3 Problem Statement 10

1.4 Thesis Contributions 11

1.5 Thesis Organization 14

2 Related Work **15**

2.1 Run-Time Parallelization Schemes 16

2.1.1 Software-Based Run-Time Parallelization Schemes 17

2.1.2 Hardware Based Speculative Run-Time Parallelization Schemes 27

2.1.3 Multithreaded Processors 34

2.1.4 Speculative Multithreading Versus Existing Run-Time Parallelization Schemes 35

2.2 Dynamic Processor Allocation Schemes 36

2.2.1 Dynamic Serialization 37

2.2.2 Adaptive Parallelism in SUIF Compiler-generated Code 38
2.2.3 Loop-Level Process Control 39
2.2.4 Automatic Self-Allocating Threads 40
2.2.5 Comprehensive Dynamic Processor Allocation Scheme Versus Existing Processor Allocation Schemes 41
2.3 Techniques for Improving Java Performance 42
2.3.1 Java Parallelization Techniques 42
2.3.2 Dynamic Compilation 43
2.4 Conclusion 45

3 Speculative Multithreading Parallelization Model 46
3.1 Superthreaded Architecture 47
3.1.1 Thread Partitioning 47
3.1.2 Thread Pipelining Execution Model 49
3.2 Thread pipelining execution model in Speculative Multithreading 52
3.2.1 Mapping the Thread Pipeline Stages to Software 53
3.3 Execution-Time Overhead of the Speculative Multithreading Model 59
3.3.1 Parallelization Overhead 59
3.3.2 Misspeculation Overhead 61
3.4 The Speculative Multithreading Library 62
3.4.1 C Implementation 62
3.4.2 Java Speculative Multithreading Library 67
3.5 Parallelization in the Context of Java Language Features 73
3.5.1 Exception Handling 73
3.5.2 Polymorphism 76
3.5.3 Garbage Collection 76

3.6 Conclusion . . . 76

4 A Comprehensive Dynamic Processor Allocation Scheme 78

4.1 Dynamic Processor Allocation Scheme . . . 80

4.1.1 Multiple Version Code . . . 81

4.1.2 Parallel Execution Decision Heuristic . . . 82

4.1.3 Estimating the Sequential Execution Time . . . 83

4.1.4 Determination of Parallelization Overhead . . . 86

4.1.5 System Load Determination . . . 86

4.2 Run-time System . . . 87

4.2.1 Multiple Version Code Generation . . . 88

4.2.2 Parallelization Overhead of the Speculative Multithreading Model . . . 88

4.2.3 System Load Determination . . . 89

4.2.4 Dynamic Processor Allocation Algorithm . . . 91

4.3 Library Routines for Dynamic Processor Allocation . . . 96

4.3.1 C library . . . 96

4.3.2 Java Library . . . 99

4.4 Conclusion . . . 102

5 Performance Evaluation 103

5.1 Test System . . . 104

5.2 Test Programs . . . 105

5.3 Performance Evaluation of the Speculative Multithreading Parallelization Model . . 107

5.3.1 Parallelization Overhead on the Test System . . . 107

5.3.2 Performance With Application Programs . . . 110

5.3.3 Performance Comparison . . . 119

5.4 Performance Evaluation of the Comprehensive Dynamic Processor Allocation Scheme 123

5.4.1 Dynamic Processor Allocation Overhead 124

5.4.2 Dynamically Adapting to a Program's Characteristics 126

5.4.3 Dynamic Processor Allocation versus Time-Sharing in a Multiprogrammed System 130

5.4.4 Effect of Load Sampling Interval on the Performance of the Dynamic Processor Allocation Scheme 146

5.4.5 Comparison of Different Parallelization Heuristics for *Do-While* Loops 152

5.5 Conclusion 159

6 Conclusion 161

6.1 Thesis Contributions 162

6.2 Future Work 164

6.3 Summary 167

List of Figures

1.1 Data dependences across loop iterations. 3

1.2 An example loop with a potential cross-iteration dependence. 4

1.3 An example *for* loop with known iteration space. 5

1.4 Loops with unknown iteration space: (a) *for* loop with early exit; (b) *do-while* loop. 6

1.5 An example *while* loop where parallel execution (a) will not overshoot; (b) may overshoot. 7

1.6 Parallel execution using *fork-join* model. 8

2.1 An example loop with a run-time data dependence. 18

3.1 Basic organization of a superthreaded processor. 48

3.2 Pipelined execution of concurrent threads in a superthreaded processor. 49

3.3 Pipelined execution of threads in the speculative multithreading parallelization model. 54

3.4 Some of the library call overhead can be overlapped in a parallel loop due to the pipelined execution of threads. 60

3.5 An example sequential code segment from the Unix utility program *wc*. 65

3.6 The code from the example in Figure 3.5 parallelized using the speculative multithreading library routines. 66

3.7 An example sequential Java code segment to be parallelized. 70

3.8 The `ParLoop_1` class corresponding to `Loop_1` in `Method_1` shown in Figure 3.7. . . . 70

3.9 The code that replaces the original sequential code in the example in Figure 3.7 to initiate multiple Java threads for parallel execution. 70

3.10 An example sequential code segment from the Java version of the Unix utility program *wc*. 71

3.11 The code from the example in Figure 3.10 parallelized using the JavaSpMT library routines. 72

3.12 Exception handling in Java. 73

3.13 Modifications to parallel Java code to handle exceptions. 75

4.1 An overview of the dynamic processor allocation system. 81

4.2 Dynamic processor allocation algorithm. 92

4.3 Program instrumentation required to incorporate the dynamic processor allocation mechanism in a C code. 98

4.4 Program instrumentation required in a Java code to enable dynamic processor allocation. 101

5.1 Speedups of the *Matrix Multiplication* program, parallelized using the C implementation of the speculative multithreading model on 2,4, and 8 processors of the SGI Challenge multiprocessor system : (a) *speedup excluding the effect* of the thread creation overhead; (b) *speedup including the effect* of the thread creation overhead. . . . 111

5.2 Speedups of the JavaSpMT version of the *Matrix Multiplication* program on 2,4, and 8 processors of the SGI Challenge multiprocessor system. 112

5.3 Speedups of the *Gaussian Elimination* program parallelized using the C implementation of the speculative multithreading model. 113

5.4 Speedups of the Java version of the *Gaussian Elimination* program parallelized using JavaSpMT model. 114

5.5 Speedups of the Perfect Club benchmark *MDG* as the number of processors is varied in (a) the C implementation; (b) the Java implementation. 115

5.6 Speedups of (a) the C version of *wc* and (b) the Java version of *wc*. 116

5.7 Speedup of the *simulate* loop in 183.equake on 2 to 8 processors of the SGI Challenge multiprocessor system using (a) the C implementation and (b) the Java implementation of the speculative multithreading model. 117

5.8 Speedup of the *simulate* loop in 183.equake on 2 to 6 processors of two different shared-memory multiprocessor systems: (a) C implementation; (b) Java implementation. . 118

5.9 Comparison between the C and the Java implementations of the speculative multi-threading parallelization model. 120

5.10 Comparison of the C implementation of the speculative multithreading model and the Privatizing DOALL Test run-time scheme. 122

5.11 Comparison between the JAVAR and the JavaSpMT execution techniques. 124

5.12 Execution time of the Java version of *gauss* (N=100) on 1, 3, and 6 dedicated processors of the Sun SparcServer 1000 system. 127

5.13 Slowdown factors of the C application programs when using time-sharing and dynamic processor-allocation with one sequential application running. 132

5.14 Slowdown factors of the C application programs when using time-sharing and dynamic processor-allocation with two concurrently executing parallel C applications. 134

5.14 (Contd..) Slowdown factors of the C application programs when using time-sharing and dynamic processor-allocation with two concurrently executing parallel C applications. 135

5.15 Slowdown factors of the C application programs when using time-sharing and dynamic processor-allocation with three concurrently executing parallel C applications. 137

5.15 (Contd..) Slowdown factors of the C application programs when using time-sharing and dynamic processor-allocation with three concurrently executing parallel C applications. 138

5.16 Slowdown factors of the C application programs when using time-sharing and dynamic processor-allocation with four concurrently executing parallel C applications. 139
5.17 Slowdown factors of the C application programs when using time-sharing and dynamic processor-allocation with five concurrently executing parallel C applications. 140
5.18 Slowdown factors of the Java application programs when using time-sharing and dynamic processor-allocation with one sequential application running. 142
5.19 Slowdown factors of the Java application programs when using time-sharing and dynamic processor-allocation with two concurrently executing parallel Java applications. 144
5.20 Slowdown factors of the Java application programs when using time-sharing and dynamic processor-allocation with three concurrently executing parallel Java applications. 145
5.21 Slowdown factors of the Java application programs when using time-sharing and dynamic processor-allocation with four concurrently executing parallel Java applications. 145
5.22 Synthetic Java program to evaluate the effect of the load sampling interval on parallel execution performance. 146
5.23 Effect of load sampling interval on parallel execution performance of a *for* loop with a normalized workload of 112x. 149
5.24 Effect of load sampling interval on parallel execution performance of a *for* loop with a normalized workload of 223x. 151
5.25 Effect of load sampling interval on parallel execution performance of a *while* loop with a normalized workload of 107x. 153
5.26 Effect of load sampling interval on parallel execution performance of a *while* loop with a normalized workload of 209x. 154
5.27 Synthetic loop to compare the performance of different parallelization heuristics for *do-while* loops. 156
5.28 Comparison of different parallelization heuristics in the C implementation of the dynamic processor allocation scheme. 157

5.29 Comparison of different parallelization heuristics in the Java implementation of the dynamic processor allocation scheme. 157
5.30 Comparison of different parallelization heuristics in the C implementation with alternating parallel and sequential loop workload. 158
5.31 Comparison of different parallelization heuristics in the Java implementation with alternating parallel and sequential loop workload. 159

List of Tables

3.1 Differences in the pipeline stage implementations in C and Java. 59

5.1 Thread creation overhead (in seconds) for 2, 4, and 8 processors on the SGI Challenge system. 107

5.2 Overhead (in microseconds) associated with the different pipeline stages of the C implementation of the speculative multithreading model as measured on the test system. 108

5.3 Parallelization overhead of the JavaSpMT model (in milliseconds) on the SGI Challenge System. 109

5.4 Dynamic processor allocation overhead (in milliseconds) on the 6-processor Sun SparcServer multiprocessor system. 125

5.5 Performance of C applications with a dynamically changing program workload. . . 128

5.6 Performance of Java applications with dynamically changing program workload. . . 129

5.7 Characteristics of C programs used in measuring slowdown factors in the presence of system load. 132

5.8 Characteristics of Java programs used in measuring slowdown factors in the presence of system load. 142

Chapter 1

Introduction

The need for higher execution-time performance for application programs has lead to better processor and compiler technology. One way to execute programs faster is to use a faster microprocessor. With the rate of improvement in semiconductor technology slowing down, however, uniprocessor performance will saturate at some point. An alternative way to improve performance is to exploit the parallelism inherent in application programs. Different processor architectures exploit the parallelism at different granularities [29]. Superscalar [21] and very-long instruction word (VLIW) [33] processors, for example, exploit fine-grained instruction-level parallelism (ILP) in programs using such techniques as multiple instruction issue, static and/or dynamic scheduling, register renaming, and speculative execution. While these techniques can provide better performance than a single processor machine, the amount of parallelism they can exploit is limited since they can only exploit parallelism from a single thread of execution [28].

Another alternative is to exploit coarse-grained loop-level parallelism on shared-memory multiprocessors [29]. Since loops usually execute many times in a program, they are likely to provide a large amount of parallelism to be exploited. The loop is partitioned so that each processor executes a different iteration (or a number of iterations) concurrently with the other processors. Parallelizing compilers are available that can identify and extract the loop-level parallelism from sequential

programs [6, 18]. Compile-time analysis, however, has its limitations. For example, parallelizing compilers cannot successfully determine data dependences in loops where data access patterns depend on run-time (or input) values. The limitations in compile-time information, thus, often makes it difficult for compilers to extract much of the available loop-level parallelism, especially in general-purpose programs.

1.1 Loop-Level Parallelism

Parallelizing compilers perform data-dependence analysis on loops to determine the data-dependence relations across loop iterations. Depending on the compile-time analysis, the loop can be executed in parallel as a DOALL or a DOACROSS loop. If the compiler can not determine the data-dependence relations, the loop is executed sequentially. The compiler also needs to know the iteration space of the loop to statically partition the loop among the processors. In the following, we discuss the type of data dependences that may occur in a loop and how the dependence pattern may affect loop parallelization. We also discuss different loop constructs and how they are handled by traditional parallelizing compilers.

1.1.1 Data Dependence in Loop Iterations

To exploit loop-level parallelism, it is necessary to identify any potential data dependence across the loop iterations. There are three types of data dependence, *flow dependence*, *anti-dependence*, and *output dependence*, caused by data accesses in loops [1].

Flow dependence

If a read access to a memory location is followed by a write access to the same location, the dependence between these two accesses is called a *flow dependence* or read-after-write (RAW) hazard. In this case, the read operation is dependent on the previous write operation since it has to wait for the write to complete in order to read the correct value.

```
        for(i=0; i<N; i++){

S1          A(i) = B(i+1) + C(i);
S2          B(i) = A(i-1) + D(i);

        }
```

Figure 1.1: Data dependences across loop iterations.

Anti-dependence

Anti-dependence or write-after-read (WAR) hazard is caused by a write access followed by a read access to the same location. The dependence is due to the fact that, if the write access is executed before or in parallel with the read access, the read will not return the correct value.

Output dependence

If a write access to a memory location is followed by another write access to the same location, the dependence among the accesses is called an *output dependence* or write-after-write (WAW) hazard. In this case, if the second write is performed out of order, its update will be lost.

Example

We use the loop examples in Figures 1.1 and 1.2 to demonstrate data dependences between loop iterations. There are two assignment statements, *S1* and *S2* in the example in Figure 1.1. In *S1*, the value of $A(i)$ is dependent on the read value of $B(i+1)$, which is written in the next iteration by *S2*. Similarly, in *S2*, value of $B(i)$ depends on the read value of $A(i-1)$, which is written in the previous iteration by *S1*. Thus *S2* is *flow dependent* on *S1* across different iterations because of array A. *S2* is also *anti-dependent* on *S1* due to the access to array B across iterations. In the example of Figure 1.2, if, for example, $B(1) = B(2) = x$, then statement *S1* writes to $A(x)$ in both iteration 1 and iteration 2. Thus, this would produce an *output dependence* between statements *S1* in iteration 1 and iteration 2.

```
        for(i=0; i<N; i++){
            ...
S1          A(B(i)) = ...
            ...
S2          ... = A(C(i)) + ...
            ...
        }
```

Figure 1.2: An example loop with a potential cross-iteration dependence.

1.1.1.1 DOALL and DOACROSS Loops

A loop can be fully parallelized provided that there are no data dependences among its loop iterations. Such a loop is called a DOALL loop. In a DOALL loop, iterations can be executed in any order and thus, in parallel. In the example in Figure 1.2, if the subscript arrays B and C contain unique values for each element, there is no dependence between statements *S1* and *S2*. Hence, we can call this loop a DOALL loop. If a loop, on the other hand, has some dependence relations, i.e., *flow, anti-,* or *output dependences* across loop iterations, then the loop iterations must be executed in an order that enforces the data dependences.

Loops with cross-iteration dependences are called DOACROSS loops. Such loops cannot be fully parallelized. Iterations can be partially overlapped using some synchronization scheme to enforce data dependences, so that dependent iterations do not execute out of order. In the example in Figure 1.2, if the arrays B and C have some common values, there will be data dependences between *S1* and *S2* across iterations. Thus, in this case, the loop will be a DOACROSS loop.

1.1.1.2 Run-time Data Dependence

In loops such as in Figure 1.2, sometimes it is not possible to determine the dependence relations among loop iterations at compile time. For example, if the index arrays B and C are initialized with inputs to the program during runtime, it is not possible for the compiler to statically determine the dependence relations among iterations. Hence, even if the loop turns out to be fully or mostly

```
for(i=0; i<N; i++){
    ...
    ...
    ...
}
```

Figure 1.3: An example *for* loop with known iteration space.

parallel, compile-time parallelization schemes will fail to parallelize the loop. Loops with data accesses through complex index calculations and pointers also cannot be analyzed for data dependence at compile-time. For such loops, the data-dependence test needs to be performed at run-time to determine whether the loop iterations can be executed in parallel. This parallelization approach is called *run-time parallelization*.

1.1.2 Loops with Unknown Iteration Space

The iteration space of a loop, i.e., the number of times the loop is actually executed, may or may not be known at compile time. For example, a *for* loop with a pre-defined iteration bound has a known iteration space. In the example loop shown in Figure 1.3, the loop iteration count, i, varies from 0 to N. Thus, the loop has N iterations and the value of N is always available before the loop begins execution. Traditional parallelizing compilers can easily parallelize such loops with known iteration space provided the data-dependence relations can be analyzed. The compiler can divide the N loop iterations (for any given value of N) among the processors for parallel execution.

Loops with unknown iteration spaces are difficult to parallelize, however. If the loop termination conditions are determined at run-time as the loop executes, the iteration space is not known when the loop begins execution. *For* loops with early exits and *do-while* loops are examples of loops with unknown iteration spaces. The *for* loop in Figure 1.4 (a) has a conditional statement in the loop body which may force the loop to terminate early if the *condition* is satisfied. Such a loop is referred to as a *for* loop with an early exit. Depending on when the condition is satisfied, the loop

```
for(i=0; i<N; i++){
    ....
    ....
    ....
    if(condition) break;
}
```

(a)

```
while(not condition){
      ....
      ....
      ....
}
```

(b)

Figure 1.4: Loops with unknown iteration space: (a) *for* loop with early exit; (b) *do-while* loop.

may go through any number of iterations between 1 and N. Thus, the actual number of times the loop is executed is not known when the loop begins execution. Similarly, in a *do-while* loop (shown in Figure 1.4(b)), the loop termination condition determines the actual number of loop iterations executed. Since the number of iterations in these type of loops is non-deterministic, the compiler can not divide the iterations among the processors for parallel execution.

Furthermore, if the loop termination conditions can not be evaluated independently by all iterations, parallel execution may result in execution of the loop beyond the point where the original sequential loop would have terminated. For example, the loop shown in Figure 1.5(a) accesses a linked list and terminates when the list is empty (the list pointer, *tmp*, becomes NULL). Since any iteration beyond the last element will find the pointer NULL, the iterations cannot execute beyond the last element.

The loop in Figure 1.5(b), however, may result in execution beyond the last valid sequential iteration, when executed in parallel. One of the loop termination conditions is a conditional statement $f(i) < V$ where $f(i)$ is a function of the loop iteration count variable i. Each iteration i will evaluate $f(i)$ independently of the others. While the condition may be satisfied in iteration $i = k$, other concurrently executing iterations greater than k may continue executing even though the condition is true. Any parallelization technique handling these loops must undo the effects of the later iterations that overshot (i.e., executed beyond the last valid iteration).

Due to the difficulties with unknown iteration spaces and with the recovery process for iterations

```
tmp = head(list);
while(tmp != NULL){
      ....
      ....
      tmp = next(tmp);
}

        (a)
```

```
i = 0;
while(f(i) < V && i < N){
      ....
      ....
      i = i+1;
}

        (b)
```

Figure 1.5: An example *while* loop where parallel execution (a) will not overshoot; (b) may overshoot.

that overshot, traditional parallelizing compilers treat *do-while* loops and *for* loops with early exits as sequential loop constructs. General-purpose application programs often contain these traditionally sequential loop constructs with a large amount of inherent parallelism. However, due to the inability of existing parallelizing compilers to extract the parallelism in such loops, general-purpose programs often cannot be parallelized for better execution-time performance.

1.2 Efficient Parallel Execution

Identifying and extracting the parallelism inherent in sequential application programs is only a first step in improving performance. To achieve the best performance, the transformed programs need to be efficiently executed on a given shared-memory multiprocessor system. Efficient execution of parallel programs depends on a variety of parameters including the program workload, target system configuration, and the system load.

1.2.1 Parallel Execution Overhead

Parallel execution of application programs on a shared-memory multiprocessor system usually is based on a *fork-join* programming model [2]. A parallel program typically has interleaving sequential and parallel code regions. At the start of a parallel region, a number of child processes are spawned (or forked) on different processors to concurrently execute the code. At the end of the parallel section, these child processes are synchronized (or joined) and the program begins its next sequential code

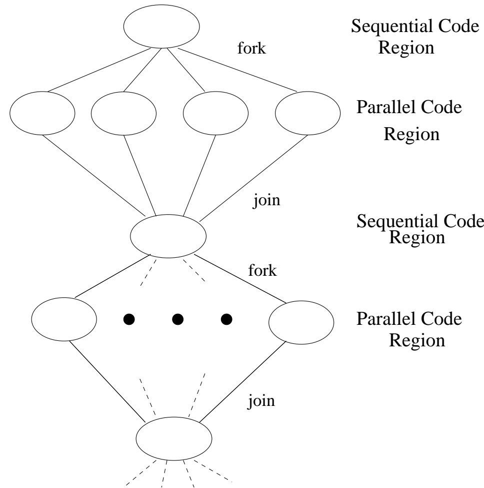

Figure 1.6: Parallel execution using *fork-join* model.

section. This *fork-join* execution model is shown in Figure 1.6. The processes executing the parallel code region may use synchronization operations to access shared variables to maintain the correct order of program execution.

Parallel execution is associated with certain overheads that include the cost of process creation and synchronization in the *fork-join* model, as well as the cost of some additional operations that may be needed for the parallel execution but are not part of the original sequential code. Given a sequential program with a single processor execution time, $T_s = T_{s1} + T_{s2}$, the corresponding parallel execution time of the program on P-processors of the same system will be,

$T_p = T_{s1} + \frac{T_{s2}}{P} + T_{oh}$

where T_{s1} is the time required to execute the sequential code regions of the program, T_{s2} is the time required by the parallel code regions when executed sequentially, and T_{oh} is the overhead associated with the parallel execution on P processors.

Parallel execution results in a performance speedup only if the parallel code region (e.g., a loop) can amortize the associated overhead cost, i.e., $T_{s2} < \frac{T_{s2}}{P} + T_{oh}$. If the workload of a loop is too small, its parallel execution will be dominated by the overhead and, thus, will be slower than the corresponding sequential execution. Thus, the degree of parallelism in a parallel code region and the

actual overhead on the target system will determine if parallel execution will result in performance improvement. Consequently, using a static mapping of parallel code regions to a fixed number of processors may not produce the best performance. For programs with dynamically varying behavior, such as programs where the number of loop iterations changes dynamically across invocations, it is not possible to determine at compile-time the number of processors that can be used most efficiently at each invocation. Furthermore, in the case of portable programs, the target machine configuration (e.g., information about number of processors available and parallelization overhead) may not be known at compile time. Without the knowledge of the target machine, the compiler can not make any decision about effective parallelization even when the program behavior remains unchanged during execution.

1.2.2 Parallel Execution in Multiprogrammed Systems

In a standard multiprogrammed environment, application programs are executed using a time-shared approach. While time-sharing can improve overall system utilization and can allocate each process a fair share of the systems' resources, this approach can lead to significant performance degradation for parallel applications when executed simultaneously with other sequential or parallel applications. A number of studies [16, 32, 35, 47] have shown that parallel programs often do not perform well in multiprogrammed environments due to the following reasons:

- **Context switching.** If there are more processes in the system than the actual number of physical processors, the execution of the processes will be multiplexed using the time-shared approach. When the *time quantum* of an active process expires, the operating system scheduler suspends the process and puts it at the end of the job queue. The corresponding processor then executes the process at the beginning of the queue. This swapping of the processes is referred to as *context switching* and adds overhead to the execution time of a process.

- **Poor cache utilization.** When a process is swapped out by the scheduler due to context switching, the processor's cache is overwritten by the next process. When the former process is scheduled again for execution, it must start with an empty cache and will not be able to take advantage of the previous working data set.

- **Locking overhead.** Parallel code regions often use synchronization operations to control access to shared variables to maintain correct program state. In a multiprogrammed environment, if a process holding a synchronization lock is swapped out, another active process which is waiting to acquire the lock will not be able to proceed. This waiting process will busy-wait on the lock wasting the system's processing resource.

The performance degradation in a standard multiprogrammed system due to time-sharing can be avoided if a given number of processors can be dedicated to each parallel application in the system so as not to overload the system. However, since the system load changes dynamically, the number of processors that can be dedicated to a parallel application in a given time period also changes. Compiler parallelized application program uses a pre-defined number of processors and, thus, cannot guarantee effective parallel execution performance in a multiprogrammed environment.

1.3 Problem Statement

Parallel execution of application programs may lead to significant performance improvement over sequential execution. Parallelizing compilers are available that can automatically identify loop-level parallelism in sequential programs and translate the sequential code to corresponding parallel code for shared-memory multiprocessors systems [6, 18]. Traditional parallelizing compilers, however, cannot often extract much of the loop-level parallelism in application programs due to the existence of run-time data dependences or unknown iteration spaces. The parallelism in such loops can only be exploited using run-time parallelization techniques since the information required to parallelize these loops are only available at run-time. Prior studies have shown that a large fraction (about

50%) of all scientific applications have run-time data dependences [40]. Furthermore, most general-purpose applications have irregular loop structures, i.e., loops with unknown iteration spaces. Thus, a run-time parallelization technique is needed to exploit the loop-level parallelism in these programs that represent a large fraction of the application domain.

Once a program has been parallelized, it is necessary to efficiently execute the program in parallel to obtain performance improvement. The compiler often cannot predetermine the number of processors a parallel code region can most efficiently use in cases where the program behavior changes dynamically or the system load changes (in case of a multiprogrammed environment) as the program executes. Furthermore, when dealing with portable parallel programs, such as parallel Java programs, the compiler cannot take advantage of prior knowledge of the target machine to predetermine the number of processors the parallel code can use efficiently. In these situations, it is necessary that the processors for a parallel code region be dynamically allocated at run-time when complete information about program workload, machine configuration, and system load are available.

Thus, to improve performance of general-purpose application programs with inherent loop-level parallelism, compile-time analysis must be complemented with run-time techniques that can efficiently exploit the parallelism and execute the transformed parallel code using complete information about the program and the target system.

1.4 Thesis Contributions

This thesis addresses issues related to the efficient parallel execution of application programs on shared-memory multiprocessor systems that can not be handled by compiler parallelized programs. First, it addresses the problem of extracting parallelism from sequential programs that contain loops with run-time data dependences and/or loops with unknown iteration spaces. Second, it addresses the problem of dynamic adaptation of parallel programs so that the number of processors

allocated to a program provides the best performance under varying execution environment. The main contributions of this thesis are:

- A new run-time parallelization model, called *speculative multithreading*, is proposed to exploit coarse-grained loop-level parallelism from general-purpose application programs on shared-memory multiprocessor systems [23, 24]. The speculative multithreading parallelization model, which is based on the execution model of the superthreaded processor architecture [44, 45, 46], uses a thread pipelined execution model with run-time data dependence checking and control speculation to parallelize loops with potential data dependences that cannot be analyzed at compile-time as well as loops with traditionally sequential constructs. The run-time dependence test in this parallelization model allows loops with a wide variety of data structures, including lists and pointer structures, to be easily parallelized. The control speculation allows loops with non-deterministic termination conditions, and loops with complex branch structures (e.g. nested *if-then-else*), to be executed speculatively. This speculation results in better performance than sequential execution if the speculation is usually correct.

- For efficient parallel execution of application programs on a multiprogrammed multiprocessor system, a *comprehensive dynamic processor allocation scheme* is proposed to dynamically change the number of processors allocated to different parallel code regions of a program as the program executes [25]. This dynamic processor allocation scheme uses run-time information about program workload, target machine configuration, and current system load to allocate an appropriate number of processors to an application program. The number of processors allocated is guaranteed to maximize the parallel execution performance without overloading the system in the presence of other processes executing in the system.

  Using this dynamic processor allocation mechanism, a parallel application program is allocated as many processors as are currently available, up to the maximum number it can efficiently utilize. If the program's characteristics are such that the code region cannot amortize the

associated overhead, or if the system is overloaded, the code region is dynamically serialized. The processor allocation decision is dynamically updated as both the program behavior and the system load change during a program's execution.

- The proposed speculative multithreading parallelization model targets general-purpose application programs. This parallelization model was implemented in C, which is a widely-used general-purpose programming language. In addition, we considered Java as another language for the implementation of the parallelization model. Java with its portability, object-oriented model, multithreading, distributed programming, and automatic garbage collection features, is gradually becoming the language and platform of choice for general-purpose application programs [26]. Furthermore, Java's platform-independent support for multithreading allows us to develop portable parallel programs that can potentially execute on a variety of shared-memory multiprocessor systems supporting the Java Virtual Machine(JVM).

  We applied the comprehensive dynamic processor allocation scheme to application programs parallelized using the speculative multithreading model. Consequently, we have both C and Java implementations of the dynamic processor allocation mechanism to dynamically control the parallel execution of application programs.

- Finally, the performance of the speculative multithreading parallelization model is evaluated by applying it to a number of application programs. The application-level performance of this parallelization model is compared to the performance of some existing parallelization techniques.

  The effectiveness of the comprehensive dynamic processor allocation scheme is studied in detail under varying program workload and varying system load conditions. The performance of parallel applications using this processor allocation mechanism is further studied under varying degrees of system load. The results are compared with that of a time-shared approach.

1.5 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 describes some related work on run-time parallelization and dynamic processor allocation schemes as well as some parallelization and dynamic techniques specific to Java execution. Chapter 3 presents the proposed run-time parallelization model, *speculative multithreading*, with detailed discussion about the implementation of the model in C and Java. The *comprehensive dynamic processor allocation* scheme is presented in Chapter 4 discussing specific C and Java implementation issues. Chapter 5 evaluates the performance of these proposed techniques on shared-memory multiprocessor systems. Finally, the thesis contributions are summarized in chapter 6.

Chapter 2

Related Work

A variety of existing run-time parallelization techniques, both software [11, 37, 38, 39, 54] and hardware-based [34, 43, 53], can exploit coarse-grained loop-level parallelism from application programs that cannot be easily parallelized using traditional parallelization tools. Most of the software-based techniques can handle loops with run-time data dependence only. Other techniques can parallelize loops with run-time dependence as well as loops with unknown iteration spaces (e.g. *do-while* loops or *for* loops with early exits). These techniques, although efficient, require extensive hardware support for speculative execution of loop iterations. The speculative multithreading parallelization model proposed in this thesis extends these existing techniques, providing a unique software-only approach for the speculative execution of loop iterations with run-time data-dependence tests. We discuss some of this previous work in Section 2.1.

The efficient execution of parallel application programs on multiprocessor systems also has been addressed in a number of dynamic processor allocation schemes [17, 41, 48, 49, 50, 51, 52]. Some of these techniques consider dynamic program behavior in allocating an appropriate number of processors to the application program [48, 49, 50]. Some other techniques allocate processors based on system load for efficient execution in multiprogrammed environment [17, 41, 51, 52]. The comprehensive dynamic processor allocation scheme proposed in this thesis is a new approach in processor

allocation for the efficient execution of parallel application programs on shared-memory architecture using dynamic program behavior as well as system load information. The existing processor allocation techniques are discussed in Section 2.2 to show how the approach proposed in this thesis differs from these techniques.

The speculative multithreading run-time parallelization model and the dynamic processor allocation scheme are applied to application programs written in the C and the Java programming languages. The unique language features of Java, such as portability and support for multithreading, make parallelization simpler than compiled languages in that once a program is parallelized it can potentially execute on a variety of hardware platforms. A variety of parallelization tools have been developed specifically for Java programs using Java multithreading support [4, 5, 19, 20, 31]. Some dynamically adaptive execution techniques, though not in the context of parallel execution, have also been developed for Java programs [7, 15]. We briefly discuss these techniques in Section 2.3.

2.1 Run-Time Parallelization Schemes

Traditional parallelization techniques rely on compile-time data-dependence analysis to identify loop-level parallelism. These techniques often cannot extract much of the loop-level parallelism in application programs due to the presence of run-time data dependences. Traditionally sequential loop constructs, such as *do-while* loops and *for* loops with early exits, also cannot be parallelized by these techniques since the loop bound is not known at compile-time. A number of schemes have been developed to exploit medium to coarse-grained loop-level parallelism in programs in which the parallelism cannot be detected at compile-time [11, 34, 37, 38, 39, 43, 53, 54]. These schemes are generally referred to as run-time parallelization schemes.

Most of the run-time parallelization techniques are based on *inspector-executor* algorithms [11, 37, 38, 54]. These techniques, most of which are implemented entirely in software, consist of an *inspector* stage followed by an *executor* stage. The *inspector* determines the dependence relations

among data accesses across loop iterations that actually exist at run-time. The *executor* uses the information provided by the *inspector* to then execute the iterations in parallel in an order that preserves the dependence relations. These *inspector-executor* based schemes generally cannot handle the parallelization of *do-while* type loops.

A number of techniques have been proposed that require hardware support for run-time parallelization [34, 43, 53]. In general, these techniques speculatively execute possibly dependent loop iterations in parallel relying on the cache-coherence protocol hardware to detect any data-dependence violations. Additional support from the hardware is needed to abort and restart loop iterations that are misspeculated. Because of speculative execution, these techniques can potentially handle loops with unknown iteration spaces.

In subsequent sections, we describe different software and hardware-based run-time parallelization techniques that exploit loop-level parallelism from application programs for parallel execution on shared-memory multiprocessor systems.

2.1.1 Software-Based Run-Time Parallelization Schemes

In the following, we describe the different software-based parallelization techniques that have been proposed for the parallelization of loops with run-time dependence relations and loops with traditionally sequential constructs. The algorithms proposed differ in the types of dependence patterns handled and the required system or architectural support.

In discussing some of the run-time parallelization techniques, we will use the loop shown in Figure 2.1 as an example. In this loop, the data array A is accessed through the elements of arrays B and C. Arrays B and C are referred to as index arrays. If the elements of these index arrays have some common values for $i = 1$ to N, there will be data dependences between statements *S1* and *S2* across iterations. The values for the array elements may not be known at compile time, however. For example, if the index arrays B and C are initialized with inputs to the program, or are calculated during runtime, it is not possible for the compiler to statically determine the dependence

```
        for(i=0; i<N; i++){
            ...
S1          A(B(i)) = ...
            ...
S2          ... = A(C(i)) + ...
            ...
        }
```

Figure 2.1: An example loop with a run-time data dependence.

relations among iterations. Hence, even if the loop turns out to be fully or mostly parallel, compile-time parallelization schemes will fail to parallelize the loop. Run-time parallelization techniques are needed to parallelize such loops.

2.1.1.1 The Zhu-Yew Algorithm

The first run-time parallelization scheme proposed by Zhu and Yew [54] can handle any type of loop-carried dependence pattern in array-based data structures. Array variables that require data dependence enforcement are transformed by the compiler into a special data structure. Each element of such an array A has two fields - the *data* field that contains the actual data value of the element, and, the *key* field which is used to order accesses to the elements by different iterations. Each loop execution consists of several *inspector-executor* passes. During each pass, the *inspector* first determines the set of iterations that can be executed in parallel. The *executor* then executes these iterations in parallel. The *inspector-executor* passes continue until all iterations in the loop are executed.

Inspector

In a loop such as that shown in Figure 2.1, this scheme allows iteration i to proceed at runtime only if all accesses to the array elements *A(B(i))* and *A(C(i))* have completed for all iterations for which $j < i$. During the *inspector* stage, all unexecuted iterations check the *key* fields of the array

elements they need to access. The iterations store their own iteration numbers in the *key* field of these array elements, if it is less than the value already stored in the *key* field. After the *inspector* phase, the numbers remaining in the *key* fields of the array elements are the numbers of the iterations that can be executed in parallel during the current pass.

Executor

In the *executor* phase, iterations check if the *key* field of the elements that they want to access have a number equal to their own iteration number. If so, the iteration has no unexecuted predecessor and so can proceed. Otherwise, the iteration must wait until the next pass when the *inspector* stage determines that this iteration can execute.

Both the *inspector* and the *executor* stages in Zhu-Yew's algorithm are fully parallel. The *inspector* stage consists of DOALL loops with synchronization performed by atomic *compare&swap* (if $A(B(i)).key > i$ then $A(B(i)).key = i$) instruction. The *executor* stage also consists of a DOALL loop. In this stage, synchronization among processors is enforced by the *key* fields of the array elements. However, the *inspector* and the *executor* are tightly coupled. Hence, the *inspector*'s results cannot be reused even if the dependence relations remain unchanged across different invocations of the loop, such as occurs when an inner loop of a nested loop structure is parallelized. Furthermore, this scheme requires a large number of memory accesses. In each iteration of the loop, each array reference that may cause data dependence requires at least three additional memory accesses. Two of the references are in the *inspector* phase to initialize the corresponding *key* field and to perform the atomic *compare&swap* operation on the *key* field. The other access is in the *executor* phase to check the *key* field to determine if the iteration can proceed during that pass. Another limitation of this algorithm is that it does not treat input dependences (read-after-read hazards) any differently than the other dependence patterns. Hence, consecutive reads to the same array element by different iterations are serialized in this scheme.

2.1.1.2 The CYT Algorithm

The CYT algorithm [11] is another *inspector-executor* based run-time parallelization scheme. This scheme has completely separate *inspector* and *executor* stages. Thus, the CYT scheme allows the *inspector's* results to be reused across invocations of the loop. It also allows a partial overlap of the execution of dependent iterations.

The *inspector* stage gathers and stores all the dependence information in a table called the *ticket* table. The *executor* uses this table, possibly in several different invocations of the loop, to execute iterations in parallel in an order that enforces all data dependences. In the following description of CYT algorithm, the array A in Figure 2.1 is called the *target array*. Arrays B and C are referred to as *index arrays*. All accesses to a single element of the *target array*, ordered by their serial execution order, form a *dependence chain*.

Inspector

The goal of the *inspector* stage is to store sequence numbers in the *ticket* table entries that order the references in each *dependence chain* from left to right. The X-dimension of the table is equal to the number of iterations in the loop, N_{it}, while the Y-dimension is equal to the number of accesses to the target array in each iteration, r. Building the *ticket* table requires interprocessor communication which reduces parallelism. Consequently, the *inspector* phase is partitioned into local and global phases to reduce the amount of interprocessor communication. The local phase is a communication-free phase during which each processor fills its portions of the *ticket* table with sequence numbers by performing a dependence analysis of the N_{it}/P consecutive iterations. Here P is the number of processors used.

Each processor allocates a working array with as many entries as *ticket* table entries it must fill. Each entry of the working array is initialized to the subscript value of the access to which it corresponds. Entries with identical values belong to the same *dependence chain*. Then the work-

ing array is sorted, moving entries with identical values to adjacent array locations. At this point, consecutive sequence numbers are assigned to the accesses in each *dependence chain*, except for the first access in each chain. Finally, the sequence numbers are copied to the *ticket* table. In the global phase, adjacent processors communicate with each other to fill the unassigned entries of the table with globally consistent numbers.

Executor

The *executor* phase actually executes the user's application code. All accesses to the target array are performed in parallel except for those that have dependences specified in the *ticket* table. To enforce these dependences, a *key* field is associated with each element of the target array. The *key* fields of all array elements are initialized to zero. At any instance during execution, the *key* field of an element contains the sequence number of the next access to be made to that particular element. During the *executor* stage, processors are assigned iterations for execution according to some scheduling policy. Before a processor attempts access j to the target array in iteration i, it must obtain the sequence number S from *ticket(i,j)*. The processor waits until *A(x).key* is equal to S. Once this is true, it performs the access and then updates *A(x).key* to the value expected by the next access in the *dependence chain*.

All iterations that find the $A(x)$.*key* value equal to the corresponding ticket table entry execute in parallel. In each iteration, each access to the shared array is synchronized with its own *key* value and the corresponding *ticket* table entry. Hence, if the iterations have multiple accesses to the shared array, an iteration can continue executing the access whose dependence has been resolved by previous iterations. The iterations can begin execution before dependences to all the accesses are resolved. Thus, execution of iterations can be partially overlapped as long as accesses in concurrent iterations are independent to further enhance performance.

2.1.1.3 The Privatizing DOALL Test

The Privatizing DOALL test [37] is a run-time parallelization technique that targets DOALL loops only. This algorithm can be applied in two modes. One is the *inspector-executor* mode where the *inspector* determines whether the loop is a DOALL loop. If it is, the *executor* then runs the loop iterations in parallel. The other mode is the *speculative* mode where both the *inspector*'s test and the loop iterations themselves are executed concurrently. If the *inspector* ultimately determines that the loop is not a DOALL loop, the entire loop is re-executed sequentially. If the loop is usually fully parallel, then the speculative mode of execution yields higher speedup, since the overhead of the *inspector* phase is overlapped with the execution of the loop iterations themselves. The *inspector* stage is fully parallel and requires no synchronization, and the *inspector*'s results can be reused across different invocations of the loop.

The DOALL test in the *inspector* stage is augmented by privatization of variables to eliminate dependences if possible, thereby increasing the parallelism available in the loop. A *private variable* is defined as a variable that can be accessed only by the loop iteration to which it belongs. The *privatization criterion* used in this scheme determines that a shared array A accessed in a loop can be privatized if every read access to an element of A is preceded by a write access to the same element of A within the same iteration.

Inspector

The DOALL test in the *inspector* stage has two phases - a *marking* phase followed by an *analysis* phase.

Marking Phase

For each shared array A, a read shadow array A_r, a write shadow array A_w, and a shadow array to flag array elements that cannot be privatized, A_{np}, are defined. All elements of the shadow arrays are initialized to zero. The following actions are performed in parallel for all accesses to the shared

array A for each iteration i of the loop.

- Writes: If this is the first write to this element in this iteration, then set the corresponding element in A_w.
- Reads: If this element is never written in this iteration, set the corresponding element in A_r. If this element has not been written in this iteration before this read access, set the corresponding element in A_{np}, i.e., mark it as not privatizable.
- Count the total number of write accesses to A that are marked in this iteration. Store the result in $tw_i(A)$, i.e., $tw_i(A) = \sum A_w[j]$, where $A_w[j]$ are elements marked in iteration i.

Analysis Phase

The analysis phase uses the arrays A_r, A_w, A_{np}, and $tw_i(A)$ to determine whether the loop in consideration is a DOALL loop. If the loop is a DOALL loop, then this phase also determines which shared variables are privatizable. The steps performed in this phase are:

- Compute $tw(A) = \sum tw_i(A)$, i.e., the total number of writes that were marked by all iterations in the loop, and $tm(A) = \sum A_w(i)$, i.e., the total number of marks in A_w.
- If any $A_w(i) \wedge A_r(i) = 1$, i.e., the marked areas are common anywhere, the loop is not a DOALL loop and the phase ends.
- If $tw(A) = tm(A)$, i.e., there are no output dependences, the loop is a DOALL loop and the phase ends.
- If any $A_w(i) \wedge A_{np}(i) = 1$, i.e., there is at least one dependence that cannot be removed by privatization, the loop is not a DOALL and the phase ends.
- Finally, if none of the above conditions hold true, the loop can be made into a DOALL loop by privatizing all elements in the shared array that are written in the loop.

Executor

The *executor* stage executes the loop iterations in parallel if the *inspector* stage determines that it is a DOALL loop. In this case, the elements marked by A_{np} are privatized before the actual execution of the iterations. However, if the *inspector* stage determines that the loop is not a DOALL loop, the loop will be executed sequentially.

As mentioned earlier, the Privatizing DOALL Test can be applied in speculative mode. In this mode, the *inspector* stage and the actual parallel execution of the loop iterations are carried out concurrently. If the loop is actually a DOALL loop, the overlap of the *inspector* and *executor* would minimize the performance impact of the overhead of the *inspector*. However, if the *inspector* eventually determines that the loop is not a DOALL loop, the parallel execution has to be terminated and all shared variables that have been updated must be restored using copies of the data variables saved before execution. Then the entire loop must be re-executed sequentially.

2.1.1.4 Run-Time Methods for Parallelizing Partially Parallel Loops

Rauchwerger, Amato and Padua proposed an algorithm [38] that performs run-time parallelization of DOACROSS loops using an extension of the Privatizing DOALL Test [37]. This algorithm increases the amount of parallelism in a loop by using array privatization and "reduction parallelization". Reduction parallelization refers to executing reduction operations, such as $x = x \oplus exp$, where $\oplus$ is an associative operator and exp does not occur in x, in parallel.

Inspector

As in the Privatizing DOALL algorithm, the *inspector* preprocesses the memory references to determine the data dependences. This dependence information can be used to identify array elements that are read-only, privatizable, or reduction variables. The *inspector* has two main tasks.

1. For each array element $A(x)$, the *inspector* collects the iteration number and the access type,

read or write, for all the references to $A(x)$ into an array R_x in iteration order.

2. For each array element *A(x)*, the *inspector* determines the data dependences between all its references and stores them in a data structure H_x for later use by the *scheduler* (or *executor*), as described below.

   Once R_x is available, the *inspector* computes H_x as follows. The relations between the references to *A(x)* can be organized (conceptually) into an array element dependence graph, D_x. If adjacent references in R_x have different access types, then a flow or anti-dependence exists. If adjacent accesses are both writes, then there is an output dependence. These dependences are reflected by a parent-child relationship in D_x. If adjacent references are both reads, then there is no dependence between the elements. They may have a common parent (or child) in D_x, however, which is the last write preceding (or first write following) them in R_x. A *level* field is added to each element in R_x. The reference's level in the dependence graph D_x is stored in the corresponding *level* field. Then, for each level, the index in R_x for the first reference at that level is stored in H_x. Thus, H_x is an array whose i^{th} element contains the index in R_x of the first reference at level *i*. H_x serves as a look-up table for the first reference in R_x at any level.

The *inspector* can be augmented to find array elements that are independent (accessed in only one iteration), read-only, privatizable, or reduction variables.

Executor/Scheduler

The task of the *scheduler* is to derive a valid parallel execution schedule for a loop using the dependence relation found by the *inspector*. In such a schedule, the iterations are partitioned into ordered subsets called *wave-fronts* such that all cross-iteration dependences go from an iteration in a lower numbered wavefront to an iteration in a higher numbered wavefront. The *executor* uses this schedule to execute the iterations in that order. The *scheduler* and the *executor* can be interleaved

so that iterations can be executed as soon as they are found to be ready.

2.1.1.5 Do-While Parallelization

A general framework to parallelize *do-while* loops is proposed in [39]. The authors define a *while* loop as a loop with one or more *recurrences* that can be detected at compile-time, a *remainder*, i.e., the loop body, and one or more *termination conditions*. The termination conditions (or terminators) can be a part of the recurrence (e.g., termination condition in a *while* loop), or they can occur in the remainder (e.g., conditional exit in a *for* loop). The dominating recurrence in the loop is called the *dispatcher*. Provided the remainder is parallel, this approach evaluates the recurrences in parallel, and speculatively executes the remainder concurrently. Later, the effects of the iterations that overshot the terminator (i.e., the iterations that would not have been executed in the original sequential execution) are undone. For loops with run-time data dependences, this approach combines the Privatizing DOALL test [37] with the following methods to speculatively execute the loop iterations in parallel.

Parallelizing the dispatcher

This framework uses different techniques to compute the loop dispatcher in parallel depending on the type of the recurrence. If the recurrence is an induction, such as $d(i) = c(i) \times i + b$, the dispatcher can be evaluated simultaneously by all processors. The loop is executed as a DOALL and the test for the loop termination is inserted in the loop body. During execution, each processor keeps track of the lowest iteration it executed that satisfied the termination condition. The minimum of these processor-wise minima is the last valid iteration in the corresponding sequential execution.

If the dispatcher is an associative recurrence, such as $x(i) = a \times x(i-1) + b$, it can be computed in parallel using the parallel prefix computation algorithms [30]. In this case, the original loop is transformed into two DOALL loops – one that evaluates the dispatcher only and the other that

executes the loop iterations using the values of the dispatcher computed in the first loop. The last valid iteration of the loop can be found by tracking the minimum iteration meeting the termination condition in the second loop.

For all other recurrences which are inherently sequential, the authors proposed several alternative methods to determine the dispatcher. In the first method, the processors calculate the dispatcher in a pipelined manner thereby dynamically assigning iterations to processors. This method requires synchronization among processors for correct computation of the recurrence. Another alternative that eliminates the need for synchronization is to compute the entire recurrence in each processor's local memory and then assign to processor i the recurrence value k such that $k = i \ mod \ nproc$, where *nproc* is the number of processors. A third alternative also eliminates the need for synchronization while dynamically assigning iterations similar to the first method. Each processor records the last iteration (*prev*) it executed and the value of the dispatcher at that point. When it is assigned a new iteration i, it calculates the values of the dispatcher between *prev* and i. The last valid iteration can be found in the same way as in the induction or the associative recurrence cases.

Undoing overshot iterations

The execution of iterations beyond the last valid iteration are undone by checkpointing the loop before the execution and by maintaining a timestamp (i.e., iteration number) of when a memory location is updated. Once the DOALL loop terminates and the last valid iteration is known, the values of memory locations written during the additional iterations are restored using the checkpointed values.

2.1.2 Hardware Based Speculative Run-Time Parallelization Schemes

A number of hardware-based schemes have been proposed that speculatively execute loop iterations to exploit loop-level parallelism on multiprocessor systems [34, 43, 53]. These schemes execute loop iterations in parallel speculating that data dependences do not exist. Data-dependence violations

are detected in hardware using extensions to the cache coherence protocol. The advantage of these hardware schemes over the software schemes is that dependence checking is performed in hardware as the program executes. Hence, there is no separate *inspector* stage which results in lower overhead for dependence-checking compared to the software-only approaches. Another advantage is that the hardware-assisted approaches can be applied to any type of data-structure including pointer-based C codes, since data-dependence violations are detected at hardware level. Some of these schemes use control speculation as well as data speculation which allows them to parallelize *do-while* type loops [34, 43].

2.1.2.1 Thread-Level Speculation

Thread-level speculation (TLS) [34] provides software and hardware support to exploit parallelism in general-purpose application programs. It supports speculative thread-level execution on a single-chip multiprocessor system to parallelize loops with potential data dependences and loops with unknown iteration spaces. TLS requires hardware support to detect data-dependence violations at run-time and to maintain separate speculative and non-speculative memory states.

A code region to be speculatively executed, such as a loop, is decomposed into sequential tasks by the compiler. In case of a loop, a task may correspond to a loop iteration. Each task is assigned a number according to their original execution order. The tasks are cyclically assigned to the processors. Speculative execution begins with an explicit *begin_speculation* operation. At the beginning of speculative execution, each of the P processors in the system is assigned one of the first P tasks. After a processors completes task t, it begins executing task $t + P$ and so on until the speculative execution completes. Because of the dynamic scheduling of tasks, TLS can handle parallel execution of *do-while* loops for which the number of iterations is not known when the loop begins execution.

Each task has an associated state - *current* or *speculative*. The processor which executes the task with the lowest number own the task in the *current* state while the other tasks are in the *speculative* state. A speculative thread performs its writes in its local speculation buffer to maintain correct

memory state. A task can commit only when it is in the current state. This requires that the tasks commit in sequential program order. Memory states, in the speculation buffer, are written back to shared memory when the task commits successfully. Speculative execution terminates with a call to the *terminate_speculation* operation. This operation can proceed only after all the previous tasks have completed, i.e., when this task is in the current state. The execution of *terminate_speculation* flushes the speculative states of all subsequent tasks and signals to the other processors to stop speculative execution.

A data-dependence violation is detected by the hardware when a task writes to a memory location after another task with a higher number has speculatively read the same location. Since data updates are done in local buffers, anti- and output-dependences are eliminated. The hardware passes the task number that violated the data-dependence to a software handler. The software handler is responsible for rolling back and restarting this task and all subsequent tasks with the correct data values.

An extension to a write-invalidate cache-coherence protocol is used in TLS to detect data-dependence violations. Each cache line in a processor's primary data cache has two bits, *modified* and *pre-invalidate*, associated with it. Additionally, each 32-bit word in the cache is associated with a *read* bit. The read bit indicates if the word has been speculatively read by the processor. The modified bit is set when the cache line contains a speculative write state. The pre-invalidate bit is used as a pending invalidation bit. When a task commits, all lines with the pre-invalidate bit set are invalidated. When a write with a task t appears on the bus, all matching cache lines in the processors executing tasks less than t set their pre-invalidate bits. The write causes either an invalidation or a violation for processors executing tasks greater than t. A violation occurs only if the corresponding word has its read bit set.

2.1.2.2 Thread-Level Data Speculation

The Thread-Level Data Speculation (TLDS) [43] is another hardware-based technique that speculatively executes threads in parallel, and recovers and/or restarts the threads whenever a data-dependence violation or a misspeculation is detected at run-time. TLDS requires hardware support to detect data-dependence violations and to buffer speculative memory states until they can be safely committed. The hardware support is achieved through an extension of the invalidation-based cache coherence protocol.

A compiler breaks down a program into potentially independent dynamic instruction sequences called *epochs*. An epoch may correspond to a loop iteration, for example, if the compiler believes that cross-iteration dependences are unlikely to occur. Each epoch is assigned an *epoch number* according to their original sequential execution order. The epochs are speculatively executed in parallel. If the hardware detects a data-dependence violation, a software recovery code is invoked to recover from the misspeculation and restart execution with the correct data. If the epochs are executed with control speculation (e.g., iterations of a *do-while* loop with an unknown termination condition), epochs must be destroyed when a misspeculation occurs. In this case, destroyed epochs do not need to be restarted. A cancellation message is sent from an earlier epoch to a later one to set a flag indicating that the epoch needs to be destroyed. Each processor buffers its memory states until it knows that the speculation has succeeded. The epoch with the lowest number sends a *homefree token* to its successor when it is done committing its data updates to memory. An epoch receiving the homefree token is allowed to commit its updates. Thus, memory updates are essentially serialized in the TLDS model.

TLDS allows threads to be switched between epochs if there is load imbalance between epochs. The load imbalance may delay later epochs as they wait for the homefree token to arrive. To improve performance, the waiting thread is allowed to suspend its current epoch and begin executing a new epoch. The thread switches back to the old epoch once the homefree token arrives. Because of the

overhead associated with thread switching (for saving and restoring thread states), it is done only when the expected delay in the current epoch is greater than the thread switch time and the new thread has a high probability of succeeding.

An extension to the invalidation-based cache coherence scheme is used in TLDS to track data-dependence violation at run-time. The state of each cache line is augmented to indicate whether the cache line has been speculatively loaded and/or modified. Each cache also maintains the epoch number of the current epoch the corresponding processor is executing, and a flag indicating if a data-dependence violation has occurred. When an epoch performs a speculative load, the corresponding cache line is marked as speculatively loaded. When an epoch stores to a cache line, an upgrade request is generated with its epoch number. When the upgrade request is received by another cache, a data-dependence violation is detected as follows. If the corresponding cache line is present and it is marked as speculatively loaded, there is a RAW violation, provided that the epoch number of the cache generating the upgrade request is lower than the current epoch number. This indicates that the upgrade request came from a less speculative epoch and, hence, the speculative load is invalid. In case of a RAW violation, the violation flag in the cache is set. Any cache line with a speculative state must remain in the cache until the corresponding epoch receives its homefree token. If a speculative cache line must be replaced, it is treated as a dependence violation and the epoch is re-executed.

2.1.2.3 Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors

Zhang et al. [53] proposed a hardware-based technique for speculative run-time parallelization in distributed shared-memory multiprocessor systems. This scheme is a hardware extension to the speculative version of the Privatizing DOALL test [37]. The basic idea is to speculatively execute the code in parallel and use extensions to the cache coherence protocol hardware to detect any dependence violations. As soon as a dependence is detected, the execution is terminated, the state

is restored, and the code is re-executed sequentially. This scheme does not use control speculation and, hence, can not handle *do-while* loops.

This scheme extends the cache coherence protocol hardware with additional transactions to flag any cross-iteration data dependences at run-time. The new transactions are grouped into two sets, *non-privatization* and *privatization* algorithms. These algorithms are used by the non-privatized and privatized arrays under consideration. An array is considered as privatizable if every element of it is read-only or if every read access of an element is preceded by a write access of the element in the same iteration. Such an array can be privatized by using a private copy in each processor to eliminate anti- and output dependences.

Non-Privatization Algorithm

In the non-privatization algorithm, a loop is identified as parallel if each element in the array under test is either read-only (*ROnly*) or is accessed by only one processor, i.e., not shared (*NoShr*). The cache directory maintains some state for each element in the array under test including a *ROnly* bit and a *NoShr* bit that indicate whether the element is read-only or not-shared, respectively. Another bit, *First*, keeps the *id* of the processor that first accesses the element. Before the loop starts execution, all these bits are cleared. If the first access to an element is a write, both *First* and *NoShr* are set. If the first access is a read, only the *First* bit is set. In the latter case, if the next access is a read by a different processor, *ROnly* is set. If it is a write by the same processor, *NoShr* is set. In all cases, a write to a *ROnly* element causes the failure of parallelization. If a processor other than *First* tries to access a *NoShr* element or tries to write the element, the parallelization again fails.

Privatization Algorithm

In this algorithm, each processor works on a private copy of the array under test. For each element some state is needed in the directories of both the shared array and its private copy. This algorithm has three versions. The Basic Privatization Algorithm (BPA) is the most conservative in

determining privatizable elements and requires the least amount of hardware support. The Advanced Privatization Algorithm (APA) is the most general and requires the most support. Finally, the Blocked Advanced Privatization Algorithm (BAPA) parallelizes a superset of the loops parallelized by APA, but requires less support than APA. In the following we describe the basic APA approach.

If an iteration reads a private element before the same iteration writes it, the iteration is called a *read-first* iteration for that element. The directory of the shared copy of the array maintains two timestamps for each array element. One timestamp keeps the number of the highest *read-first* iteration for the element executed so far by any processor (*MaxR1st*). The other keeps the number of the lowest iteration executed so far by any processor that involved writing the element (*MinW*). The parallelization fails when *MaxR1st* is larger than *MinW*.

During speculative execution, processors access private data. To identify *read-first* iterations, the directories of the private copies of the array keep two timestamps for each element. The first one is *PMaxR1st*, the number of the highest *read-first* iteration for the element executed so far by the processor. The second one is the number of the highest iteration executed so far by the processor that involved writing the element (*PMaxW*).

The algorithm then proceeds as follows. Every time a processor reads an element, it checks whether this is a *read-first* iteration for that element. If the element is in the cache, the processor uses the state of the cache-tags where both *Read1st* and *Write* will be zero for a *read-first* iteration. If the element is not in the cache, the state of the directory for the private array is used. For a *read-first* iteration both *PMaxR1st* and *PMaxW* are less than the current iteration number. If the iteration is a *read-first*, the directory for the shared array is notified. In the directory, the current iteration number is compared to *MinW*. If the former is larger, the parallelization fails. Otherwise, *MaxR1st* is set to the maximum of its current value and the current iteration number. The cache tags and the directory are updated as necessary.

Every time a processor writes to an element of the array, a check is made to see if it is the first write to the element in this iteration. If the element is in the cache, the cache tag state *Write* will

be zero. If the element is not in the cache, the *PMaxW* timestamp in the directory will be less than the current iteration number. If it is the first write, the directory of the private array is notified for two reasons. The first reason is to update *PMaxW* to the current iteration number. The second reason is that this may be the very first write by the processor to the element. In this case, the directory for the shared array has to be notified. In the directory, the current iteration number is compared to *MaxR1st*. If the former is lower, the parallelization fails. Otherwise, *MinW* is set to the minimum of its current value and the current iteration number.

This dependence checking is done for each read and write access of the array under test. As soon as parallelization fails for an access, the execution is aborted and restarted sequentially from the very beginning with all shared data restored to their original values.

2.1.3 Multithreaded Processors

A number of multithreaded processor architecture models [14, 42, 44] use control and/or data speculation to concurrently execute multiple threads. Unlike the hardware-based speculative run-time schemes discussed in the previous section that support speculative execution on existing multiprocessor architecture, these processor architecture models use specialized hardware to execute threads concurrently. The multiscalar architecture [42], the single-program speculative multithreading (SPSM) architecture [14], and the superthreaded architecture [44, 45, 46] are some multithreaded processor models that support control speculation to concurrently execute threads for which the termination conditions are not known when execution begins. Speculative threads are either committed (if speculation succeeds) or squashed (if speculation fails) after an earlier thread determines the termination conditions. Multiscalar and SPSM also support data speculation. Memory address disambiguation hardware is used to detect data-dependence violations at run-time. The hardware keeps track of the memory access operations generated by all the active threads and signals a dependence violation when an earlier thread writes to a location after a later thread has read from it. Once a dependence violation is detected, the later thread and all of its successor threads are squashed and re-executed

with the correct data values.

2.1.4 Speculative Multithreading Versus Existing Run-Time Parallelization Schemes

The speculative multithreading parallelization model proposed in this thesis exploits coarse-grained loop-level parallelism on shared-memory multiprocessors. It uses a thread pipelining execution model with run-time dependence analysis and control speculation to parallelize loops with run-time data dependences as well as traditionally sequential constructs such as *do-while* loops. Unlike the existing software-based run-time parallelization schemes [11, 37, 38, 54], there is no separate inspector and executor stages in the proposed model. The dependence checking and execution are overlapped in concurrent threads executing in a pipelined manner. Since the proposed scheme does not require any additional overhead to execute the *inspector* stage, it is expected to perform better than the *inspector-executor* algorithms. The *inspector-executor* based schemes can parallelize *for* loops with run-time data dependences on array data structures only. These schemes are not general enough to handle run-time dependence checking of general pointer structures. The run-time dependence checking in the speculative multithreading model is applicable to any type of data structure, including arrays, linked lists, pointers, and so forth.

The speculative multithreading parallelization model can execute loop iterations speculatively, as can the Privatizing DOALL test [37] and the hardware-based schemes [34, 43, 53]. This speculation results in better speedup if the speculation is usually correct. However, the Privatizing DOALL test applies data speculation to parallelize DOALL loops. The speculative multithreading model, on the other hand, uses control speculation to parallelize loops with unknown iteration spaces. The hardware-based schemes [34, 43, 53] do use control speculation. However, unlike these hardware-based schemes, the speculative multithreading execution model is implemented entirely in software and, thus, does not require any additional support from the hardware. This software-only implementation makes the proposed parallelization model a cost-effective approach to improving

application-level performance on existing shared-memory multiprocessor systems instead of using costly specialized hardware as needed by the existing speculative thread-level execution models [34, 43, 53].

2.2 Dynamic Processor Allocation Schemes

Once a sequential application program has been parallelized using an appropriate parallelization tool, the parallel program must efficiently be executed on multiprocessors to obtain speedup. The performance improvement through parallel execution depends on the workload of the parallel code region as well as the number of processors the application program can use without introducing too much contention for processors. If the parallel code region does not have a sufficiently large workload to amortize the associated parallelization overhead, the parallel execution may even result in slower execution than its serial counterpart (i.e., a slowdown, or a speedup less than one). Parallel execution may also lead to poor performance, even with sufficiently large workloads, if the program runs simultaneously with other programs in a multiprogrammed environment. In a standard multiprogrammed environment, parallel applications time-share processor resources with other parallel or sequential applications. The high overheads associated with context-switching, together with poor cache utilization, can result in poor parallel execution performance in time-shared multiprogrammed environments [16, 32, 35, 47].

For programs with dynamically varying behavior, such as programs where the number of loop iterations changes dynamically across invocations, it is not possible to determine at compile-time the number of processors that can be used most efficiently at each invocation. Furthermore, if the target system on which the program will execute is not known at compile time, the parallelization overhead is also unknown at that time making any decision about the number of processors to use impossible to determine. The system load in a given multiprocessor system also changes across time making it difficult to assign a fixed number of processors to a program for efficient parallel execution.

Thus, it is important that the number of processors allocated to a program be determined during run-time based both on program workload and on system load on a given target system.

A number of run-time systems have been proposed to dynamically allocate processors to parallel program regions. Some of these systems use dynamic program behavior information to decide if loop parallelization will be profitable [48, 49, 50]. Based on a run-time decision, these systems allocate either one processor or all available processors to the application program. Other systems use system load information to allocate as many processors as possible without overloading the system [41, 51, 52]. A few systems consider both the program behavior and the system load in allocating processors to parallel loops [17].

2.2.1 Dynamic Serialization

The dynamic serialization [48, 49] approach dynamically serializes parallel program regions that cannot amortize the parallelization overhead. The dynamic serialization framework uses an inspector-executor model to identify loops that need to be serialized. It is implemented in the Polaris parallelizing compiler [6] to dynamically serialize parallel applications written in the OpenMP shared-memory programming application program interface (API) [13].

Initially, each loop is classified as PARALLEL and is allowed to execute once in parallel in the WARMUP state before any timing and decision-making can be done. After the initial run, a loop goes through the TEST state where it is run in parallel again. This parallel test run is timed to decide if the loop should subsequently be executed sequentially or in parallel. If the test decides that the subsequent execution should be serial, the loop moves to the SERIAL state. Otherwise, the loop moves to the PARALLEL state. In both cases, the current number of loop iterations is recorded. Once a classification has been made, a loop remains in the corresponding state until the program ends or there is a significant change in the loop iteration count which may change the classification. In the latter case, the loop is moved to the WARMUP state and the entire process is repeated.

The loop classification is performed using two different approaches. The *scaled-test* approach

compares the measured parallel execution time to an estimated serial execution time and classifies the loop as SERIAL if the parallel run time is longer. The estimate of the serial time is based on a sequential execution profile of the program on a base machine. This base time is subsequently scaled to the target machine using a matrix multiplication kernel to generate the scaling factor.

The *overhead-test* approach classifies a loop as serial if the measured parallel runtime is below certain threshold as determined by the parallelization overhead. The parallel execution time, T_p, is modeled as the sum of the work done by the loop, t_p, and the overhead, t_{oh}. The overall parallel execution time of the loop is then, $T_p = t_p + t_{oh}$. T_p must be less than the sequential execution time, T_s, for the parallel run to be profitable. Assuming a perfect speedup, T_s can be approximated as, $T_s = t_p \times p$, where p is the number of processors. Thus, the parallelization decision is based on the test, $t_p \times p > T_p$ which can be translated to $T_p > t_{oh} \times p/(p-1)$. In this approach, the overhead is assumed to be comprised only of the fork-join costs. The overhead is measured by timing an empty parallel loop on the target machine. The *overhead-test* is quite similar to the parallelization heuristic used in the dynamic processor allocation scheme proposed in this thesis, except that we estimate the parallel runtime using a different approach.

The processor-allocation scheme in the dynamic serialization approach allocates either one processor or all P processors in the system to execute the loop. The workload of the loop determines whether to use one processor or all P processors. The system load effect is not considered at all in determining the number of processors that can be effectively used, however. Thus, this technique is better suited for a dedicated system than a multiprogrammed system.

2.2.2 Adaptive Parallelism in SUIF Compiler-generated Code

The Stanford University SUIF compiler [50] generates parallel SPMD C programs from sequential Fortran or C programs. All parallelizable loops have a sequential and a parallel version. The run-time system dynamically serializes loops that do not have sufficient granularity for profitable parallelization. The serialization decision is based on static information about the per iteration work

of the loop and dynamic information about the number of loop iterations actually executed. This system is similar to the dynamic serialization [48, 49] approach in that both use either one or all available processors to execute the loop. However, in SUIF, the serialization decision is made only once for the first invocation of the loop. The loop classification does not change across invocations and, thus, the system cannot handle dynamically varying loop behavior.

The adaptive run-time system proposed in [17] extends the basic SUIF run-time system to efficiently map parallel loops to processors using run-time information. The processor allocation decision is based on the measured speedups of the parallel region in previous invocations. The speedup is computed using the execution time previously measured for the loop in a sequential pre-run of the program. Initially, a program is assigned the number of threads it requested. The number of threads are adapted upwards or downwards on subsequent loop invocations based on the observed loop behavior. If the measured speedup of a loop is above a certain factor of the ideal speedup, the system gradually increases the number of threads allocated to the application program. Similarly, the number of threads is gradually decreased if the observed speedup is below some factor of the ideal value. Since speedup is affected by changes in the loop granularity as well as changes in the system load, this technique indirectly uses program behavior and system load information in its processor allocation scheme. To avoid penalizing programs for brief and spurious performance degradation, the system waits for a number of consecutive poor parallel execution performances before it adapts down the number of threads for this loop. Once the parallel execution time of a loop falls below a given threshold, the loop is classified as sequential and never again executed in parallel during this run of the program.

2.2.3 Loop-Level Process Control

Loop-Level Process Control (LLPC) is a processor-allocation technique that dynamically adjusts the number of processors used by each parallel region of the application program based on the current system load and a static estimate of the amount of parallelism in the loop [51, 52]. While

this technique allows applications to utilize as many processors as possible without overloading the system, it cannot allocate processors based on dynamically varying program behavior.

In LLPC, each application checks the system load at the beginning of the execution of a parallel code region. The application determines the number of processors, $allow_p$, available for execution of the current parallel region using the following heuristic:

$$allow_p = max[1,\ min(no_needed,\ total_physical_P - system_load)]$$

where $total_physical_P$ is the total number of processors in the system, $system_load$ is the current system load, and no_needed is the number of processors required to attain the maximum parallelism in the parallel region. The application is allocated $allow_p$ processors for the current parallel execution. When the execution of the parallel code region is complete, all the processors are released and made available to the system.

In the SGI Challenge system implementation of the LLPC algorithm [51], the system load is determined at the user-level by having LLPC-enabled applications update a shared memory location to indicate their respective load requirements. If non-LLPC-enabled applications are running in the system, however, the system load information will be inaccurate. In [52], a Solaris operating system feature called scheduling control is used to measure the system load more accurately. This approach allows LLPC-enabled applications to coexist with other applications running in the system.

2.2.4 Automatic Self-Allocating Threads

The Automatic Self-Allocating Threads (ASAT) is another approach that dynamically adjusts the number of processors allocated to a program based on system load [41]. ASAT uses an indirect measurement of system load by occasionally measuring the time required to synchronize threads. If the total number of threads executing in the system exceeds the available number of processors, the time required at the synchronization barriers will be much higher. Thus, the barrier synchronization time indicates if the system is overloaded. In that case, the number of threads allocated to a program is reduced one at a time until the system load is balanced. To increase the number of

threads allocated to a program, the barrier test is performed with one more thread than is already allocated. If the barrier synchronization time indicates that this additional thread can be executed efficiently, one more thread is allocated to the executing program.

2.2.5 Comprehensive Dynamic Processor Allocation Scheme Versus Existing Processor Allocation Schemes

Most existing processor allocation schemes make their allocation decision using either only program workload or only system load information. Techniques such as the dynamic serialization framework [48, 49] and the dynamic parallelization mechanism in the SUIF compiler [50] allocate either one or all of the processors in the system to the application program based on the current workload. These techniques are, thus, applicable to dedicated systems only. Furthermore, these techniques do not consider any intermediate configuration of the target system for effective parallelization. For example, a parallel code region may have sufficient workload to be effectively parallelized using only 8 processors on a 16-processor system. These techniques will simply serialize the code region as it does not have sufficient workload to be parallelized on 16 processors.

Other techniques, including LLPC [51, 52] and ASAT [41], dynamically allocate processors to parallel applications based on the current system load. Thus, they work well in multiprogrammed environments, but cannot handle programs with dynamically varying behavior. In contrast to these existing approaches, the comprehensive dynamic processor allocation scheme proposed in this thesis allocates processors to an application program considering both the dynamically varying program behavior and the varying system load information. Thus, this approach works well in both dedicated and multiprogrammed environments. In this respect, it is similar to the technique in [17], although the implementation is quite different from their approach. For instance, as will be discussed later in Chapter 4, the proposed scheme uses direct measurements of the system load and the loop workload in its decision-making process. Furthermore, loop classifications can be dynamically updated at any time during a program's execution, whereas in most of the previous approaches (including [17]),

loops can switch from parallel to sequential execution, but never switch back. Finally, the proposed dynamic processor allocation scheme can handle dynamic parallelization of both *for* loops and *do-while* loops. Existing techniques [17, 50, 48, 49] can only handle well-defined *for* loops for which loop workload can be estimated quite accurately using loop iteration count.

2.3 Techniques for Improving Java Performance

Java's language-level support for multithreading allows development of platform independent parallel application programs. Once a sequential Java program has been transformed to a parallel program, it can potentially be executed on a variety of hardware platforms supporting the JVM. A number of techniques have been developed to improve execution-time performance of Java programs by parallelizing Java source code or bytecodes using Java multithreading support [26]. These techniques use traditional parallelization models to extract loop-level parallelism from application programs. These techniques cannot parallelize loops with run-time data dependences or loops with traditionally sequential constructs, however.

Some dynamically adaptive run-time techniques have been proposed to improve Java execution-time performance. In the context of Java, dynamic adaptation takes the form of dynamic compilation [3]. Thus, these techniques deal with efficient native code generation from Java bytecodes during run-time.

2.3.1 Java Parallelization Techniques

The JAVAR [4] and the JAVAB [5] parallelization tools exploit implicit parallelism in loops and multiway recursive methods in sequential Java programs to generate parallel code using the standard Java multithreading mechanism. The JAVAR tool is a source-to-source restructuring compiler that applies the parallel transformation at the Java source code level. It relies on explicit annotations by the programmer to transform a sequential Java source code into a corresponding parallel code. The

JAVAB [5] tool, on the other hand, works directly on Java bytecodes. It can automatically detect and exploit implicit loop-level parallelism from the original sequential bytecodes.

The Do! project [31] provides a parallel framework embedded in Java and a library of generic classes to support both data and control (or task) parallelism. Do! also provides a framework for the automatic generation of distributed-memory programs from shared-memory Java programs. The distributed framework relies on the standard Java remote method invocation (RMI) interface and a runtime library that supports remote creation of objects.

Tiny Data-Parallel Java [19] extends the Java language to support data-parallel programming. It defines data-parallel classes whose methods are executed on a large number of virtual processors. An extensible Java preprocessor, EPP, translates the data-parallel code into standard Java code using the Java thread libraries and synchronization primitives. The preprocessor can produce Java code for multiprocessor and distributed systems as well. DPJ [20] defines a parallel framework through a Java class library for data-parallel programs.

2.3.2 Dynamic Compilation

Java bytecodes are often compiled dynamically at run-time to generate native codes for the target architecture to improve performance over the interpreted execution of Java bytecodes. This on-the-fly compilation of Java bytecodes is referred to as just-in-time (JIT) compilation. Since the compilation is done as the program is executed, the compilation time in JIT compilation adds directly to the application's total execution time. Thus, the quality of the code optimization is severely constrained by compilation speed. Dynamic compilation addresses this problem of JIT compilation by optimizing only the portions of the code that are most frequently executed, i.e., program *hotspots*. Most programs spend the majority of the time executing only a small fraction of their code. Thus, optimizing only the hotspot methods should yield a large performance gain while keeping the compilation speed relatively fast.

Sun's *Hotspot* Java Virtual Machine [15] uses dynamic compilation to generate optimized native

machine code during runtime. The Hotspot engine contains both a run-time compiler and an interpreter. The first time a method is executed, it is interpreted using a profiling interpreter that gathers run-time information about the method. This information is used to detect hotspots in the program and to gather information about program behavior that can be used to optimize generated native code in later stages of program execution. After the hotspot methods are identified, they are dynamically compiled to generate optimized native machine code. Infrequently executed code continues to be interpreted, decreasing the amount of time and memory spent on native code generation. Because a program is likely to spend the majority of its execution time in the hot-spot regions detected by the interpreter, the compiler can spend more time optimizing the generated code for these sections of the program than a JIT-compiler while still producing an overall improvement in execution time. During code generation, the dynamic compiler performs conventional compiler optimizations, Java specific optimizations, and inlining of static and dynamic methods. The inlining optimizations are designed to be reversible due to the problems associated with dynamic class loading.

IBM's *Jalapeno* Java Virtual Machine [7] includes an adaptive dynamic optimizing compiler that generates optimized machine code as the program is executed. The Jalapeno JVM does not use an interpreter. Instead, the first execution of a method is handled by quickly compiling a method into an unoptimized executable code. The dynamic optimizing compiler later generates optimized executable code from the bytecodes of the hotspot (i.e., frequently executed) methods as determined by runtime profile information. The program adaptation in Jalapeno starts with the instrumentation and recompilation of executable code. The executable code is instrumented to gather context sensitive profile information to aid optimization. The profile information is used to detect program hotspots. When a certain performance threshold is reached, the dynamic optimizing compiler is invoked to recompile the hotspot methods using context specific optimizations. The unoptimized code is then replaced by optimized code based on the collected profile information. Program adaptation continues in this cycle, with executable code being improved on every optimization iteration.

2.4 Conclusion

In this chapter, we presented some previous work on run-time parallelization and dynamic processor allocation. We also identified the differences in our proposed solutions for run-time parallelization and dynamic processor allocation with those of the existing techniques. The following chapters will elaborate on the implementation of our proposed schemes.

Chapter 3

Speculative Multithreading Parallelization Model

Speculative multithreading is a new parallelization technique for exploiting loop-level parallelism from general-purpose application programs in shared-memory multiprocessor systems. This parallelization model allows concurrent execution of potentially dependent loop iterations in a pipelined fashion with run-time data-dependence checking and control speculation. The run-time data-dependence checking makes it possible to parallelize loops for which data dependences cannot be analyzed at compile-time (e.g. loops with array accesses through subscripted subscripts). Furthermore, the combination of speculative execution and run-time data-dependence checking allows the parallelization of traditionally sequential constructs, such as *do-while* loops and nested *if-then-else*. Thus, the speculative multithreading parallelization model is applicable to a wide variety of loop constructs that cannot be parallelized using traditional parallelization techniques.

The speculative multithreading parallelization model is based on the fine-grained thread pipelining model proposed for the *superthreaded* processor architecture [44, 45, 46]. The superthreaded architecture uses a thread pipelining execution model in which threads are dynamically initiated

and executed. We extend the fine-grained superthreaded model by implementing it in a set of software library routines to parallelize coarse-grained applications on off-the-shelf shared-memory multiprocessor systems. The speculative multithreading parallelization model is applied to both C and Java application programs using special library developed in C [23] and Java [24] respectively.

In Section 3.1, we present an overview of the thread pipelining execution model of the superthreaded processor architecture. Section 3.2 describes how the fine-grained thread pipelining execution model is adapted to the shared-memory multiprocessor architecture to parallelize coarse-grained application programs. Parallel execution of programs transformed using the speculative multithreading model is associated with certain execution-time overhead which are discussed in Section 3.3. In Section 3.4, we present the C and Java library implementations of the speculative multithreading parallelization model. Finally, some Java language-specific issues are discussed in Section 3.5 that need to be addressed when parallelizing Java programs.

3.1 Superthreaded Architecture

The superthreaded architecture [44, 45, 46] exploits task-level parallelism using multiple threads of control. A superthreaded processor consists of a number of *thread processing units* that share an instruction and data cache. At run-time, the multiple thread processing units, each with its own program counter and instruction execution data path, can fetch and execute instructions from multiple program locations simultaneously. The basic architecture of a superthreaded processor is shown in Figure 3.1.

3.1.1 Thread Partitioning

The compiler for the superthreaded architecture statically partitions the control flow graph of a program into the individual threads. Each thread is run on a separate thread processing unit. The execution of a program starts from its entry thread. It can then fork a successor thread on another

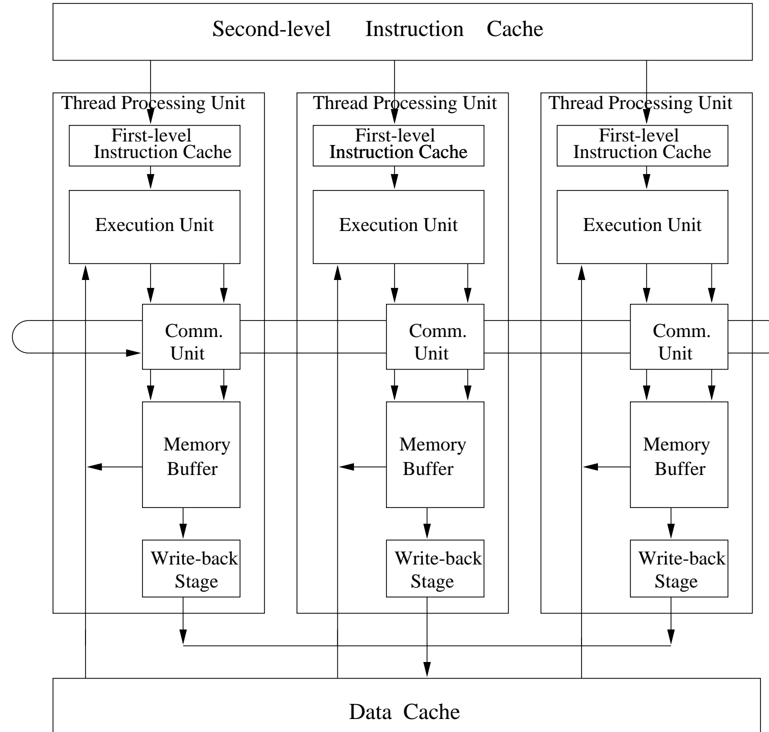

Figure 3.1: Basic organization of a superthreaded processor.

thread processing unit. This successor thread can in turn fork its own successor thread. This process continues until all thread processing units are busy. Among the multiple threads running on the superthreaded processor, the oldest thread in the sequential order is referred to as the *head thread*. All the other threads derived from the head thread are called *successor threads*. After the head thread completes its computation, it will retire and release its thread processing unit. Its successor then becomes the new head thread. The completion and retirement of the threads must follow the original sequential program order to ensure correct results.

In the superthreaded execution model, successor threads can be forked with or without control speculation. When a thread forks a successor without control speculation, it must ensure that all of the control dependences of the successor thread have already been satisfied. If a thread forks a successor thread with control speculation, however, it must subsequently verify all of the speculated

control dependences. If any of the speculative dependences evaluate to false, the thread must abort its successor thread and all of the following threads.

3.1.2 Thread Pipelining Execution Model

The superthreaded architecture uses the *thread pipelining* model to overlap thread execution and to enforce data dependences between concurrently executing threads. In this model, thread initiation and data forwarding are performed through explicit thread management and communication instructions. The execution of a thread is partitioned into several stages, each of them performing a specific function. Figure 3.2 shows the pipelined execution of contiguous threads in a superthreaded processor. The function of each of the thread pipelining stages is described in the following sections.

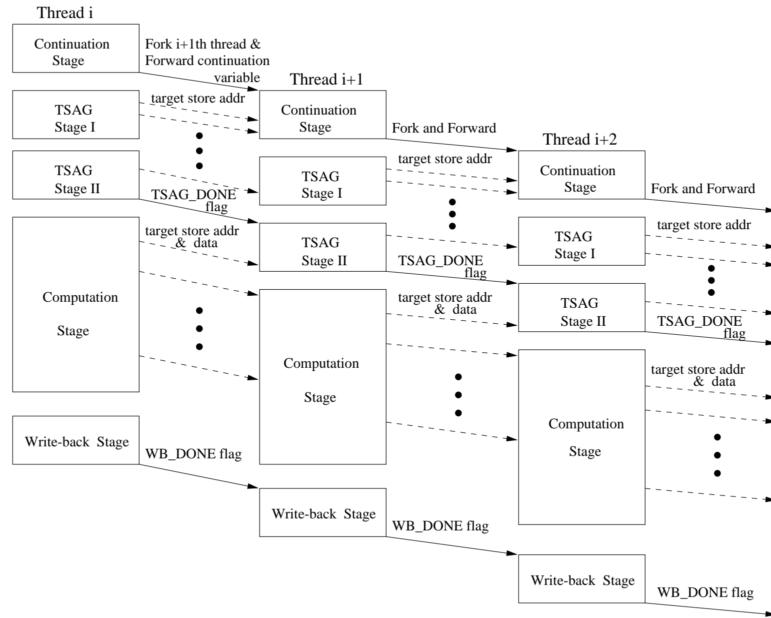

Figure 3.2: Pipelined execution of concurrent threads in a superthreaded processor.

3.1.2.1 Continuation Stage

After being initiated by its predecessor thread, each thread begins with the `continuation stage`. The main function of this stage is to compute the recurrence variables, such as loop index variables,

needed to fork the next thread. All of these types of variables are called `continuation variables`. The values of the continuation variables must be forwarded to the next thread processing unit before the next thread can be activated. This stage ends with a `fork` instruction, which ensures serialization of the continuation stages of successive threads. This serialization is necessary because the computation of the continuation variables in the current thread is dependent on the results from the continuation stage of the previous thread.

3.1.2.2 Target-Store-Address-Generation (TSAG) stage

The `Target-Store-Address-Generation stage` computes the addresses of the write operations upon which concurrent threads may be data-dependent. These addresses are referred to as `target store addresses`. These store addresses are computed at run-time, and are forwarded to the memory buffers of successor threads using the `allocate_ts` instruction. To guarantee program correctness, a successor thread is not allowed to perform any load operation which can be data dependent on those store operations marked with an `allocate_ts` instruction until its predecessor thread has completed the TSAG stage and has forwarded all the target store addresses to its memory buffer. This ordering is enforced using two synchronization instructions. The `release_tsag_done` instruction is placed at the end of the TSAG stage to send a `tsag_done_flag` to its successor thread. A corresponding `wait_tsag_done` instruction is placed in the successor thread before its corresponding load operations. This mechanism effectively implements run-time data-dependence checking.

To allow more overlap between threads, the TSAG stage can be further partitioned into two parts. The first generates target store addresses that do not have any data dependences on earlier threads. This part does not need to wait for the `tsag_done_flag` and so can complete quickly. The part that is data dependent on previous threads needs to wait for the `tsag_done_flag` as before.

3.1.2.3 Computation Stage

The `computation stage` performs the main computation of a thread. If a thread executes a load operation whose address matches that of a target store entry in its memory buffer during the computation stage, the thread will either read the data from the entry if it is available, or it will wait until the data is forwarded to its memory buffer by an earlier thread. If the thread is computing the value of a target store, it needs to forward the data to the memory buffers of all its concurrent successor threads using a `store_ts` or `release_ts` instruction.

If a thread completes normally without being aborted by a predecessor thread, it will end with a `stop` instruction. After executing the `stop` instruction, the thread waits until it becomes the head thread and then performs its `write-back stage`. If a thread determines that a control speculation is incorrect, however, it kills all its successor threads using the `abort_future` instruction.

3.1.2.4 Write-back Stage

In the `write-back stage`, a thread writes all the data stored in its memory buffer to the main memory through the shared data cache. The write-backs are performed in the sequential program order to preserve non-speculative memory state. This ordering also eliminates all output and anti-dependences between threads. After completing the write-back stage, a thread is retired and the thread processing unit becomes idle until it is again scheduled with a new thread. The serialization of the write-back stage is achieved with two synchronization instructions. The `wait_wb_done` instruction causes the thread to wait on the flag `wb_done`. A corresponding `release_wb_done` instruction executed in the head thread sets the `wb_done` flag for its immediate successor thread.

3.2 Thread pipelining execution model in Speculative Multithreading

The speculative multithreading parallelization model extends the basic thread pipelining model discussed in Section 3.1 to speculatively parallelize coarse-grained applications on conventional multiprocessor systems. While the original superthreaded architecture uses specially-designed hardware to support the pipelined execution of concurrent threads, this coarse-grained version of the thread pipelining model is implemented entirely in software on a standard shared-memory multiprocessor system. It uses the same four thread pipeline stages as in the fine-grained superthreaded model, i.e., the continuation, TSAG, computation and write-back stages. The functionality of the stages remains the same, but the implementation of the stages in this case is adapted to match the requirements of the shared-memory architecture.

For the adaption of the thread pipelining execution model on the shared-memory architecture, we classify loops in two categories – well-structured loops with known loop bounds i.e., *for* loops, and loops with indeterminate termination condition, such as *do-while* loops or *for* loops with early exits. The loops in the second category cannot be parallelized using traditional techniques since the loop bound is not known at compile time. The speculative thread execution in the thread pipelining model allows these types of loops to be easily parallelized, however. As discussed later, the type of the loop being parallelized imposes different implementation requirements on some of the pipeline stages. Loops in either category may have run-time data dependences that make them difficult to be parallelized using static compile-time analysis. These loops require run-time schemes for parallelization [11, 37, 38, 54]. The pipelined execution of threads with in-order execution of the TSAG stages allow the speculative multithreading model to enforce run-time dependences among concurrently executing threads. Since the run-time data-dependence testing is performed during actual execution, the speculative multithreading model is expected to parallelize these type of loops with lower overhead than existing inspector-executor based run-time techniques that require separate inspector

and executor stages to enforce the dependences. In the following, we discuss the implementation details of the thread pipeline stages on the shared-memory multiprocessor architecture.

3.2.1 Mapping the Thread Pipeline Stages to Software

In the speculative multithreading model, concurrent threads execute on multiple processors. A unique process is created on each processor to act as a thread processing unit for the threads executing on the corresponding processor. These processes initiate threads for execution and each thread goes through the different pipeline stages, retiring when it completes all of the stages. At this point, the processor running the thread waits until a new thread is initiated. It then begins executing this new thread. Threads executing on different processors communicate through the shared-memory space using specially-developed software library calls.

Figure 3.3 shows the pipelined execution of concurrent threads in the speculative multithreading parallelization model. As in the original superthreaded processor architecture, the threads in this software models go through four different stages although their implementation is different. The language features of the programming language used in the speculative multithreading model sometimes determine the requirements for the actual implementation. In the following, we describe the software implementation of the thread pipelining execution model with language-specific implementations where necessary.

3.2.1.1 Continuation Stage

As shown in Figure 3.3, each thread starts with the continuation stage. This stage serializes the threads in order of the *thread number* to ensure the correct computation of the loop control variables. Since the processes initiating the threads are all created simultaneously at the beginning of the program's execution, we need some mechanism to allow threads to wait at the entry point of a thread until a predecessor thread signals a successor thread to continue. We introduce the flag `thread_active` to indicate that a thread on a particular processor can proceed.

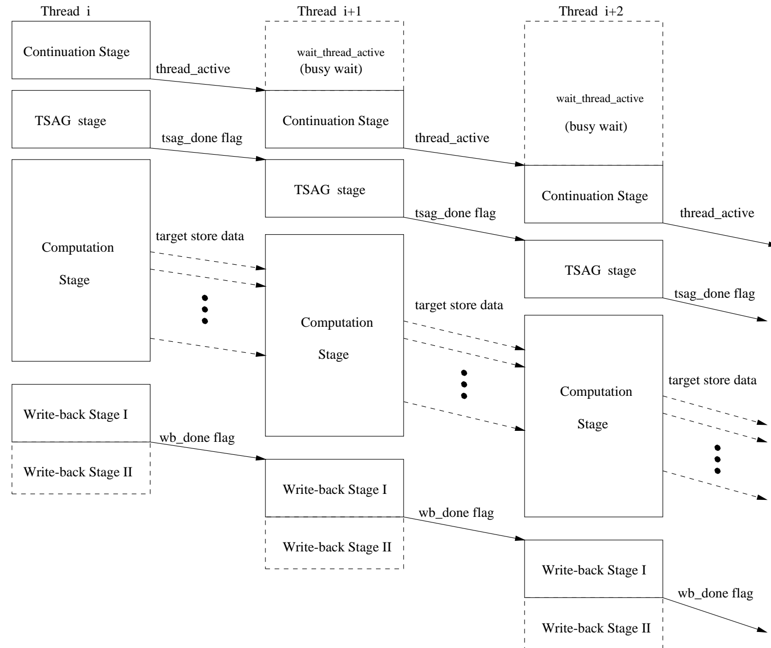

Figure 3.3: Pipelined execution of threads in the speculative multithreading parallelization model.

Initially, the thread_active flag points to the first thread, so that only the first thread enters the continuation stage. When the first thread completes this stage, it sets the `thread_active` flag for the next thread so that its successor thread can start. When this thread completes its continuation stage, it will then set its successor's flag. In this way, successive threads on different processors will be started sequentially. During this stage, each thread sets the variable `thread_num` in the processor on which it is running. This variable identifies the current thread running on the processor and is used to coordinate the write-back process when threads are aborted.

3.2.1.2 Target-Store-Address-Generation (TSAG) Stage

The superthreaded architecture uses the memory address of every data access to determine at runtime whether a dependence exists or not. To enforce such a runtime data-dependence test for every data access in the software model is expensive both in terms of memory usage and execution-time overhead. Data dependences that can be determined at compile-time can be enforced through

synchronization primitives. Thus, only memory accesses that cannot be statically analyzed for the existence of data dependences are handled by the runtime approach in the software implementation.

Threads that may be data-dependent on previous concurrent threads must wait on a flag corresponding to the dependent data item. Because these data values are allocated in shared-memory, there is no need to actually forward the data values between the threads. Instead, when the flag is set, the waiting thread knows that the data is available and can be read from the shared-memory. A successor thread can continue computation involving a dependent data item only if the corresponding flag is set. The TSAG stages are synchronized with a `tsag_done` flag using the `wait_tsag_done` and the `release_tsag_done` library calls, as in the original superthreaded model.

In C programs, data dependences may be caused by array references as well as pointer references. This requires that the actual memory addresses of the data accesses be used to resolve run-time data dependences. The runtime data-dependence test in the C implementation uses three global data structures. Each possibly dependent memory location has an entry in the `address` data structure that holds its memory address. The identifier of the thread accessing the data item is stored in the corresponding entry in the `thread` data structure. Finally, a flag indicating whether the data access is complete is stored in the corresponding `flag` data structure. During the TSAG stage, each thread sets the next available entry in the `address, thread`, and `flag` data structures for every data access that may result in a cross-iteration dependence for its successor threads. The flag entry is set to 0 indicating that the access has not occurred yet. A successor thread can continue computation involving a dependent data item only if the corresponding flag is set to 1.

Since Java does not support pointer structures, run-time data-dependence testing becomes much simpler. In this case, run-time testing involves only array accesses. For each such data array, a shared array of flags is maintained by each processor. Each thread on the corresponding processor sets the appropriate flag entries with its thread number. A thread releases its flag entry by setting it to NULL. Using a flag entry for each array element allows us to make direct access to the flag entries using array indexing. In the C implementation, on the other hand, a thread needs to make

a linear search through the entries in the `address` structure to look for a possible dependence.

3.2.1.3 Computation Stage

During this stage the threads perform the main computation of each thread's portion of the application program. The computation stages of successive threads are allowed to execute concurrently. The actual amount of concurrency is determined by dependences on previous threads. Each thread first determines its data dependences on previous threads by checking appropriate data structures. In the C implementation, a thread matches the memory addresses of its data accesses to those stored in the `address` data structure. If an address match is found and the corresponding thread entry refers to a earlier thread (as indicated by a lower thread number), the thread has a data dependence. The thread must wait on the corresponding `flag` entry before it can access the data. When a thread is done with a data item, it sets the corresponding flag to thereby allow successive threads to proceed. In the Java implementation, each thread on a particular processor checks the flag structures on the other processors. If a successor thread finds a matching entry in another processor's flag array to correspond to an earlier thread, it waits until the flag is released. In both implementation, A thread must release the flag corresponding to a data item regardless of whether the data is actually accessed or not so that successor threads can proceed in case the instructions accessing the data are not executed.

To maintain non-speculative memory states, threads that are forked with control speculation perform their writes in a private memory space. However, since data dependences are enforced through shared-memory, a shared-memory copy of each dependent data item must be made on which the threads will reflect the updates. If a thread completes successfully, it copies the local updates to the actual memory locations in the shared memory during the write-back stage. Memory update during the write-back stage reduces the amount of parallelism among concurrent threads since write-backs are performed in the original sequential order. Threads forked without speculation, such as threads for a *for* loop, on the other hand, are allowed to perform their write-updates to actual

memory locations to eliminate the write-back overhead. The overhead of write-backs for speculative threads could be reduced by privatization and reduction parallelization [37] in some cases, although we have not used this approach in our implementation.

Threads which are initiated on control speculation must check during the computation stage whether the control speculation is correct. If not, it must abort all successor threads by setting the `ABORT_FLAG`. Setting this flag signals all following threads that they must abort. This flag is also set by non-speculative threads to indicate that this is the last iteration in the loop. Each thread checks the `ABORT_FLAG` just before entering the computation stage. If the thread finds that the flag has been set by a previous thread, it will bypass the computation stage. A thread that has already started its computation stage but needs to be aborted because a predecessor thread has set the `ABORT_FLAG` will either execute the instruction that causes the abort (for example, the terminating condition in a *while* loop) and so exit the computation stage, or it will complete its computation stage and will then check for the abort signal when it begins its write-back stage.

3.2.1.4 Write-back Stage

The write-back stage is used to copy-back data updates by speculative threads that completed successfully. The write-back stages must be serialized to maintain the correct memory state for speculative execution and to ensure that all successor threads do abort in case of a misspeculation. Our model uses a `wb_done` flag to force successor threads to wait until a predecessor thread has performed its write-backs. After a thread performs its writes, it sets the `wb_done` flag for its immediate successor. The next thread then becomes the head thread. Thus, successive threads will perform their writes sequentially.

The overhead of the write-back stage can be reduced by allowing each thread to go through two separate sub-stages – write-back stage I and write-back stage II. While write-back stage I must be serialized, the second stage can be executed concurrently with other threads. To eliminate output-dependences, memory locations that are dependent on previous threads perform their write-backs

in stage I in the original program order. Those writes that are independent of all other concurrent threads are written-back in stage II thereby increasing the amount of concurrency among threads.

Those threads which are to be aborted do not perform any write-backs. Instead, the head thread restores the recurrence variables to the values with which the thread started so that new threads can begin from that point of execution, if necessary. Threads that completed the computation stage without being aborted, which includes all threads preceding the first one aborted, perform their write-backs. At the end of the write-back stage, each thread checks the `ABORT_FLAG`. If it is set, no more threads need to be initiated on this processor for the current execution phase. If the flag is reset, however, the corresponding processor begins executing from the continuation stage of the next thread in sequential order.

Because of the language-level differences in C and Java, the adaption of the thread pipelining execution model on a shared-memory multiprocessor system results in some key differences in the C and the Java implementation. As already discussed earlier, the main difference is in the implementation of the run-time data-dependence checking. Since C supports pointer structures, it is necessary to use physical addresses of data elements in the run-time dependence test. Java, on the other hand, does not support pointers, which allows us to use flag structures corresponding to each data element instead of actual addresses. This makes checking for data dependences more efficient in Java, since a direct access to flag structures can be made using the index of the array element. In C, the matching process has to be done through a linear search.

Table 3.1 highlights these difference in the C and the Java implementation for the different pipeline stages. The *continuation* and the *write-back* stages are the same in both cases. The difference is in run-time data-dependence test which takes place in both the *TSAG* and the *computation* stages.

Pipeline Stage	C Implementation	Java Implementation
Continuation	compute loop control variables	compute loop control variables
TSAG	set data flags: use actual addresses in a single global flag to disambiguate memory locations	set data flags: use separate flags on different processors, corresponding to each array location
Computation	check data-dependence & compute: match corresponding entry in global flag using linear search	check data-dependence & compute: match entries in other processors' flags using direct indexing
Write-back	copy local updates in shared-memory	copy local updates in shared-memory

Table 3.1: Differences in the pipeline stage implementations in C and Java.

3.3 Execution-Time Overhead of the Speculative Multithreading Model

The implementation of the speculative multithreading technique requires the creation of threads as processes executing on multiple processors. This process creation requires a system call for thread fork-join. The parallel code also needs to execute some library calls to enforce the pipelined execution of threads. This thread creation, plus the execution of the library routines, adds to the overall parallel execution time of an application program and can be considered as the parallelization overhead.

Furthermore, the speculative execution of threads introduce some additional overhead during execution. Threads need to be aborted (and may be restarted) when they execute on an incorrect speculation. If the aborting threads do not get the abort signal before they begin their respective computation stages, these threads will be executing iterations that would not otherwise be executed in the original sequential execution of the code. We call this overhead due to incorrect speculation the misspeculation overhead.

3.3.1 Parallelization Overhead

The parallelization overhead of the speculative multithreading model has two components – the *thread creation* overhead and the *library call* overhead. The *thread creation* overhead, T_{cr}, is in-

curred when the processes executing the threads are created. This overhead depends entirely on the underlying thread fork-join mechanism. The library call overhead, T_{lib}, is incurred for each thread being executed as each thread needs to call these library routines as it goes through the different pipeline stages. T_{lib} is the sum of the overheads of each of the four pipeline stages.

The library call overhead observed by the concurrent threads is overlapped due to the pipelined execution of the threads. In a P-processor system, every set of P threads will incur an overhead of $T_{lib} + (P-1) \times T_{wb}$. However, as shown in Figure 3.4, the write-back overhead will be overlapped with the T_{lib} overhead of the next set of P threads, except for the last P threads. Thus, if there are a total of N iterations to be executed in a P-processor system with $iter$ iterations assigned to each thread, the total library call overhead for that parallel loop will be,

$\frac{N}{(iter \times P)} \times T_{lib} + (P-1) \times T_{wb}$.

For a large number of iterations, we can ignore the effect of T_{wb} and approximate the total library call overhead as, $\frac{N}{(iter \times P)} \times T_{lib}$.

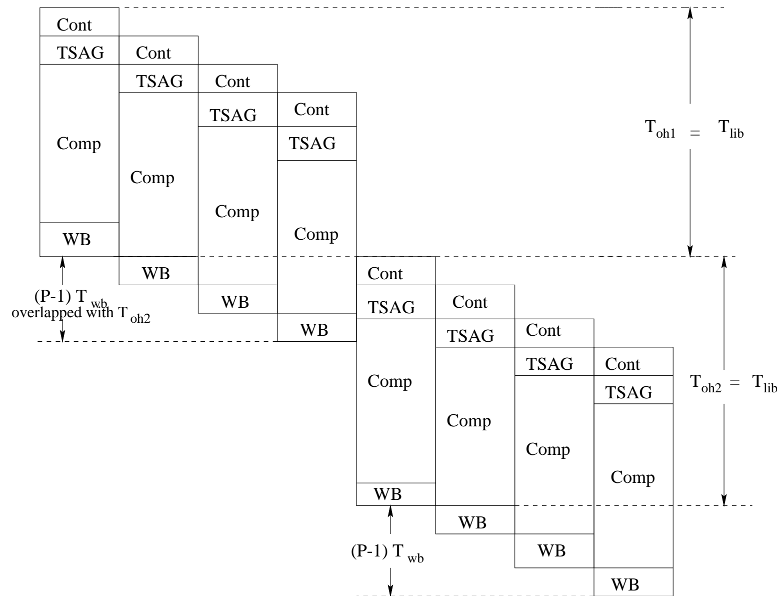

Figure 3.4: Some of the library call overhead can be overlapped in a parallel loop due to the pipelined execution of threads.

Application programs that are parallelized using this model must have sufficiently large granularity (i.e., execution time) for each parallelized portion of the code to overcome the overhead effect. The greater the amount of work involved in each thread of the resulting parallelized code, the smaller the effect of the thread creation and the library call overhead on the overall parallel execution time. To increase the amount of work per thread, one thread execution can include several loop iterations, for instance.

3.3.2 Misspeculation Overhead

The misspeculation overhead is incurred when threads executing an incorrect speculation need to be aborted and restarted. If the threads that need to be aborted receive the abort signal from a predecessor thread before starting the computation stage, the overhead is negligible. However, once a thread begins its computation stage, it does not sense the abort signal until it completes the computation stage. In the worst case, all successor threads will be in their respective computation stages when a predecessor thread identifies an incorrect speculation and sets the `ABORT_FLAG`. All these successor threads will complete their computation stages and will be ready to begin their write-back stages when they detect the abort command. Since the computation stages of successive threads are overlapped, the overhead due to incorrect speculation will be, in the worst case, the time required for the last thread to complete its computation stage plus the time it spends waiting to begin its write-back stage. If W is the work per thread done in the computation stage and P is the total number of threads active simultaneously, the worst case overhead due to misspeculation will be $W + (P-1) \times T_{wb}$. Thus, in the worst case, the cost of misspeculation is determined by the work done within each thread and the total number of threads initiated.

3.4 The Speculative Multithreading Library

The implementation of the different thread pipeline stages requires mechanisms to allow communication between threads. The shared-memory architecture used for the software implementation of the thread pipelining model enables us to carry out the communications implicitly through flags and variables in the shared memory space. A thread that needs to communicate with another thread can simply update an appropriate flag or data item in the shared memory. The other threads then can read the values from the shared memory when needed. In this section, we discuss the various library routines needed to implement the speculative multithreading model both in C and Java. We also present some example codes showing how a given sequential code is transformed to its corresponding parallel code using these library routines.

3.4.1 C Implementation

C application programs are transformed to speculative multithreading parallel code using a user-level library which is also written in C [23]. The library does not need any special operating system support. Threads communicate and synchronize through the shared-memory address space. The only system-level support needed in the parallel applications is for thread fork/join. In the following, we describe the different library routines that are used to generate parallel code using the speculative multithreading model in C.

- `initialize()`: This is an initialization routine that is invoked once for each parallel loop before the threads are created. It initializes different flags and variables in shared-memory.

- `wait_thread_active()`: This routine enforces serialization of the continuation stages of successive threads. Each thread busy waits on its `thread_active` flag to enter its continuation stage.

- `thread_init()`: This routine allows the successor thread to start its continuation stage by setting the `thread_active` flag for its successor.

- `wait_tsag_done()/release_tsag_done()`: This pair of routines enforces serialization of the TSAG stages of successive threads. Each thread makes a call to `wait_tsag_done()` to wait on its `TSAG_DONE` flag. Once a thread is done with its TSAG stage, it sets the `tsag_done` flag for its successor with a call to `release_tsag_done()`.
- `abort_thread()`: This routine is invoked by threads when they reach the loop termination condition. It sets the `ABORT_FLAG` to indicates that no more threads need to be created for this parallel loop. For speculative threads it also indicate that all successor threads must abort.
- `check_abort()`: Each thread calls this routine to check if a predecessor thread has issued an abort signal.
- `abort_next()`: This routine is invoked by an aborting thread in its write-back stage to force its successor thread to abort. This is necessary in situations where a thread that needs to abort has already start its computation stage and hence, has not sensed the abort signal from its predecessor.
- `stop_thread()`: This routine is called by a retiring thread to set the `head_thread` variable to its successor thread's *id*.
- `wait_wb_done()/release_wb_done()`: These pair of routines serializes the write-back stages of successive threads by setting and releasing the `wb_done` flag.
- The following four routines enforce run-time dependence checking:
  - `set_dependency()`: sets entries in the `address`, `thread`, and `flag` structures for each data access; this routine is invoked in the TSAG stage.
  - `check_dependency()`: determines the `flag` locations of dependent memory accesses; this dependence checking is done at the beginning of the computation stage.
  - `check_data_flag()`: waits on a `flag` variable for it to be released; this routine is invoked before each data access.

– `set_data_flag()`: releases the `flag` for the data access and is invoked after the data access.

3.4.1.1 Example Code

We use the sequential code shown in Figure 3.5 to demonstrate how the thread library routines described above can be used to parallelize a sequential code that would be difficult to parallelize with traditional compiler approaches. The code segment shown in Figure 3.5 is a *do-while* loop from the main procedure of the Unix word count utility program, *wc*. There is a loop-carried dependence caused by the update of the variable `in_word`. There are also output-dependences caused by the memory updates of the variables `lines` and `words`.

Figure 3.6 shows the corresponding parallel code using the speculative multithreading library. Each thread is assigned one loop iteration. In this code, P is the total number of processors in the system and *id* is the *process id* of the process running a thread. The threads are initiated speculatively since the loop termination condition cannot be determined ahead of time. The continuation stage begins with a call to `wait_thread_active` and terminates with a call to `thread_init`, which initiates the next thread. Each thread reads BUFFER_SIZE bytes from the input file in its local buffer, *buf*, thereby advancing the file pointer, *fd* for the next thread. A synchronization flag, `flag_in_word`, is used to enforce the data dependence on the variable `in_word`. Since there are no run-time data dependences, the TSAG stages do not involve any work. The TSAG stages are synchronized with the `wait_tsag_done` and `release_tsag_done` function calls.

Before a thread begins its computation stage, it invokes the method `check_abort` to check if a predecessor thread has been aborted. If the function returns true, which indicates that the current thread needs to be aborted, the thread will bypass the computation stage and will wait for the start of its write-back stage. Otherwise, the thread will begin its computation stage. In this stage, each thread begins by checking the loop exit condition, which in this case occurs when the end of the file is reached. If the condition is true, the thread invokes `abort_thread` and then jumps out of the

```
while ((bytes_read = safe_read (fd, buf, BUFFER_SIZE)) > 0){
    register char *p = buf;

    chars += bytes_read;
    do{
        switch (*p++){
          case '\n': lines++;
          case '\r':
          case '\f':
          case '\t':
          case '\v':
          case ' ' : if (in_word){
                        in_word = 0;
                        words++;
                     }
                     break;
          default:   in_word = 1;
                     break;
          }
    }while (--bytes_read);
}
```

Figure 3.5: An example sequential code segment from the Unix utility program *wc*.

loop. Otherwise, the thread continues with the computation of the loop body. Since a thread cannot continue until the value of `in_word` from the previous iteration is available, the necessary waiting is enforced by the *while* loop busy-waiting on the `flag_in_word` variable. Once the flag is set to its corresponding processor's id, the thread continues with the computation. The thread reads the value of `in_word` into its local memory location `my_in_word` for its own use. It updates `in_word`, if necessary, and sets `flag_in_word` for its successor thread. The thread then proceeds with the rest of the computation.

Each thread updates the variables `lines` and `words` in its own local memory copies, called `my_lines` and `my_words`. The threads arrive at, and then execute, their write-back stages sequentially. This serial order is enforced through the `wait_wb_done` and `release_wb_done` calls. A thread that successfully completed its computation stage will perform its write-back stage, which in this example is to update the variables `lines` and `words` with the thread's local copies, and will then retire using a call to `stop_thread`. The conditional statement at the beginning of the write-back stage determines whether a thread should perform its write-back or should instead abort. A thread

```
L1: /* Continuation Stage */
    wait_thread_active(id);
    my_bytes_read = safe_read(fd, buf, BUFFER_SIZE);
    if(my_bytes_read>0) chars += my_bytes_read;
    thread_init(id);

    /* TSAG Stage */
    wait_TSAG_DONE(id);
    release_TSAG_DONE(id);

    /* Computation Stage */
    if(check_abort(id)) goto L2;
    if(my_bytes_read <= 0){
      abort_thread(id); }
    else{
      while(flag_in_word != id);
      my_in_word = in_word;
      switch (buf[byte_cnt-1]) {
                case '\n':
                case '\r':
                case '\f':
                case '\t':
                case '\v':
                case ' ' : in_word =0; break;
                default  : in_word=1; break; }
      flag_in_word = (id+1)%P;
      p = buf; my_lines = 0; my_words = 0;
      do{
             switch (*p++){
                case '\n': my_lines++;
                case '\r':
                case '\f':
                case '\t':
                case '\v':
                case ' ' : if (my_in_word){
                              my_in_word = 0; my_words++; } break;
                default  : my_in_word = 1; break; }
     }while (--my_bytes_read); }

L2: /* WB stage */
    wait_WB_DONE(id);
    if(thread_abort[id]){
        abort_next(id);
        if(aborted==thread_num[id]){
          bytes_read = my_bytes_read; } }
    else{
       words += my_words; lines += my_lines;
       stop_thread(id); }
    release_WB_DONE(id);
    if(!ABORT_FLAG) goto L1;
```

Figure 3.6: The code from the example in Figure 3.5 parallelized using the speculative multithreading library routines.

that needs to be aborted forces its successor thread to abort by calling `abort_next`. All threads following the one that first aborted will also be aborted. If an aborting thread is the head thread at the time of its write-back stage, i.e., the thread to abort first, it should restore the recurrence variable, which in this case is `bytes_read`. After the write-back stage, the `ABORT_FLAG` is checked to determine if new threads need to be initiated for the current code segment. If not, the processes will return and wait for the next phase of execution.

3.4.2 Java Speculative Multithreading Library

The speculative multithreading parallelization model in Java (hereafter referred to as Java Speculative Multithreading or JavaSpMT) [24] is implemented using the standard Java multithreading mechanism. We extend the Java `Thread` class to create the threads for a parallel loop. The thread communication needed to implement the thread pipeline stages in the JavaSpMT model is carried out through library calls using flags and variables in the shared-memory space as is done in the C implementation.

We defined two Java classes that effectively implement the JavaSpMT model. The first one is an abstract class, `ParLoop`, which is an extension of the Java `Thread` class. It deals with the creation of native Java threads for speculative parallel execution of threads. For each parallel portion (e.g. loop `x`) of a Java program, a new class, `ParLoop_x`, derived from `ParLoop`, is created to carry out the parallel execution. The class `MyThread` provides the communication primitives used by the concurrent threads.

3.4.2.1 The ParLoop Class

The `ParLoop` class is used to provide an abstraction of a parallel loop that executes a portion of a code in parallel using multiple threads. An instance variable, *id*, identifies a Java thread running on a particular processor. `ParLoop` defines a class method, `StartThread()`, that initiates the parallel execution. `StartThread` uses the `start()` method of the Java `Thread` class to create native Java

threads. Once all the Java threads have been forked to execute the code in parallel, `StartThread` invokes `join()` to wait for the threads to return upon completion.

3.4.2.2 The MyThread Class

The `MyThread` class defines the flags and variables used for communication between threads as well as the methods to access and update these flags or variables for thread communication and synchronization.

- `g_init()`: This method initializes global flags and variables and is called once for each parallel loop before the threads are created.
- `initialize()`: This method initializes each thread's local flags and variables. It is invoked by each thread before beginning its parallel execution.
- `wait_thread_active()`: This method enforces serialization of continuation stages.
- `thread_init()` Each thread invokes this method to allow the successor thread to start its continuation stage.
- `wait_tsag_done()/release_tsag_done()`: This pair of method enforces serialization of the TSAG stages.
- `abort_thread()`: This method is invoked by threads to set the `ABORT_FLAG`.
- `check_abort()`: This method checks whether a predecessor thread has issued an abort signal.
- `abort_next()`: A call to this method forces the successor thread to abort.
- `get_abort()`: This method returns the status of the `ABORT_FLAG`.
- `stop_thread()`: This method sets the `head_thread` to the successor thread's *id*.
- `wait_wb_done()/release_wb_done()`: This pair of methods enforces synchronization of the write-back stages.

- Methods for run-time dependence checking:
  - set_dependency(): sets entries in the data flag structure.
  - check_dependency(): determines if there are data dependences on previous threads by matching entries in other processors' flag structures.
  - check_data_flag(): waits on a flag entry for it to be released before the data can be accessed.
  - set_data_flag(): releases the flag entry for the data access.

3.4.2.3 Parallel Code Generation

We can parallelize a sequential Java code x using the JavaSpMT classes as follows. First, a new method, run_x(), is created to contain the corresponding parallel code. The original sequential code is converted to the parallel code by inserting the thread library method calls in the appropriate places in the code similar to the way the C transformation is done. Next, a new class, ParLoop_x, is created whose run() method will execute the transformed parallel code. Finally, the original sequential code is replaced with a new piece of code that invokes the ParLoop class method StartThread() to fork Java threads on multiple processors.

Given a sequential Java code such as that shown in Figure 3.7, a new method run_1 is created which contains the parallel code for Loop_1 in Method_1(). Then the new class ParLoop_1 is created (refer to Figure 3.8). The variables v1,v2,...,vn are the parameters passed to the parallel loop. It is to be noted that the target_class variable is required only if Method_1 is an instance method. For class methods, which are identified by the static quantifier, we do not need to use the target_class instance. Instead, the run() method of the ParLoop_1 class will invoke Class_1.run_1().

Finally, we need to replace the original sequential code in Loop_1 in Method_1 by the block of code in Figure 3.9. This replaced code instantiates a ParLoop_1 class object loop_1. It then invokes the StartThread() method to create P parallel Java threads that will speculatively execute the parallel

```
class Class_1{
  ......
  [static] type Method_1(..){
    ....
    Loop_1:{            // loop to be parallelized

    }
    ....
  }
}
```

Figure 3.7: An example sequential Java code segment to be parallelized.

```
class ParLoop_1 extends ParLoop{
   Class_1 target_class;
   type1 v1;
   ...
   typen vn;

   ParLoop_1(Class_1 target_class, type1 v1, ... typen vn){
       this.target_class = target_class; // only for instance methods
       this.v1 = v1;
       .....
       this.vn = vn;
   }
   public void run(){
       target_class.run_1(id,v1,...vn);
   }
}
```

Figure 3.8: The ParLoop_1 class corresponding to Loop_1 in Method_1 shown in Figure 3.7.

```
Loop_1:  ParLoop_1[] loop_1 = new ParLoop_1[P];
         MyThread r_thread = new MyThread();

         r_thread.g_init(P);  // global initialization of thread package
         for(int i_1=0; i_1<P; i_1++)
            loop_1[i_1] = new ParLoop_1(this,v1,...,vn);
         ParLoop.StartThread(loop_1);
```

Figure 3.9: The code that replaces the original sequential code in the example in Figure 3.7 to initiate multiple Java threads for parallel execution.

```
  try{
    while ((bytes_read = fd.read (buf, 0,BUFFER_SIZE)) > 0){
      p = buf; chars += bytes_read;
      do{
          switch ((char)(p[i++])){
            case '\n': lines++;
            case '\r':
            case '\f':
            case '\t':
            case ' ': if (in_word){
                          in_word = false; words++; }
                      break;
            default: in_word = true; break;
          }
      }while (--bytes_read>0);
      ......
  }
}catch(Exception e){}
```

Figure 3.10: An example sequential code segment from the Java version of the Unix utility program *wc*.

code in `run_1` in a pipelined fashion. `StartThread()` creates the actual Java threads which then begin execution of the corresponding `run()` methods in the `ParLoop_1` class. The `this` parameter is omitted from the `ParLoop_1` constructor invocation if `Method_1` is a class method. Before creating the Java threads, the JavaSpMT thread library is initialized using an instance `r_thread` of the `MyThread` class.

3.4.2.4 Example Code

We use a Java version of the main loop in the *wc* program, shown in Figure 3.10, to demonstrate how we transform a serial Java code to its corresponding parallel code. We used the different thread library routines in the `MyThread` class (using a `MyThread` instance `p_thread`) to generate the parallel code for this serial code segment. The parallel code segment in Figure 3.11 is very similar to the C parallel code, only using Java syntax and Java thread library instead.

```
p_thread.initialize(my_id); // initialization
do{  // Continuation Stage
     p_thread.wait_thread_active(my_id);
     try{
       my_bytes_read = fd.read (buf, 0,BUFFER_SIZE);
       if(my_bytes_read >0) chars += my_bytes_read;
     }catch(Exception e){}
     p_thread.thread_init(my_id);

     // TSAG Stage
     p_thread.wait_tsag_done(my_id);
     p_thread.release_tsag_done(my_id);

     if(!p_thread.check_abort()){ // check if aborted
       // Computation Stage
       if(my_bytes_read<0)  p_thread.abort_thread(my_id);
       else{
         while(flag_in_word != my_id) Thread.currentThread().yield();
         my_in_word = in_word;
         switch ((char)(buf[my_bytes_read-1])){
              case '\n':
              case '\r':
              case '\f':
              case '\t':
              case ' ': in_word = false; break;
              default: in_word = true; break;
         }
         flag_in_word = (my_id + 1)%P; p = buf; my_lines = 0; my_words =0;
          do{
            switch ((char)(p[i++])){
                case '\n': my_lines++;
                case '\r':
                case '\f':
                case '\t':
                case ' ': if (my_in_word){
                             my_in_word = false; my_words++;}
                          break;
                default: my_in_word = true; break;
            }
         }while (--my_bytes_read>0);
      }
    }
    // Write-back Stage
    p_thread.wait_wb_done(my_id);
    if(!p_thread.thread_abort && !p_thread.check_abort()){
       lines += my_lines; words += my_words;
       p_thread.stop_thread(my_id);
    }
    else{ p_thread.abort_next();
          if(aborted==p_thread.thread_num) bytes_read = my_bytes_read; }
    p_thread.release_wb_done(my_id);
}while (!p_thread.get_abort());
```

Figure 3.11: The code from the example in Figure 3.10 parallelized using the JavaSpMT library routines.

```
try{
     code
     code
     code
}
catch(ExceptionType e}{
     exception handler code for this type
}
```

Figure 3.12: Exception handling in Java.

3.5 Parallelization in the Context of Java Language Features

Java as a programming language offers some unique features that are not present in compiled languages such as C. These language features, such as polymorphism, exception handling, and automatic garbage collection, may require special handling by the parallelization model used to parallelize a sequential Java program. In the following, we discuss how these unique features are handled by the JavaSpMT parallelization model.

3.5.1 Exception Handling

In Java, any code that may throw an exception must be explicitly handled through an exception handler. The program terminates immediately after an exception if there is no exception handler in the program to take care of the exception. The exception handler code can be placed right after the code which may result in an exception. This is achieved through a `try-catch` block in the Java program as shown in Figure 3.12:

If any of the code inside the `try` block throws an exception, the JVM immediately executes the exception handling code inside the `catch` block. The remaining code in the try block are not executed. The exception can also be thrown back to the calling method which will then catch the exception and execute any necessary exception handling code.

If exceptions are explicitly handled in the original sequential loop, the transformed JavaSpMT code must also explicitly handle those exceptions. If one thread in the parallel loop throws an

exception, the execution of the loop should terminate at that point and an appropriate exception handler should be invoked. If we allow an individual thread to explicitly handle the exception, the other threads will not know of the exception and continue executing. Thus, we need to handle the exception outside the parallel loop and at the same time have a mechanism to terminate all successor threads in case of an exception.

In JavaSpMT, when a thread executes a code that results in an exception, it invokes the `abort_thread()` method to abort all of its successor threads. The thread is not allowed to throw the exception immediately, however. If the thread terminates without releasing the `wb_done` flag for its successor, the successor will be waiting forever to enter its write-back stage and will never get the abort signal. The thread is allowed to throw the exception only after it executed the `release_wb_done()` method. This way, it is guaranteed that all the successor threads will terminate immediately after the exception is thrown. The thrown exception is caught in the corresponding thread's `run()` method [4]. The `run()` method for loop x stores the exception in an additional field `e_x` in the class ParLoop_x. The exception can be handled in the calling method that initiates the parallel loop right after the parallel loop returns. The exception may also be thrown back to another level up in the method hierarchy to handle it at an appropriate point. Figure 3.13 shows the modifications to the code in the ParLoop_1 class and the loop invocation code (shown in Figures 3.8 and 3.9 respectively) needed to handle exceptions.

If multiple exceptions occur in different threads of the same parallel loop, only one such exception is thrown back to the main thread. In Java, it is not guaranteed which one of those exceptions will be thrown back to the calling thread. Consequently, if the order in which exceptions may be thrown in a loop is essential for the correct operation of the original sequential program, then parallelization of the loop must be disabled.

```
class ParLoop_1 extends ParLoop{
    .....
    Exception e_1;

    .....

    public void run(){
        try{
             target_class.run_1(id,v1,...vn);
        }
        catch(Exception e_1){
             this.e_1 = e_1;
        }
    }
}


Loop_1:   .....

          for(int i_1=0; i_1<P; i_1++)
              loop_1[i_1] = new ParLoop_1(this,v1,...,vn);

          ParLoop.StartThread(loop_1);

          for(int i_1=0; i_1<P; i_1++)
              if(loop_1[i_1].e_1 != null)
                  throw loop_1[i_1].e_1; // the exception handler code can also be placed here

          .....
```

Figure 3.13: Modifications to parallel Java code to handle exceptions.

3.5.2 Polymorphism

As an object-oriented language, Java supports polymorphism. This feature allows methods with the same name but different signatures to be invoked based on the actual parameters used. JavaSpMT analyzes the given serial code and applies a source-to-source transformation to generate the corresponding parallel code. If the loop to be parallelized invokes some other method, that method must be transformed as well. As long as the source code for the methods referenced are available during code transformation, polymorphism will not cause any problem. For the parallel code to execute correctly in the presence of polymorphism, each corresponding method with the same name must be transformed to its equivalent JavaSpMT code. However, if the referenced class is not available during the parallel code transformation, parallelization must be disabled for that loop.

3.5.3 Garbage Collection

Java's automatic garbage collection does not have any affect on the semantics of the JavaSpMT parallel codes. The problem with garbage collection is that it may slow down the parallel execution. If one JavaSpMT thread is garbage collected in the middle of its execution while others are not, for instance, it is possible for these other concurrent threads to complete before the garbage-collected thread. In that case, these other threads must wait for the garbage-collected thread to complete before they can begin the next phase of execution. Thus, garbage collection can affect the load balance among the processors on which the threads are running, but the program will still execute correctly. Garbage collection will not be a problem for JavaSpMT codes with small memory footprints as the garbage collector typically will not even be invoked in these applications.

3.6 Conclusion

In this chapter, we have proposed a new parallelization model, called the speculative multithreading parallelization model [23, 24], that can exploit coarse-grained loop-level parallelism in shared-memory

multiprocessor systems. This parallelization model is based on the fine-grained thread pipelining execution model of the superthreaded processor architecture [44, 45, 46]. The pipelined execution of threads with run-time data-dependence checking and control speculation allows it to parallelize a wide variety of program constructs, including loops with run-time data dependences and traditionally sequential constructs (e.g. *do-while* loops). We have developed a C and a Java library that allow application programs to be parallelized using the speculative multithreading parallelization model in C and Java respectively.

Chapter 4

A Comprehensive Dynamic Processor Allocation Scheme

A wide variety of application programs exhibit significant amounts of loop-level parallelism that can be exploited for performance gains on shared-memory multiprocessor systems [6, 18]. The degree of parallelism in the application program determines how much performance speedup can be achieved through parallel execution. Parallel execution is also associated with certain overheads, such as fork-join synchronization and communication delays which adds to the overall parallel execution time. Parallel execution results in a performance speedup only if the parallel code region (e.g., a loop) can amortize the associated overhead delays. For programs with dynamically varying behavior, such as programs where the number of loop iterations changes dynamically across invocations, it is not possible to determine at compile-time the number of processors that can be used most efficiently at each invocation. Furthermore, in a standard multiprogrammed multiprocessor system, parallel applications do not perform well when there are other processes contending for the time-critical processor resource. A given number of processors must be dedicated to the parallel application to obtain the best overall performance. However, since the system load changes dynamically, the

number of processors that can be dedicated to a parallel application in a given time period also changes. An adaptive processor allocation mechanism is needed that can dynamically change the number of processors allocated to an application based on its workload and current system load.

We propose a comprehensive dynamic processor allocation scheme for shared-memory multiprocessor systems with a multiprogrammed workload that allows parallel application programs to dynamically adapt to both the program's varying behavior and the system load [25]. Based on the program's behavior, the dynamic processor allocation system determines the number of processors a parallel code region can profitably use. Then the application is allocated as many processors as are currently available, up to the maximum number it can efficiently utilize. If the program's characteristics are such that the code region cannot be profitably executed in parallel, or if there are no processors available to allow parallel execution, the system serializes the code region. The run-time system dynamically updates its parallelization decision as both the program behavior and the system load change during a program's execution.

In this thesis, the comprehensive dynamic processor allocation scheme is used to dynamically control the execution of application programs running on the Solaris operating system [56]. The application programs are parallelized using the speculative multithreading parallelization technique presented in Chapter 3. We consider both C and Java applications for dynamic parallelization. The concept of dynamic processor allocation discussed here is general enough to be applied to any parallel language paradigm, however.

The remainder of this chapter is organized as follows. First, in Section 4.1, we present an overview of the dynamic processor allocation scheme followed by a detailed discussion of the different issues in dynamic processor allocation. Section 4.2 describes the run-time system for dynamic processor allocation as implemented in Solaris for both C and Java application programs. Finally, Section 4.3 describes the different library routines that implement the dynamic processor allocation mechanism in C and Java.

4.1 Dynamic Processor Allocation Scheme

The proposed dynamic processor allocation scheme uses run-time information about program behavior and system load to determine the number of processors that can be allocated to a parallel code region for the best overall system performance. This processor allocation scheme needs to be incorporated in the run-time system that executes application programs so that the execution of parallel applications can dynamically adapt to the current program behavior and the system load. If the workload of a loop for the current invocation is not large enough to be profitably executed in parallel, the run-time system will automatically execute the loop sequentially. Loops with larger workloads will be allocated as many processors as they can use efficiently. Finally, the run-time system will allocate as many processors as can be completely dedicated to the parallel application to eliminate performance degradation due to resource contention.

Figure 4.1 gives a general overview of this dynamic processor allocation system. The run-time system first generates an execution profile of the target loop by executing it sequentially. This execution profile is later used to estimate the sequential time for the current instance of the loop. The run-time system periodically samples the system load to determine the maximum number of processors that can be used for the parallel application without overloading the system. Based on the estimate of the current loop workload (i.e., estimated sequential execution time) and the current system load, the run-time system allocates an appropriate number of processors to the parallel code region for the best possible overall system performance. Thus, the dynamic processor allocation system needs to consider the following issues:

- how to generate the parallel code for a loop when parallelization seems profitable
- what heuristics to use to make the parallelization decision
- how to estimate the sequential execution time of a loop based on some existing execution profile

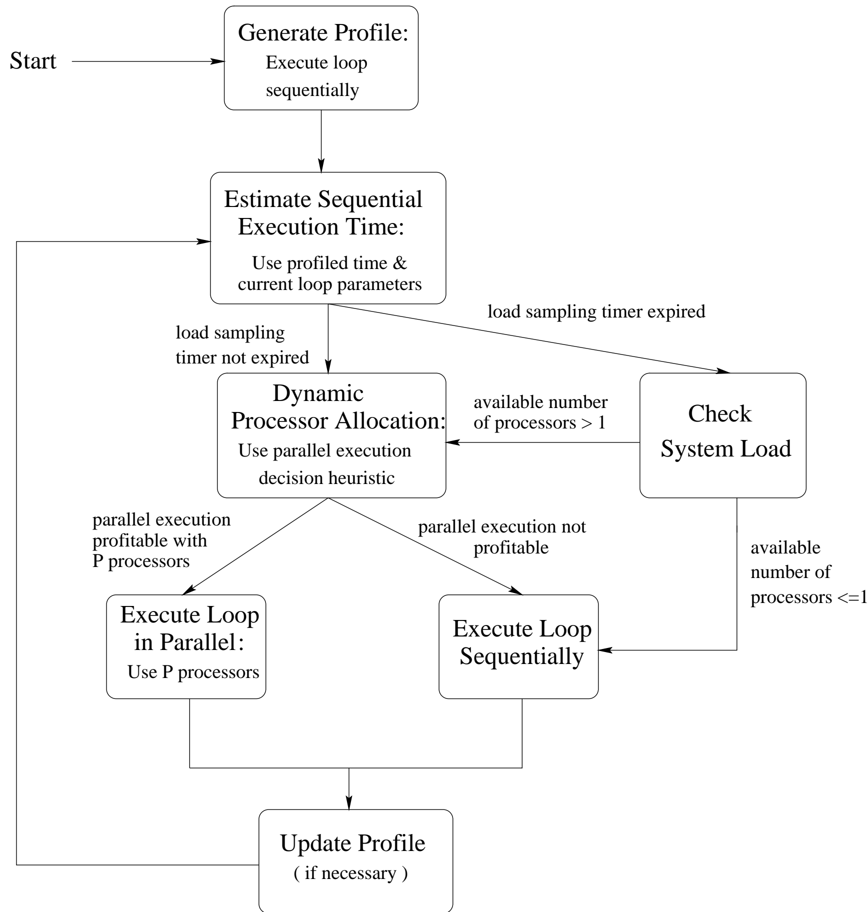

Figure 4.1: An overview of the dynamic processor allocation system.

- how to estimate the parallelization overhead
- when and how to determine the current system load

In the following, we discuss our approach to dynamic processor allocation in details.

4.1.1 Multiple Version Code

The run-time system extended with the dynamic processor allocation scheme dynamically determines whether a specific loop should be executed in parallel or sequentially depending on the varying program and system parameters. This determination requires the run-time system to have access to both the serial and the parallel versions of the target loop to dynamically select the appropriate

version for execution. There are two approaches that can be used for this purpose. One is the *multiple version code* [8] approach which requires a static transformation of the original source code to generate a version of the program that will contain both the original sequential code and the transformed parallel code. The other approach is *dynamic compilation* [3] which generates the parallel code on demand as the program executes.

We use multiple version code in our system. A source-to-source transformation is first applied to the sequential application program to generate code that contains both the sequential version and the parallel version of each target loop. The choice of which version to execute each time the loop is invoked is made using a run-time flag that is dynamically set by the run-time system. Each time the run-time system decides to execute a loop in parallel, it sets the corresponding flag variable in the program. If the loop needs to be serialized, the flag is reset so that the sequential code is executed.

While the multiple version code approach increases the code size compared to dynamic compilation, it has the advantage of allowing faster switching to the parallel code during run-time. If the run-time system instead had to generate the parallel code on demand, the execution-time overhead would be substantially larger. With the parallel code already available when the program begins execution, the only overhead of the run-time system will be due to any necessary profiling and decision making.

4.1.2 Parallel Execution Decision Heuristic

The parallel execution of a loop will result in a performance improvement compared to a sequential execution if the original sequential loop has sufficient granularity to amortize the parallelization overhead. Thus, any heuristic used to decide between parallel execution or sequential execution must consider the sequential execution time and the parallelization overhead on the target system.

The basic criterion used to determine whether execution of the parallel code will result in a performance speedup over sequential execution can be formulated using the following approximation. Let T_s be the sequential execution time of a loop and T_{oh} be the parallelization overhead with P

processors. If we assume a perfect speedup, i.e., the speedup with P processors is P, then the parallel execution time (excluding the overhead) for the loop is T_s/P. The parallel execution time on P processors, including the overhead, is then estimated to be

$$T_p \ = \ T_{oh} \ + \ T_s/P \ .$$

The parallel execution of the loop on P processors will be profitable only if,

$$T_p \ < \ Ts, \text{or}, T_{oh} \ + \ T_s/P \ < \ T_s.$$

Rearranging the terms in this inequality we get,

$$T_{oh} \ \times \ P/(P-1) \ < \ T_s.$$

Thus, any loop satisfying this inequality will provide a performance speedup when it is executed in parallel with P processors. Since the actual performance speedup on P processors is T_s/T_p, the number of processors that provides the minimum value of T_p will result in the maximum performance speedup. If the loop being considered is a partially parallel loop with a fraction f of the sequential execution time being parallelizable, the sequential time used in the above inequality will be $T_{s1} \ = \ T_s \ \times \ f$.

4.1.3 Estimating the Sequential Execution Time

The parallelization heuristic used in the dynamic processor allocation scheme requires the sequential execution time for the current invocation of the loop. Since the decision making is performed before the loop is actually executed, we can only use an estimation of the sequential execution time in the heuristic. This estimation can be done by using an execution profile of the original sequential loop. The execution profile can be generated at runtime during program execution. In this case, the first execution of any target loop should always be sequential so that the later executions can be made parallel or sequential based on this profile. Executing the loop sequentially the first time may result in some performance loss if the loop has a sufficiently large workload to benefit from parallel execution. If the loop is executed many times during program execution, however, the effect of the profiling execution will become less significant compared to the overall performance improvement through

parallel execution. Another approach to obtain an execution profile is to execute the program once sequentially on a base machine and profile the target loops [48, 49]. The base machine timings can then be scaled to the target machine by using an appropriate scaling factor for the target machine. This latter approach requires that a scaling factor for the target machine be calculated by executing some benchmark code on both the base machine and the target machine. We currently use run-time profiling for the dynamic processor allocation scheme. The system can easily be extended to use the static profiling approach.

The structure of the loop being parallelized plays an important role in the way we estimate the sequential execution time of a loop based on earlier execution profile. We use the following different estimation techniques corresponding to the loop structure.

- **Well-structured loops**. If the target loop is a well-structured loop with known termination conditions, such as a *for* loop, estimating the sequential execution time is quite straightforward. For such loops, the loop iteration count is always available before the loop is executed. If the profiled sequential execution time of a loop with N_1 iterations is T_{s1}, we can estimate the per iteration execution time to be simply $T_{iter} = T_{s1}/N_1$. Now, if the next invocation of the same loop uses a loop iteration count of N_2, the estimated sequential time will simply be $T_{s2} = T_{iter} \times N_2$. If the loop being profiled is a nested loop structure, then the per iteration execution time has to be calculated accordingly. For instance, a two-level nested loop with N iterations in each loop level has a total of N^2 iterations. Thus, the per iteration execution time in this case will be $T_{iter} = T_{s1}/N^2$.

- **Loops with indeterminate termination conditions**. Estimating the sequential execution time of loops with indeterminate termination conditions, such as *do-while* loops or *for* loops with early exits, is somewhat complicated by the fact that the loop iteration count is not available at the time of invocation. As a result, since the parallelization decision needs to be made at the time of invocation, we cannot simply use loop iteration counts for such loops.

For these loops, the parallelization decision-making is based on the observed behavior of (a number of) previous invocations of the same loop under the assumption that future invocations of a loop will exhibit behavior similar to its previous invocations. If the profiled loop has sufficiently large workload to satisfy the parallelization heuristic, the next invocation of the same loop has a high probability to have a similar workload and thus, will benefit from parallel execution. Thus, for *do-while* type loops, the run-time system records the total execution time of the sequential loop and use this profiled time as an estimation of the execution time for the next invocation. However, since loop behavior may change anytime during program execution, it is necessary to record the current behavior of the loop and update the execution profile as well as the loop classification, if necessary.

If the actual execution time of a loop classified as parallel becomes much smaller than its ideal parallel execution time (based on the profiled time), the workload of the loop has reduced significantly. It may no longer be profitable to execute the loop in parallel or the loop may profitably use a smaller number of processors. In this case, the execution profile should be updated and the parallelization heuristic should be applied on the new profile to reflect any change on the classification of the loop. Ideally, the parallel execution time of a loop with a sequential execution time T_s can be no smaller than T_s/P when running on P processors. If the parallel execution time, T_p, is smaller than the ideal value of T_s/P , we consider the loop workload to have been reduced and thus, update the loop profile with an estimation of the current sequential execution time.

If the parallel execution time of the loop, on the other hand, becomes much larger than the ideal parallel execution time, the workload of the loop has increased significantly, and so, the profile must be updated with an estimation of the current sequential execution time. We can identify an increased workload if the the parallel execution time, T_p, becomes larger than the profiled sequential execution time.

In either case, we can estimate the sequential execution time for the current instance of the loop from the recorded parallel execution time. Since the parallel execution time, $T_p = T_{oh} + T_s/P$, the corresponding sequential time will be, $T_s = (T_p - T_{oh}) \times P$. The loop profile will be updated with this estimated value of T_s.

4.1.4 Determination of Parallelization Overhead

The parallelization overhead of a parallel application program includes such costs as fork/join cost and synchronization cost. The fork/join and thread synchronization costs depend on the thread library implementation of the operating system on a particular hardware platform. Furthermore, the parallelization technique used may add some overhead due to additional library calls needed to enforce the parallel execution.

For any given system, we can determine the parallelization overhead by running an empty parallel loop on that system. In a P-processor system, we must determine the overhead for each different processor configuration, e.g., 2,3,4, ... ,P processors. Since the parallelization overhead often increases as more threads are introduced in the system, the same code that exhibits speedup with 2 processors may not do so with 4 processors, for example. Consequently, we need to obtain overhead values for each different parallel machine configuration. These overhead values are determined using a dedicated system so that the values represent the overhead when all the processors are available for the current application. In our current implementation, we determine the overhead values once for each system and each configuration. These values are then used by the run-time system to make its parallelization decision.

4.1.5 System Load Determination

In a multiprogrammed system, it cannot be guaranteed that the number of processors required for a parallel program's execution will always be available. If there are more processes in the system than the number of available processors, processes will be executed using a time-shared approach.

This sharing of the critical resource can significantly degrade the performance of the application programs. Thus, in a multiprogrammed multiprocessor system it is important to allocate as many processors as possible while not overloading the system.

The operating system kernel maintains the overall system load information. Information about current processes, including the execution state of each process, the recent cpu usage of a process, and the number of threads used by each process, can be obtained by reading the kernel data structure. Ideally, we would like to get the current system load information each time a parallel code region is executed so that the program is always allowed to use as many processors as are currently available. However, obtaining the system load information by reading kernel data structures introduces additional overhead to the program's execution. If the system load is sampled for every invocation of a loop, the overhead may be quite significant, especially for loops with smaller workloads. Thus, the run-time system should sample the system load after certain time intervals. On the other hand, if the sampling interval is too long, the system load may change in the mean time and the program will be executing with too few or too many processors in that period. Thus, it is important to select a load sampling interval that would not have significant effect on the parallel execution time but still provide an accurate estimate of the current load.

4.2 Run-time System

Using the parallelization heuristic discussed in the previous section together with the current system load information we can dynamically allocate an appropriate number of processors to a parallel application program that would result in the best overall system performance. We have implemented the dynamic processor allocation scheme in the run-time system for both parallel C and Java application programs. The implementation is done at the user-level using special library support. For the C implementation, the library calls need to be included in the application program itself. The higher level of abstraction in Java, through the Java Virtual Machine (JVM), however, allows us to

incorporate the library support for dynamic processor allocation in the JVM code. Thus, parallel Java applications programs can be dynamically controlled by the modified JVM requiring little or no instrumentation in the program code. We implemented the dynamic processor allocation scheme in the interpreter of the JVM.

The difference in the parallel program execution model in C and Java (e.g., execution-time performance, thread implementation), in some cases imposes different requirements on the implementation of the dynamic processor allocation mechanism in the run-time system. In the following, we discuss the details of the run-time system emphasizing the differences in implementation for C and Java as necessary.

4.2.1 Multiple Version Code Generation

As discussed in the previous section, we use the *multiple version code* approach to generate the parallel code for the dynamic processor allocation scheme. We apply a source-to-source transformation to the original sequential application program to generate the speculative multithreading parallel code for each target loop while keeping the original sequential loop as it is. A run-time flag is used to conditionally select the appropriate version for execution dynamically. Currently, we apply the transformation manually. However, a restructuring compiler can be developed to apply the transformation automatically.

4.2.2 Parallelization Overhead of the Speculative Multithreading Model

The parallelization decision in the dynamic processor allocation scheme is based on the overhead associated with the parallelization model used. Since our target parallelization model is the speculation multithreading technique, the run-time system uses its parallelization overhead. As discussed in Chapter 3, the overhead for the speculative multithreading execution model has two main components. The *thread creation overhead*, T_{cr}, is the result of forking multiple threads on physical processors. The *library call overhead*, T_{lib}, is the execution time overhead for calling the library

routines that enforce the pipelined execution of concurrent threads. The library call overhead is incurred by each of the concurrent threads, although it is partially overlapped as a result of the pipelined execution of the threads. The overall cost for the library call for a loop with N iterations can be approximated as,

$$T_{lib}(total) = \frac{N}{(iter \times P)} \times T_{lib}$$

where P is the number of processors and $iter$ is the number of iterations per thread. Thus, the total overhead of the parallel execution for a loop with N iterations is,

$$T_{oh} = T_{cr} + T_{lib}(total).$$

In the C implementation, the run-time system calculates T_{oh} for a given processor configuration using predetermined values of T_{cr} and T_{lib} for the corresponding configuration measured on the target system. In Java, the value of T_{lib} is usually quite small compared to T_{cr} and can be safely ignored. Thus, in the Java implementation, we approximate the parallelization overhead as $T_{oh} = T_{cr}$.

4.2.3 System Load Determination

The run-time system periodically samples the system load by reading kernel data structure. The load sampling interval is controlled by the user and must be provided to the run-time system during initialization. The run-time system determines the system load at the beginning. Thereafter, the load is checked each time before the parallelization decision is made provided the load sampling interval has expired.

The Solaris operating system provides system call interface to get such information as the average load over the last 1, 5, and 15 minutes interval. But this load information is too coarse-grained to be used by our system. We need a more up-to-date load information for the processor allocation decision to be reflective of the current system load. The operating system also provides an interface to read some data structures that it maintains for each process. The process status data structure provides the recent cpu usage of the corresponding process as well as the number of light-weight processes (*lwps*) or threads created by that process. The cpu usage of a process is the percentage of

total cpu resource utilized by that process. For example, an active process executing on 2-processors will consume 25% of total cpu time on an 8-processor system. The run-time system can calculate the total percentage of the processor resources being used by all the processes currently running. This percentage can then be translated into the number of processors being used by multiplying the total number of processors with the fraction of the processor resources being used. For example, if there are 8 processors in the system and the total percentage of processor resources used is 50%, the effective system load is $0.5 \times 8 = 4$. Another way of estimating the system load is to determine the number of threads (or *lwps*) used by each active process. For example, if there are 2 active processes in the system, one using 1 thread while the other using 4 threads, the current system load will be 5.

In the C implementation of the dynamic processor allocation scheme, the run-time system reads the process status files in the `\proc` directory to get the current system load. Most of the parallel loops in the application programs we used in our performance evaluation executed only for a few seconds in each invocation. We found that the cpu usage field of the process status data structure does not reflect the actual usage for such small duration of execution. Thus, adding the cpu usage values for all the active processes does not give us an accurate estimate of the system load. So, instead, we decided to get the system load from the total number of threads (or *lwps*) in use by the active processes.

While estimating system load this way is accurate, it requires us to preset the initial number of threads a parallel application is allowed to use under the dynamic processor allocation scheme when multiple parallel applications are running simultaneously. Otherwise, one of the parallel applications may be monopolizing all the available processors. For example, when two parallel applications are executing simultaneously, it is unlikely that both processes will check the system load exactly at the same instance of time. Whichever checks the system load first will find one other processor in use (by the other process) and, thus, grab the remaining $P - 1$ processors for its parallel execution. The other process will then find that all remaining processors are busy and, thus, begin execution in sequential mode. Depending on the program characteristics, it may happen that the second

process never gets to execute in parallel while the first one is utilizing all the available processors. By initially setting the number of threads a process can use, we can ensure fairness of processor allocation among the different concurrent processes in the system.

Because of the complexity of the JVM code, we could not directly incorporate the system call to access process status files in the JVM code. Instead, we opted for a simpler implementation which requires executing a shell script that determines the system load using the Unix `top` command. The `top` call provides the current state and the cpu usage of each process in the system. Since Java programs execute sufficiently longer in the interpreted mode of the JVM, we found that the cpu usage values do reflect the actual usage. Thus, in this case, we can determine the system load based on the total cpu usage of all the active processes.

The advantage of using this approach is that we do not need to preset the number of threads when multiple parallel applications are executing in the system. If there are two parallel application running, for example, each using all P processors, each process will actually be able to utilize only 50% of cpu resource. So, the next time the run-time system will sample the load it will find that the other process is utilizing 50% of cpu and thus, will allocate half the number of processors to its application program. Using a Unix utility such as `top` does result in significant overhead for load determination. If the system call needed to get the load information can be included in the JVM code, the overhead would become negligible.

4.2.4 Dynamic Processor Allocation Algorithm

The run-time system uses the algorithm shown in Figure 4.2 to make its processor allocation decision based on loop characteristics and system load information. The run-time system goes through three different stages. The *initialization* is done once at the beginning of a program's execution. The *processor allocation* stage precedes the execution of a target loop to determine the number of processors the loop can efficiently use. The *profiling* of a loop is done after a loop completes its execution to update its execution profile.

Initialization (executed once when the program begins execution):

1. Initialize loop profile data structures, set load sampling interval , and get parallelization overhead for the current system.
2. Obtain the current system load, $current_load$.

Processor Allocation (executed each time a loop is invoked):

1. If this is the first invocation, execute the loop sequentially.
   Otherwise, estimate the sequential time, T_s, using the previously profiled time, T_{prof}:
   - *for* loop: $T_s = T_{prof} \times N$
     where, N is the current loop iteration count.
   - *do-while* loop: $T_s = T_{prof}$
2. If the load sampling interval has expired, obtain the current system load and reset the load sampling interval timer.
3. Determine the available number of processors, $avail_proc$, as
   $avail_proc = num_of_proc - current_load$, where num_of_proc is the total number of processors in the system.
4. If $avail_proc \leq 1$, execute the loop sequentially; otherwise, continue.
5. For $i = 2$ *to* $avail_proc$ repeat:
   Estimate parallel execution time, $T_p(i) = (T_s \times f/i) + (T_s \times (f-1)) + T_{oh}(i)$ where, f is the fraction of the loop execution time that is parallelizable.
6. Determine the minimum parallel execution time:
   $T_p(min) = minimum(T_p(2), T_p(3), ...T_p(avail_proc))$;
   Set $proc = i$ such that $T_p(i) = T_p(min)$.
7. If $T_p(min) < T_s$ execute in parallel using $proc$ processors; otherwise execute sequentially.

Loop Profiling (executed each time a loop completes its execution)

1. If sequential loop execution, update profile using current sequential execution time, T_s:
   - *for* loop: determine per iteration execution time, $T_{iter} = T_s \,/\, N$, where N is the number of loop iterations;
     current profile $T_{prof} = average(T_{prof}(prev), T_{iter})$.
   - *do-while* loop: $T_{prof} = T_s$.
2. If parallel execution, update profile for *do-while* loops in the following cases:
   - case 1: loop workload has reduced significantly:
     if $T_p < \frac{T_{prof}}{P}$,
     for two consecutive runs, where T_p is the current parallel execution time using P processors, update profile,
     $T_{prof} = (T_p - T_{oh}) \times P$
   - case 2: loop workload has increased significantly:
     if $T_p > T_{prof}$
     update profile, $T_{prof} = (T_p - T_{oh}) \times P$

Figure 4.2: Dynamic processor allocation algorithm.

4.2.4.1 Initialization

When a program begins execution, some initialization is needed to set up the processor allocation mechanism. First, the data structures for loop profiling must be initialized with the corresponding loop characteristics information. Information such as, the type of the loop (*for* or *do-while*), the number of loop nesting, the fraction of the loop workload that can be executed in parallel, must be known before the loop profiling and processor allocation can begin. These information are obtained using static analysis of the program. The run-time system also needs to know the target system configuration, i.e., the maximum number of processors available, and the load sampling interval. In addition, the run-time system must know the parallelization overhead values on the target system for each possible processor configuration. In the C implementation, this information is passed to the run-time system through program instrumentation. In the Java implementation, the initialization information is provided in an external file which the JVM reads during startup.

In the Java implementation, in addition to the these initial information, the JVM needs to know the loop parameters for each target loop. Since the JVM has access to the loop parameters in the form of local or global variables in the program stack, we need to provide the run-time system with the stack variable identifiers for such loop parameters as the program variable for the loop iteration count, the variable determining the number of threads to create for the parallel loop, and the variable selecting the appropriate version of the code. The stack identifiers for these parameters can be obtained through a static analysis of the corresponding Java class file.

During system initialization, the run-time system also samples the system load and starts the load sampling interval timer. The load sampling interval is controlled by the user and provided as part of the initialization data. After the initialization, the system load is sampled each time the load sampling interval timer expires. The run-time system resets the interval timer each time the load is sampled.

4.2.4.2 Processor Allocation

The first invocation of each target loop is always executed sequentially to obtain a sequential execution profile for the loop. After the first invocation of each loop, the runtime system dynamically determines if the loop should be executed in parallel each time it is subsequently invoked. For each invocation, the run-time system first estimates the sequential execution time using the profiled execution time and the current loop parameter values. Additionally, the current system load information is queried to determine the number of processors available. If the available number of processors is less than or equal to one (i.e., all the processors are busy), the loop is executed sequentially. If there are two or more processors available, then the loop is considered for parallel execution.

The parallelization decision for this invocation is made using a combination of the estimate of the parallel execution time and the previously measured overhead values for the available number of processors. First, the parallel execution time for the loop is estimated for each available processor configuration, as shown in Step 5 of the processor allocation algorithm in Figure 4.2. The processor configuration, *proc*, that is expected to produce the minimum estimated parallel execution time is then determined. Finally, this minimum parallel execution time is compared to the estimated sequential execution time. If the sequential time is greater than the estimated parallel execution time, parallel execution of this loop on *proc* processors will be profitable. The run-time system then allocates *proc* processors to the loop for parallel execution. Otherwise, the loop is executed sequentially.

4.2.4.3 Loop Profiling

Each time a target loop is executed sequentially, the execution profile for the loop is updated with the current execution time. If the loop has known loop bounds, i.e., a *for* loop, the per iteration execution time is determined by calculating an average over the current execution time and the recorded execution time. If the loop is a *do-while* type loop, the existing profile is replaced with the

current execution time.

For *do-while* loops, the profile needs to be updated, even after a parallel execution, if the loop behavior changes significantly. If the parallel execution time is much smaller than the ideal parallel execution time for the profiled loop, the loop workload has reduced significantly. It may no longer be profitable to execute the loop in parallel or the loop may profitably use a smaller number of processors. However, once a loop has been classified as sequential, a change in classification to parallel will only occur after the loop workload has increased significantly (which will be identified by an increased sequential execution time). If the loop behavior changes from parallel to sequential and back to parallel again, then changing the classification based on a single sequential behavior will cost us the performance of the latter parallel execution. So, the run-time system does not immediately change the execution profile in such cases. The run-time system waits for two such consecutive loop behavior before updating the profile. In our current implementation, we identify a reduced workload by checking if the parallel execution time, T_p, with P processors is smaller than T_{prof}/P.

If the parallel execution time, on the other hand, is much larger than the expected parallel execution time, the loop workload has increased significantly. If the current parallel loop is not using all the available processors, it is possible that with the increased workload the loop can utilize more processors profitably. For example, consider a loop with only sufficient workload to amortize the overhead for a 8-processor configuration in a 16-processor system. The run-time system will allocate only 8 processors to this loop. Now, if the loop workload increases significantly (e.g., if the number of iterations increases), the workload may be large enough to amortize the overhead corresponding to all 16 processors. In this case, the loop will be able to improve performance by executing on more processors. Since updating the profile in this case will not make any change in the classification of the loop, the run-time system is allowed to update the profile immediately. The run-time system identifies an increased workload when the parallel execution time, T_p, becomes greater than the profiled sequential execution time, T_{prof}.

4.3 Library Routines for Dynamic Processor Allocation

The dynamic processor allocation mechanism is implemented at user level using special library support. In this section, we describe the library routines used by the run-time system to dynamically control parallel execution of application programs based on program and system behavior.

4.3.1 C library

We developed a library written in C for dynamic processor allocation in C programs. Application programs are instrumented so that these library calls are invoked as the program executes. Each parallel loop in a C application program has an associated data structure that maintains its execution profile as well as the parameters controlling its parallel execution. Each loop passes its corresponding data structure as a parameter to the dynamic processor allocation library routines. In the following, we discuss these library routines.

- `da_init()`: This is an initialization routine that is executed once at the beginning of a program. This routine sets some system parameters with values provided by the user. These parameters include the maximum number of processors available, the preset load used for multiple parallel runs, and the load sampling interval in seconds. The system load is sampled during this initialization (using a call to the `get_load()` routine) and the load sampling interval timer is reset. The overhead values for the current system are also read at this time.

- `loop_init()`: This is an initialization routine for a parallel loop. For every parallel loop in the program, this routine must be invoked once before the loop begins execution. For every loop X, this routine sets the loop `type` ('0' for *for* loops and '1' for *do-while* loops), the loop `order` (or the loop nesting), and the `fraction` of the loop that is parallelizable, in the loop data structure `loop_X`. This routine also sets the initial values of other variables in the `loop_X` data structure.

- `get_load()`: This routine is invoked by the runtime system to get the current system load. It reads the process status data structures in the \proc directory and returns the number of processors currently in use.

- `profile_loop()`: This routine updates the profile data for loop X using the current execution time, `exec_time`, the current loop iteration count, `N`, and the current iteration per thread, `iter`. The current execution time (in micro seconds) is obtained by inserting timer routines before and after the loop in the application program.

- `parallelize()`: This routine performs the actual processor allocation function for loop X. Based on the existing profile stored in the loop_X data structure, current loop iteration count (`N`), and iteration per thread (`iter`), the routine makes its parallelization decision. For *do-while* loops, the profiled iteration count is used since the actual number of loop iterations is not available at this time. During the decision making, the system load is sampled using `get_load()` if the load sampling timer has expired.

A source-to-source translation is first applied to the application program to generate the parallel and the sequential versions of each loop. Next we insert the dynamic processor allocation library calls in the source code to dynamically select the appropriate version for execution. The code segment in Figure 4.3 shows how the library calls are used to dynamically allocate processors to a loop X.

In this example, `loop_X` is the loop data structure for loop X. The `method_count` field indicates how many times the loop has been invoked so far. The `proc_used` field determines the number of processors to use for the current invocation, and is set by the run-time system. The `spec_par` flag controls the selection of the parallel or sequential code for execution. If the current invocation is the first invocation of the loop (which will be the case when `method_count` = 0), the sequential code is executed. This sequential execution is timed using timer functions before and after the loop body. The call to the `profile_loop` routine updates the loop profile with the current execution time, loop iteration count, and iteration per thread values. If the loop is a *do-while* type loop, additional

```
  /* initialization */

  da_init(...)
  loop_init(&loop_X, ...)

  ...
  ...

loop X:
        spec_par = 0;

        if(loop_X.method_count)
            spec_par = parallelize(&loop_X, N, iter);

        if(spec_par){

            p = loop_X.proc_used;
            start_time = timer();

            /* record loop count N for do-while loop */

            parallel loop

            end_time = timer();
        }

        else{
            start_time = timer();

            /* record loop count N for do-while loop */

            sequential loop

            end_time = timer();
        }

        profile_loop(&loop_X, N, iter, end_time - start_time);

  ....
  ....
```

Figure 4.3: Program instrumentation required to incorporate the dynamic processor allocation mechanism in a C code.

instrumentation is required in the source code to count the number of iterations actually executed.

On subsequent executions, a call to the `parallelize()` routine determines if the current execution should be parallel or sequential. If the execution is parallel, the `spec_par` flag is set to '1' and the `proc_used` field of `loop_X` is set to the number of processors the loop is allowed to use. If the loop is a *do-while* type loop, the parallel execution needs to be timed as well. If the `spec_par` flag is reset, it indicates that the current execution should be sequential. In any case, the `profile_loop()` routine will be called. For sequential execution, the profile data of the loop will be updated with the current parameters. For parallel execution of *do-while* loops, the profile data will be updated based on the current parallel execution profile. The call to `profile_loop()` does not have any affect for parallel execution of *for* loops.

4.3.2 Java Library

In Java, the dynamic processor allocation library is incorporated in the Java Virtual Machine (JVM). Thus, the program code itself requires very little instrumentation as shown in Figure 4.4. We have extended the interpreter in JDK version 1.2 for Solaris [55] with the library calls so that the JVM dynamically decides how many processors to allocate to a loop whenever a target loop is invoked. It is possible to turn off the dynamic processor allocation mechanism in the JVM if the standard mode of Java execution is desired.

The JVM maintains a profile data structure for each target loop which is initialized when a class file is first loaded. The parameters for this initialization is read by the JVM from an external file. The data in the file is obtained through static analysis of the Java bytecode. To allow the JVM identify loops that will use the dynamic processor allocation mechanism, we use the suffix `_SPMT` to a method name. Only method names with this suffix will be profiled and the other methods will be executed in standard mode.

The following routines are used by the JVM to dynamically control the parallel execution of target loops.

- `read_init_info()`: This routine reads the values for the initialization parameters from an external file called `input`. It first gets the system initialization values including the number of processors in the system and the load sampling interval (in milliseconds seconds). It then reads the initialization parameters for each target loop. These parameters include the loop type, loop order, fraction of code parallelizable, and the stack variable numbers for different loop variables. These parameter values are set in the corresponding loop profile data structure. This routine is invoked only once before the JVM begins executing the bytecodes. At this time, the system load is sampled by calling `get_load()` and the load sampling interval timer is reset.

- `read_overhead()`: This routine initializes the overhead table with the values read from an external file, `overhead`. The `overhead` file contains the parallelization overhead values as measured on the target system.

- `get_load()`: This routine samples the system load by executing a shell script that invokes the Unix `top` command to get the cpu usage percentage for the processes currently executing.

- `profile_method_entry()`: This routine is executed by the JVM each time a method containing a target loop is invoked. It sets the current loop parameter values in the loop data structure and starts a timer to measure the loop execution time.

- `profile_method_exit()`: This routine is executed each time the target method finishes its execution. It updates the loop profile with the current execution time.

- `parallelize()`: This routine makes the parallelization decision based on the current loop profile and the current system load. It samples the load before making the processor allocation decision in case the load sampling interval timer has expired.

- `fix_processors()`: This routine is executed when the `parallelize()` routine determines that the current execution should be parallel. It sets the run-time flag in the target method that controls the parallel execution. It also sets the variable in the method that determines the

```
method_X_SPMT(.....){

     int P; // number of processors
     boolean spec_par = false; // code selection flag
     int N,....

     ....
     ....

loop_X:
           if(spec_par){
              parallel code
           }
           else{
              sequential code
           }
}
```

Figure 4.4: Program instrumentation required in a Java code to enable dynamic processor allocation.

number of processors to use.

All the above library routines are part of the JVM code. So the actual program code does not need any instrumentation for the dynamic processor allocation mechanism to work. However, the program code should contain both the parallel and sequential version of the loop in the same method the access to which must be controlled by a run-time flag. The example code in Figure 4.4 shows a method that can use the dynamic processor allocation mechanism in Java.

In this code, `P` is the number of processors and `spec_par` is the run-time flag. The Java run-time system will set both these variables to appropriate values when `method_X` is invoked. During system initialization, the stack variable numbers for `P` and `spec_par` as well as the variable number for the loop iteration count `N` must be supplied to the JVM so that at run-time it can use and set the appropriate variables in the program stack.

When the JVM begins execution, it first looks for the `input` and `overhead` files in the source code directory. If the files are not present, the JVM turns off dynamic processor allocation and the application program is executed using standard Java interpreter instead. If dynamic processor allocation is enabled, the JVM will use current loop parameter values and profiled timings to de-

termine the number of processors that can be allocated to a loop each time a method containing a target loop is invoked. It will set the program parameters accordingly to allow the loop to execute sequentially or in parallel.

4.4 Conclusion

In this chapter, we have proposed a comprehensive dynamic processor allocation scheme for multiprogrammed shared-memory multiprocessor systems. It uses dynamic program behavior and system load information to allocate an appropriate number of processors to a parallel application program to provide the best possible performance for the application program as well as the overall system. This processor allocation mechanism can be applied to any parallel language paradigm. In this thesis, we have applied this scheme to application programs parallelized using our speculative multithreading parallelization model only. The dynamic processor allocation scheme is implemented both in C and Java to dynamically control the parallel execution of application programs on shared-memory architecture using the Solaris operating system.

Chapter 5

Performance Evaluation

The speculative multithreading parallelization model exploits loop-level parallelism on shared-memory multiprocessor systems to improve execution-time performance. The speculative thread pipelining execution model with run-time data-dependence checking allows this model to extract parallelism from a wide variety of application programs that cannot be parallelized using standard techniques. In this chapter, we evaluate the performance of the speculative multithreading parallelization model by applying it to a number of real application programs. A variety of loop constructs including standard DOALL and DOACROSS loops, as well as loops with run-time data dependences and traditional sequential constructs, are parallelized to show the wide range of applicability of this parallelization model.

The comprehensive dynamic processor allocation scheme allows parallel application programs to dynamically adapt to the current execution environment. The number of processors allocated during each invocation of a parallel loop is determined by the loop granularity and the number of available processors. By allocating as many processors as a parallel loop can fully utilize, this processor allocation scheme can provide the best performance for the application program even on a dedicated system. By allocating as many processors as can be utilized without contending for processors, this scheme can further improve the overall system performance in a multiprogrammed

environment compared to the standard time-sharing mechanism. We evaluate the effectiveness of this dynamic processor allocation scheme by applying it to the application programs parallelized using the speculative multithreading model.

In the remainder of the chapter, Section 5.1 describes the system configuration of the shared-memory multiprocessor systems used in the performance evaluation. Section 5.2 gives an overview of the application programs used in the experiments. Section 5.3 presents the performance results of the speculative multithreading model when applied to C and Java applications followed by a comparison to the performance of some related techniques. In Section 5.4, we analyze the performance results for the dynamic processor allocation scheme. Finally, we summarize the performance results in Section 5.5.

5.1 Test System

We evaluated the performance of the speculative multithreading parallelization model on a SGI Challenge shared-memory multiprocessor system. The system has eight 196 MHz MIPS R10000 processors with MIPS R10010 FPU, 32 Kbytes of separate data and instruction caches, 2 Mbytes of secondary unified cache, and 1024 Mbytes of 8-way interleaved shared memory. This multiprocessor system uses the IRIX version 6.2 operating system. The Java Virtual Machine on this system is the IRIX6.2 implementation of JDK 1.1.6 with a just-in-time (JIT) complier and a native thread package.

The comprehensive dynamic processor allocation scheme was evaluated on a 6-processor Sun SparcServer 1000 system with a processor clock rate of 50 MHz. Each processor in this system has on-chip 20 KBytes of instruction cache and 16 KBytes of data cache. The system has a total of 128 Mbytes of memory. This system uses Solaris version 2.6 as its operating system. The dynamic processor allocation scheme for Java applications was implemented in the Java 2 run-time system for Solaris [55] and the test programs were executed in the interpreted mode of the Java Virtual

Machine.

5.2 Test Programs

Five application programs are considered to evaluate the performance of the speculative multithreading parallelization model and the comprehensive dynamic processor allocation scheme proposed in this thesis. These application programs were first parallelized using the speculative multithreading parallelization model. Later we applied the dynamic processor allocation mechanism to the parallelized programs. The application programs are:

- *Matrix Multiplication (mat_mul)*
- *Gaussian Elimination (gauss)*
- *Perfect Club benchmark, MDG*
- *Unix word count utility program, wc*
- *Spec2000 FP benchmark, 183.equake (quake)*

Matrix Multiplication

The *mat_mul* program multiplies two NxN matrices and stores the result in another NxN matrix. The matrix multiplication loop is a three-level loop nest with the outer loop being completely parallelizable. The problem size of *mat_mul* is determined by the matrix dimension, N.

Gauss

Gauss solves an NxN system of linear equations. The *forward elimination* loop of *gauss* is a three-level loop nest that typically takes about 98-99% of the overall program execution time. While the outermost loop is sequential, the two innermost loops can be fully parallelized. The problem size of *gauss* is also determined by the matrix dimension, N.

MDG

The *MDG* program from the Perfect Club Benchmarks [12] performs dynamic molecular simulation of flexible water molecules. In *MDG*, we considered *Loop 1000* of the *INTERF* subroutine for parallelization. Loop 1000 is a two-level loop nest that has loop-carried dependences due to reduction operations on shared data variables. The inner loop has data dependences on fewer data variables than the outer loop, but it also has a smaller grain size. The outer loop, on the other hand, has cross-iteration dependences on more data variables than the inner loop, but it has a much higher grain size. Consequently, we selected the outer loop for parallelization. The execution of the *INTERF* routines takes about 88-90% of the total execution time. MDG has a fixed problem size.

Wc

The unix word count utility program, *wc*, determines the number of characters, words, and lines in a given input file. The main computation of *wc* consists of a *while* loop that reads a block of bytes from the input file and counts the number of lines, words, and characters in that block. The loop continues until the end of the file is encountered. The loop has cross-iteration data dependences on three shared data variables. Standard loop parallelization techniques cannot parallelize *while* loops since the number of iterations that will be executed is determined dynamically at run-time. The control speculation provided by the speculative multithreading model allows these types of loops to be easily parallelized, however. The problem size of this program is determined by the size of the input file.

183.equake

The SPEC2000 floating point benchmark 183.equake simulates the propagation of elastic waves in large, highly heterogeneous valleys to recover the time history of the ground motion everywhere within the valley due to a specific seismic event [57]. Computations are performed on an unstructured mesh that locally resolves wavelengths, using a finite element method. The *simulate* loop in this

benchmark is a DOACROSS loop with an early exit condition and run-time data dependences caused by array accesses through subscripted subscripts. The non-deterministic nature of the loop termination, as well as the presence of run-time data dependences, make this loop an ideal candidate for the speculative multithreading parallelization model.

5.3 Performance Evaluation of the Speculative Multithreading Parallelization Model

In this section, we evaluate the performance of the speculative multithreading parallelization model. First, we examine the execution time overhead of this model in the C and the Java implementations on the SGI Challenge multiprocessor system. We then evaluate the performance of the C implementation of this model [23] when applied to the application programs described in Section 5.2. We also evaluate the performance of the Java implementation (JavaSpMT) [24] by parallelizing the Java versions of the same five application programs.

5.3.1 Parallelization Overhead on the Test System

As discussed in Chapter 3, the speculative multithreading parallelization model has some associated overhead – the *thread creation overhead*, T_{cr} and the *library call overhead*, T_{lib}. We measured T_{cr} and T_{lib} on the SGI Challenge multiprocessor system for both the C and the Java implementations of the speculative multithreading parallelization model.

	Average (seconds)	**Standard Deviation** (seconds)
2 processors	0.6374	0.003
4 processors	0.7456	0.014
8 processors	0.8606	0.021

Table 5.1: Thread creation overhead (in seconds) for 2, 4, and 8 processors on the SGI Challenge system.

In the C implementation, the processes executing the threads are created using a call to the `m_fork` C library routine. For a P-processor system, the `m_fork` routine creates P-1 other processes that execute the given code in parallel with the calling process. The processes execute the subprogram and wait until they all return, at which point the `m_fork` call returns. The other threads are left busy-wait spinning until the master thread calls `m_fork` again. The overhead of creating the additional processes is done in the first call to `m_fork` and is not repeated. Thus, the thread creation overhead (T_{cr}) in the speculative multithreading model is incurred only once for the first invocation of `m_fork`. Table 5.1 shows the average values of T_{cr} on the test system for 2, 4, and 8 processors. As there is a fixed overhead per thread created, the T_{cr} overhead increases with the number of threads created. Since T_{cr} is incurred only once during program execution, its effect can be ignored for loops which execute a large number of times in the program.

Continuation Stage	9 + time to compute recurrence variables
TSAG Stage	8.25 + 0.06 × # of data accesses per thread
Computation Stage	Run-time dependence checking overhead per data access: 0.06 × # of processors used × # of data accesses per thread
WB Stage	4.5 + t(data location) × # of bytes written t(L1 cache) = 0.020 t(L2 cache) = 0.023 t(Memory) = 0.040

Table 5.2: Overhead (in microseconds) associated with the different pipeline stages of the C implementation of the speculative multithreading model as measured on the test system.

The library call overhead, T_{lib}, is incurred for each thread being executed as each thread goes through the different pipeline stages. Table 5.2 shows the overhead associated with each of the four pipeline stages in the C implementation. T_{lib} is the sum of these four pipeline stage overheads. Note that the overhead for the computation stage is for the run-time data-dependence test which can be performed concurrently by successive threads. This overhead increases with the number of dependent data accesses in each thread, as well as with the number of concurrently executing threads. The write-back stage overhead, T_{wb}, consists of a fixed overhead per thread plus an additional overhead due to write-updates. This additional write-update overhead increases with the number of bytes

written back. The write-update time depends on whether the data is in the cache or has to be accessed from the main memory.

In the Java implementation, native Java threads are created on the processors to execute the parallel threads of the JavaSpMT thread pipelining model. This thread creation is achieved through a method call in the `ParLoop` class. The `ParLoop` class is derived from the Java `Thread` class and invokes a Java `Thread` library call to initiate the JavaSpMT threads. Since JavaSpMT relies on the Java thread package for thread creation, the thread creation overhead is determined by the implementation of the Java thread package on the underlying system. Unlike the C implementation, the thread creation overhead (T_{cr}) is incurred each time a parallel code region is executed. Thus, T_{cr} has a significant effect on the parallel execution of Java applications, especially on loops with smaller granularities.

The library call overhead, T_{lib}, is due to the execution of method calls in the `MyThread` class that implements the thread pipelining stages. The Java thread library implementation results in thread contention when multiple threads try to access shared data elements (as explained later in Section 5.3.3.1). This contention increases as more threads try to access the same data location. Since the `MyThread` library is implemented through the shared memory space, the library call overhead increases with the number of processors. Hence, instead of measuring the overhead for each individual pipeline stage, we measured the total effect of the library call overhead for P threads executing on P physical processors.

	Number of Processors	First Execution (average/σ)	Later Executions (average/σ)
Thread creation (ms)	2	11.78/0.09	1.77/0.05
	4	13.86/0.09	3.58/0.26
	8	17.51/0.09	9.10/0.12
Library call (ms)	2	3.24/0.015	0.017/0.0004
	4	3.27/0.005	0.019/0.0010
	8	3.35/0.021	0.023/0.0020

Table 5.3: Parallelization overhead of the JavaSpMT model (in milliseconds) on the SGI Challenge System.

Table 5.3 shows the parallelization overhead of JavaSpMT. With JIT compilation, the execution time is higher the first time a parallel code is executed than for later executions due to the initial loading and compilation of the additional classes. Since there is a fixed overhead per thread execution, overhead increases as additional threads are executed on each additional processor in the system.

5.3.2 Performance With Application Programs

In this section, we evaluate the performance of the speculative multithreading parallelization model by applying the technique to parallelize the test application programs described in Section 5.2. To transform the serial code into the thread pipelined parallel code, we identified the portions of the code that could be parallelized and manually inserted our thread library routines in the appropriate places. We executed the parallel codes on the SGI Challenge multiprocessor system. The resulting speedups were computed using the execution time of the original sequential programs on a single processor of the test system as the basis.

For the C applications, the performance speedups are determined when excluding the initial thread creation overhead in some cases. In the following, we explicitly mention if the performance results include the thread creation overhead or not. For the Java applications, the parallel execution times used in determining the speedups exclude the initial class loading and JIT compilation overheads.

5.3.2.1 Matrix Multiplication

We applied the speculative multithreading parallelization model to parallelize a matrix multiplication program computing the product of two NxN matrices. Each thread computed the inner products of a N/P sized strip of the result matrix. The matrix dimension was varied from 100 to 300 with an increment of 50.

Figure 5.1(a) shows the speedups of the parallelized matrix multiplication program on 2, 4, and

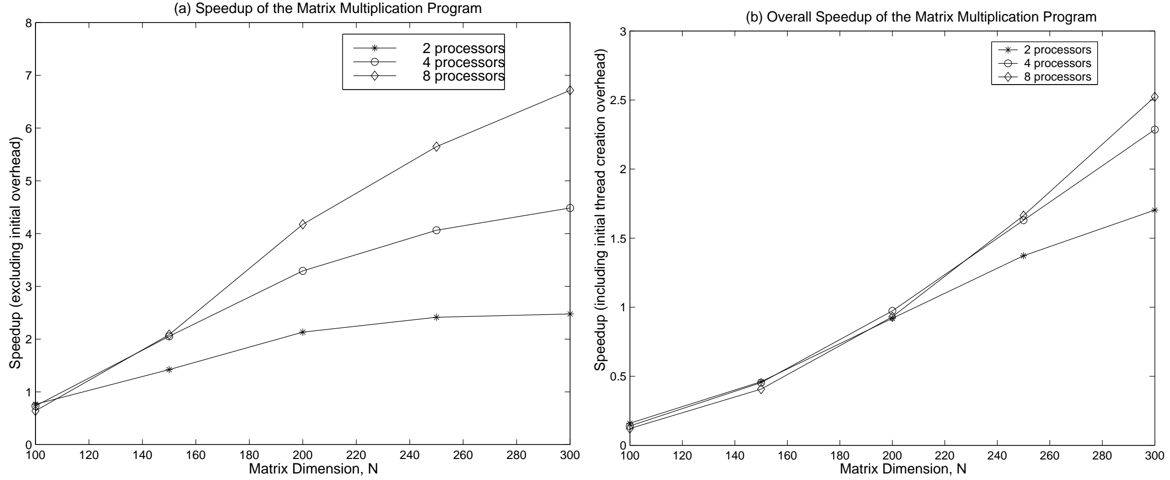

Figure 5.1: Speedups of the *Matrix Multiplication* program, parallelized using the C implementation of the speculative multithreading model on 2,4, and 8 processors of the SGI Challenge multiprocessor system : (a) *speedup excluding the effect* of the thread creation overhead; (b) *speedup including the effect* of the thread creation overhead.

8 processors of the test system as the matrix dimension is varied from 100 to 300. The speedups in Figure 5.1(a) were calculated using parallel execution times excluding the initial thread creation overhead, T_{cr}. The speedups increase as the problem size N is increased which increases the loop granularity. This larger granularity reduces the effect of the library call overhead. Furthermore, the performance scales well with the number of processors.

From the speedup plots, we observe that the speedups, in fact, exceed the ideal speedup values on 2 and 4 processors for $N = 250$ and $N = 300$. This is due to cache effects on the single processor runs. For $N = 300$, for example, the total number of elements in each matrix is $300 \times 300 \times 300 = 27$M. The total memory requirement for data in the program is then $27 \times 3 = 81$M floating point numbers for 3 matrices. Since the level-1 data cache in the SGI Challenge system is 32 Kbytes and the level-2 cache is 2 Mbytes, the entire data set will not fit in the cache. Thus, the single processor run has to bring in data blocks from the main memory, replacing the existing cache blocks, several times during program execution. In the parallel runs, on the other hand, each processor needs to access N/P blocks of each matrix and, thus, the memory requirement of each processor is much smaller

and in most cases the data will fit in the cache.

Figure 5.1(b) shows the overall speedups of the C implementation considering the effect of T_{cr}. For the matrix dimension of 200 or lower, the loop granularities are not large enough to overcome the effect of T_{cr} and to produce any performance speedup when the initial thread creation overhead is considered.

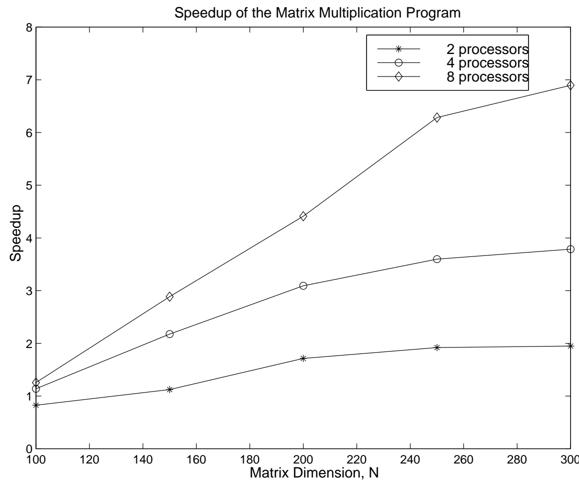

Figure 5.2: Speedups of the JavaSpMT version of the *Matrix Multiplication* program on 2,4, and 8 processors of the SGI Challenge multiprocessor system.

The performance of the JavaSpMT version of the matrix multiplication program is shown in Figure 5.2. As in the C implementation, the speedups increase with increasing problem size and scale well with number of processors. Note, that the Java performance includes both the thread creation and the library call overhead.

5.3.2.2 Gaussian Elimination

Gauss was applied on a dense matrix of dimension NxN. The problem size, N, was varied from 100 to 1000 in increments of 100. For a given problem size N, the outermost loop in the *forward elimination* code has N-1 iterations so that the the parallel code is executed N-1 times. To obtain an appropriate grain size for each thread, 8 iterations were combined to produce each thread's work.

Figure 5.3 shows the performance of the C version of *gauss* when parallelized using the speculative

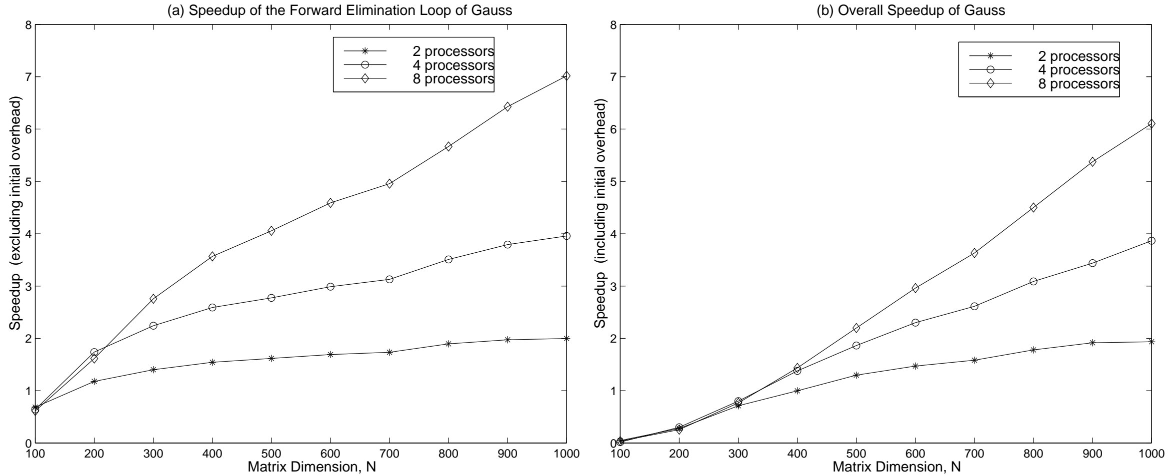

Figure 5.3: Speedups of the *Gaussian Elimination* program parallelized using the C implementation of the speculative multithreading model.

multithreading model. The speedup of the *forward elimination* loop excluding the initial thread creation overhead is shown in Figure 5.3(a). Figure 5.3(b) shows the overall speedup of the program including the initial overhead in the total parallel execution time. The *forward elimination* loop, which is completely parallelizable in a DOALL loop, shows significant speedup when parallelized using the speculative multithreading parallelization model. This loop accounts for about 98-99% of the total execution time, making the overall speedup quite high. Since the parallel loop is executed N-1 times (which is quite large for the chosen values of N), the thread creation overhead, T_{cr}, does not have a significant effect on the overall speedup. Furthermore, the overall speedup increases with increases in the problem size since larger problem sizes increase the amount of work per thread. The speedups also scale well with number of processors.

The speedup of the *forward elimination* loop in the Java implementation is shown in Figure 5.4. The speedup again increases with increases in the problem size since larger problem sizes increase the amount of work per thread. This larger grain size reduces the effect of the thread creation and the library call overheads. The performance does not scale well with increasing number of processors, however, due to the effect of the thread creation overhead in JavaSpMT which is incurred in each of the N-1 executions of the parallel code. As shown in Table 5.3, the overhead increases as more

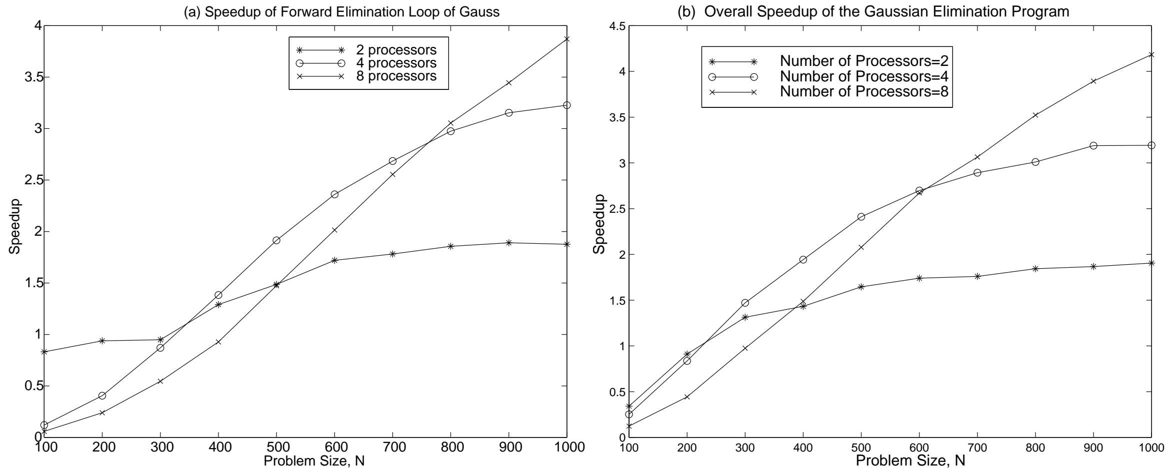

Figure 5.4: Speedups of the Java version of the *Gaussian Elimination* program parallelized using JavaSpMT model.

processors are used. Due to this increasing overhead when increasing the number of processors, the 4-processor performance is lower than that of the 2-processor system for small problem sizes. For a given problem size, there is less work per thread in the 4-processor case than in the 2-processor case, which magnifies the impact of the parallelization overhead. This is compounded by the fact that, since the overhead per thread is fixed, there is more total overhead when additional processors are used. This overhead effect is especially pronounced in the 8-processor case where a problem size of 800 is needed for the 8-processor system to exceed the 4-processor performance.

5.3.2.3 Perfect Club Benchmark MDG

The *INTERF* subroutine of the *MDG* program was executed on 2 to 8 processors of the SGI Challenge multiprocessor system. The *INTERF* subroutine is invoked 101 times in the program. Eight iterations per thread were again used to provide an adequate amount of work per thread. The cross-iteration dependence due to the reduction operation is handled through partial accumulation in local variables and then updating the actual data variables during the write-back stage.

Figure 5.5(a) shows the performance of the C version of the *MDG* program when parallelized

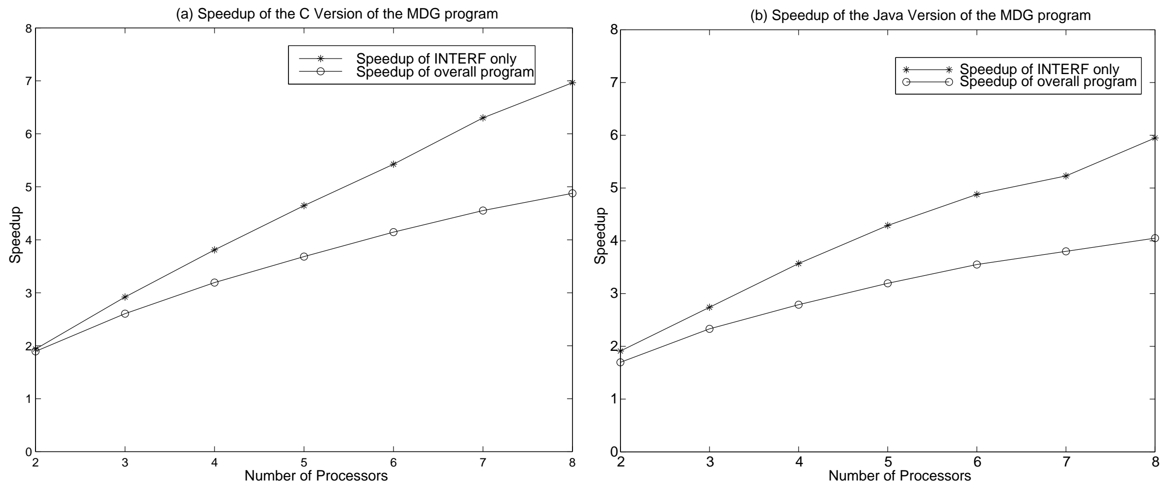

Figure 5.5: Speedups of the Perfect Club benchmark *MDG* as the number of processors is varied in (a) the C implementation; (b) the Java implementation.

using the speculative multithreading model. The speedups of the *INTERF* subroutine (without the thread creation overhead) on 2 to 8 processors of the SGI Challenge multiprocessor system scales linearly with the number of processors. The speedups are quite high considering the fact that the loop has some cross-iteration dependences. The overall speedup including the effect of the thread creation overhead is also quite high since about 90% of the program is parallelizable. Since the *INTERF* subroutine is executed 101 times, the thread creation overhead does not make a significant impact on the overall speedup.

The performance of the Java version of *MDG* parallelized using JavaSpMT is shown in Figure 5.5(b). The speedups, in this case, scale nearly linearly with the number of processors. Due to the relatively large granularity of the loop iterations, the parallelization overhead of JavaSpMT does not have a significant effect on the performance. The overhead effect is pronounced, though, for higher numbers of processors when the speedup does not scale exactly linearly with the number of processors. The overall speedup of the *MDG* program is again quite high in the Java implementation.

5.3.2.4 Word Count Program wc

The C version of the *wc* program was executed using a 0.6 Mbytes input file while varying the input buffer size. The buffer size, which determines the number of bytes read from the input file for each iteration of the *while* loop, was varied from 1 Kbytes to 20 Kbytes. Each thread was assigned one iteration of the parallel loop. The loop iterations are executed speculatively since the loop bound is unknown until runtime. There are only three data updates in each iteration that need to be written-back during the write-back stage of each thread.

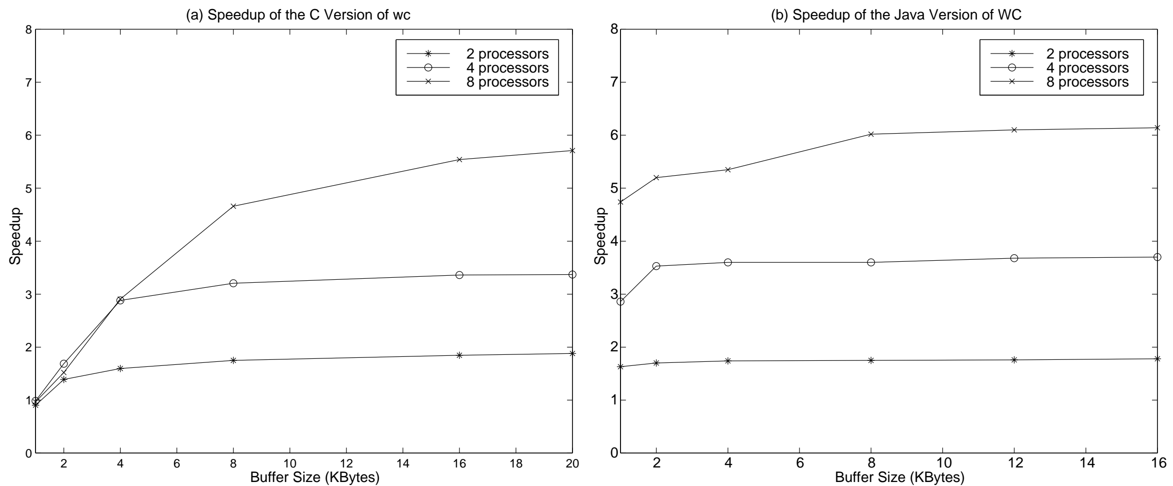

Figure 5.6: Speedups of (a) the C version of *wc* and (b) the Java version of *wc*.

The speedup of the parallel C version of the *wc* program with 2, 4, and 8 processors is shown in Figure 5.6 (a). The speedups were determined excluding the thread creation overhead. The speedup increases as the buffer size is increased, until the performance saturates, since increasing the buffer size increases the granularity of each thread.

The Java version of the *wc* program was executed while varying the input buffer size using an input file of 2 Mbytes. The buffer size, in this case, was varied from 1 to 16 Kbytes. Each thread was again assigned one iteration of the parallel loop. The speedup of the JavaSpMT parallelized *wc* program with 2, 4, and 8 processors is shown in Figure 5.6(b). The speedup again increases as the buffer size is increased, until the performance saturates at a buffer size of 8 Kbytes.

5.3.2.5 SPEC2000 Benchmark 183.equake

The 183.equake benchmark program was executed using the *test* data set. The *simulate* loop is executed only once in the program. The loop iterations are executed speculatively since the loop has an early exit condition. We assigned one iteration per thread in the transformed parallel code.

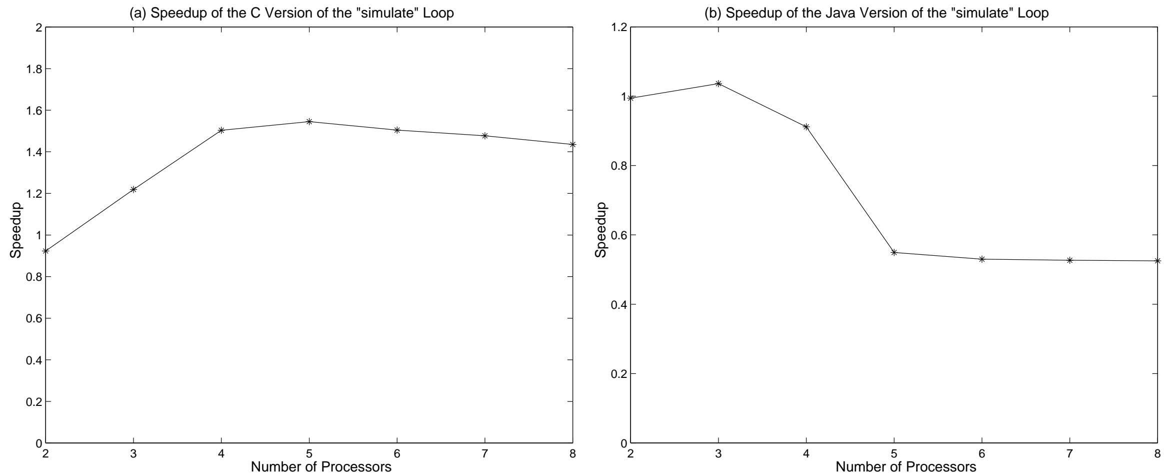

Figure 5.7: Speedup of the *simulate* loop in 183.equake on 2 to 8 processors of the SGI Challenge multiprocessor system using (a) the C implementation and (b) the Java implementation of the speculative multithreading model.

Figure 5.7(a) shows the performance of the C version of the parallel *simulate* loop on 2 to 8 processors of the SGI Challenge multiprocessor. The plot shows the speedups excluding the thread creation overhead. While the parallel execution of the loop does result in some performance improvement, the performance does not scale well. In fact, the performance starts degrading as the number of processors is increased beyond 5. The performance of the JavaSpMT parallel version of *simulate* shown in Figure 5.7(b) is even worse. There is no speedup at all and the performance degrades as more processors are used.

This poor performance of the *simulate* loop can be attributed to the cache performance on the SGI Challenge system [27]. The SGI Challenge system uses a write-back invalidate based cache coherence protocol. The write-back stages of the threads in the parallel version of the *simulate*

loop perform write updates on shared memory locations. Each time a thread performs its write-updates, the shared copies in the other threads become invalidated. These data must be brought into the cache the next time a thread needs to access them during its write-back. The loop has about 35000 iterations. The cache line invalidation in this large number of iterations results in significant execution-time overhead for the parallel execution of the loop. The cache effect becomes more significant as more processors are involved. In the Java implementation, the cache effect together with the increasing parallelization overhead makes the performance even worse with more processors.

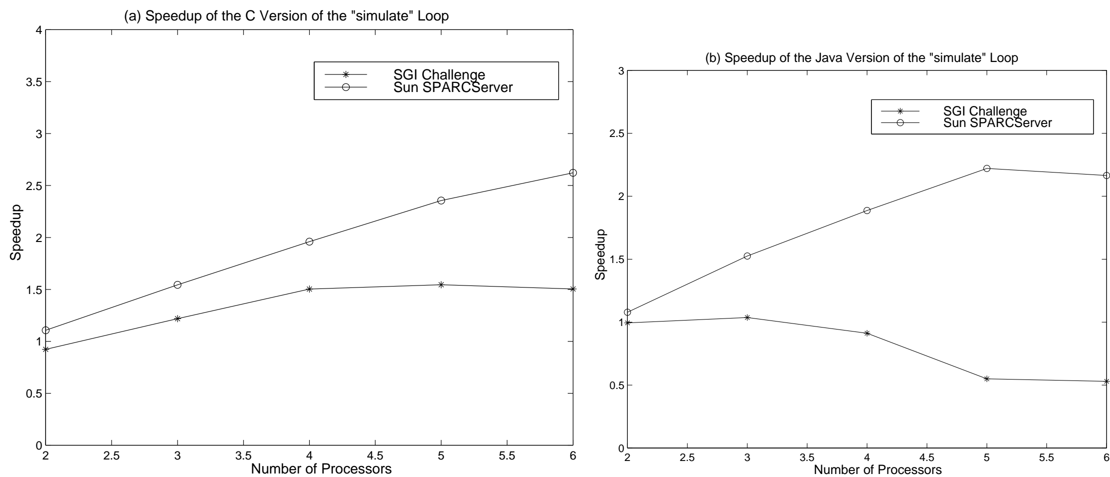

Figure 5.8: Speedup of the *simulate* loop in 183.equake on 2 to 6 processors of two different shared-memory multiprocessor systems: (a) C implementation; (b) Java implementation.

We executed both the C and the Java versions of the parallel *simulate* loop on a different shared-memory multiprocessor system to see how the performance is affected by the cache coherence protocol. We executed the 183.equake benchmark on the 6-processor Sun SparcServer 1000 shared-memory multiprocessor system that uses a write-update based write-back cache coherence protocol. Figure 5.8 shows the performance on this system together with the performance on the SGI Challenge system. Both the C and the Java implementations of the parallel code show better performance on the Sun SparcServer system. The speedup increases in both cases as more processors are used.

The write-update of cache lines by each thread improves the cache performance and thus results in better parallel execution-time performance.

5.3.3 Performance Comparison

In this section, we compare the application-level performance of the speculative multithreading parallelization model with some existing parallelization techniques. First, we identify the differences in the C and the Java implementations of the speculative multithreading parallelization model that result in relatively lower performance for parallel codes in Java. Next, we compare the C implementation of the parallelization model with an existing *inspector-executor* based run-time parallelization technique. We also compare the performance of JavaSpMT with an existing Java parallelization tool.

5.3.3.1 C Versus Java Language Implementation of the Speculative Multithreading Model

There are some fundamental differences in the way threads are executed on multiprocessors in C and Java. The basic difference on the SGI Challenge system arises from the way threads are created. As already discussed in Section 5.3.1, in C, threads are created once at the beginning and reused over the duration of the program, as needed. Thus, the overhead for thread creation is incurred only once. In Java, on the other hand, threads are created and terminated for each parallel code region. This results in the thread creation overhead being incurred in the execution of each parallel code region. Furthermore, the Java thread library implementation requires excessive implicit thread synchronization which makes the library call overhead in Java relatively higher than the C implementation. These differences in the underlying language-specific implementation of thread packages result in different performance levels for the parallel execution of programs in C and Java.

To compare the performance of the C and Java language implementations of the speculative multithreading model, we use the parallelized *MDG* program as an example. Figure 5.9(a) shows the

speedup of the *INTERF* routine of MDG using both C and Java implementations of the speculative multithreading model. These results indicate that the speedup of the Java implementation is lower than the C implementation for the same problem size. Furthermore, this difference increases as the number of processors is increased. Note that the original sequential execution time of the Java version of MDG (executed with a JIT compiler) was slower than the sequential C version by a factor of 3. Since the speedups were determined using the corresponding sequential execution times as the basis, the results do not provide a direct comparison between C and Java parallel execution performance.

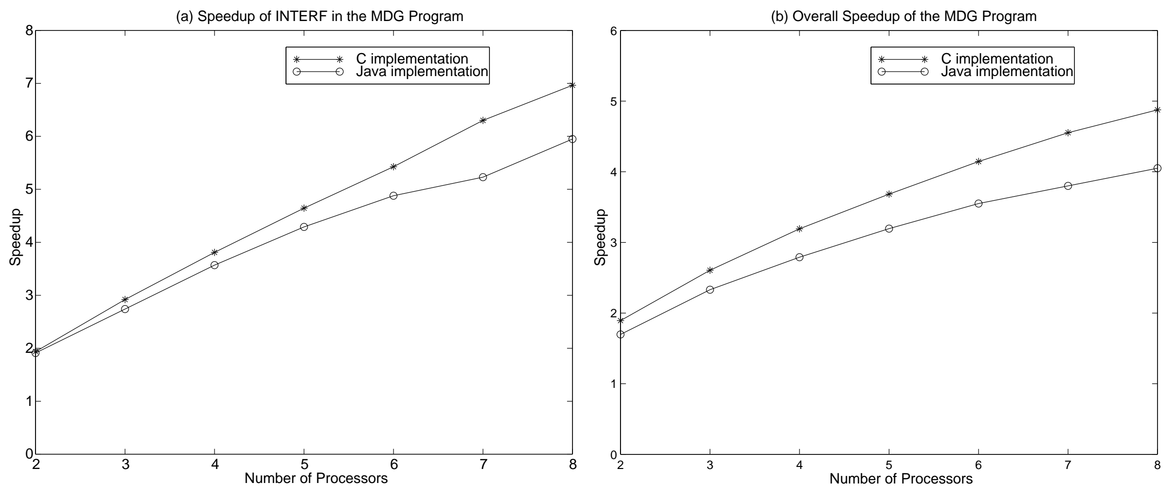

Figure 5.9: Comparison between the C and the Java implementations of the speculative multithreading parallelization model.

The performance degradation of the Java implementation is due to its relatively high parallelization overhead compared to the overhead of the C implementation. First, the Java execution had to incur the thread creation overhead in each execution of the *INTERF* routine. The C implementation incurred this overhead only during the first invocation of *INTERF*. This initial overhead is not included in the speedups plotted in Figure 5.9(a).

In addition, the standard Java libraries make extensive use of locks to ensure mutual exclusion in critical sections of the JVM code itself. Access to Java class variables requires resolving the corresponding constant pool entry of the given class. To ensure that only a single thread can access

the constant pool at any time, this access is controlled using a critical section. The JavaSpMT thread library uses various class variables for communication between threads through the shared address space. Some of these variables, such as the *thread_active* flag, are read by several threads at the same time while being updated by only one thread at a time. Since no more than one thread can update the memory location, the accesses do not need to be mutually exclusive. In Java, however, accesses to these variables must go through the critical section for constant pool resolution. The locking delay increases as more threads try to gain access to the critical section. This then leads to the higher library call overhead observed as the number of processors increases. The performance of JavaSpMT could be improved by using a version of the underlying Java Virtual Machine that had a better implementation for faster locks.

Figure 5.9(b) shows the overall speedups of the *MDG* program. The C implementation includes the effect of the thread creation overhead. Because of the higher combined overhead for thread creation and JavaSpMT thread library calls, the Java performance is still lower than the C performance even with the initial thread creation overhead of C.

5.3.3.2 Comparison with Existing Run-time Parallelization Techniques

Most of the existing run-time parallelization techniques are *inspector-executor* based techniques, as discussed in Chapter 2. In the speculative multithreading parallelization model, however, there is no separate inspector stage as the data-dependence checking and execution are overlapped in concurrent threads executing in different pipeline stages. Since our parallelization model does not require any additional overhead to execute a separate *inspector* stage, it is expected to perform better than the *inspector-executor* algorithms [11, 37, 38, 54].

To show how the speculative multithreading parallelization model performs compared to *inspector-executor* based schemes, we consider the performance of the Privatizing DOALL Test run-time scheme [37]. Since the Privatizing DOALL Test is applied to compiled languages, we consider the C implementation of the speculative multithreading model for comparison. In [37], the authors

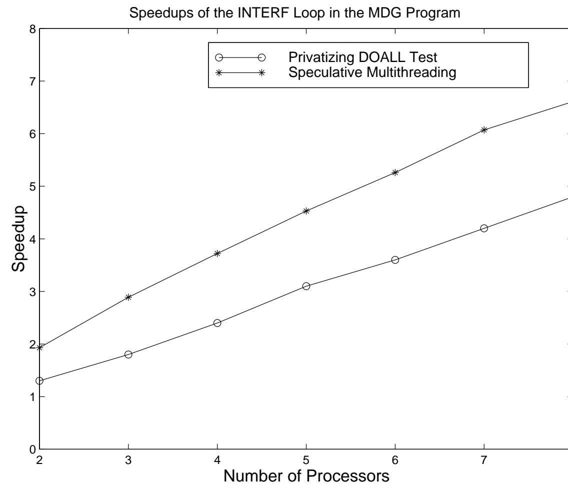

Figure 5.10: Comparison of the C implementation of the speculative multithreading model and the Privatizing DOALL Test run-time scheme.

evaluated their scheme using the Perfect Club benchmark [12] codes. We compare the speedups of both schemes for the *MDG code* from the Perfect Club Benchmarks. Figure 5.10 shows the speedups obtained for *Loop 1000* of the *INTERF* routine in *MDG* using the Privatizing DOALL Test and the C implementation of the speculative multithreading model.

The speedups of the Privatizing DOALL test is from the reported results presented in [37]. This run-time scheme uses two modes of execution - *inspector-executor* and *speculative* mode. The speedups used in Figure 5.10 are obtained using the *inspector-executor* mode of execution. The parallel execution time, in this case, includes the overhead of the separate *inspector* stage. Our speedup results include the initial thread creation overhead. The speedups of the Privatizing DOALL Test should be lower than the speedups of the speculative multithreading model which has no separate *inspector* stage. From Figure 5.10 we see that the speculative multithreading parallelization model indeed results in significantly higher speedups than the Privatizing DOALL Test, as expected.

5.3.3.3 Comparison with the JAVAR Parallelization Tool

JAVAR [4] is a Java parallelization tool that parallelizes sequential Java programs using standard Java multithreading and synchronization primitives. It can automatically parallelize loops with static data dependences as well as multiway recursive methods.

To compare the performance of the JavaSpMT model with that of the JAVAR tool, we used both techniques to parallelize the matrix multiplication program. Figure 5.11 shows the speedups of these two different parallel implementations. The matrix dimension was varied from 100 to 300 with an increment of 50. We find that the performance of the program parallelized with the JAVAR tool is about 2-4% faster than when parallelized with the JavaSpMT technique.

This small difference in performance is due to the additional overhead incurred in the pipeline stages of the JavaSpMT model. However, since JavaSpMT supports speculation on control dependences, we can parallelize a wide variety of loop structures, including *do-while* loops. It is not possible to parallelize *do-while* loops using a tool such as JAVAR. Furthermore, the run-time data-dependence checking in JavaSpMT allows loops for which the dependence patterns cannot be statically determined to be parallelized as well. Thus, although JavaSpMT has slightly higher overhead than JAVAR, it can parallelize program constructs that are impossible to parallelize without support for speculative execution.

5.4 Performance Evaluation of the Comprehensive Dynamic Processor Allocation Scheme

In this section, we show the effectiveness of the dynamic processor allocation scheme in dynamically adapting the execution of a parallel application program based on program behavior and the availability of processors in the system. First, we discuss the overheads associated with the dynamic processor allocation scheme as measured on the Sun SparcServer 1000 multiprocessor system.

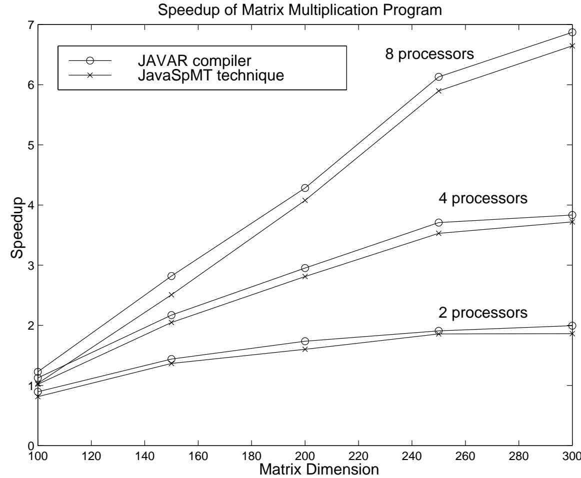

Figure 5.11: Comparison between the JAVAR and the JavaSpMT execution techniques.

Next, we present some performance results to show the dynamically adaptive nature of the processor allocation scheme. We then compare the performance of the dynamic processor allocation scheme with the standard time-sharing mechanism in a multiprogrammed environment. Finally, we use a parameterized synthetic test program to evaluate the sensitivity of the load sampling interval on execution-time performance as the system load and the amount of parallelism in the program change. We use another synthetic loop to evaluate the parallelization heuristic used in our dynamic processor allocation scheme to parallelize *do-while* type loops.

All the experiments for evaluating the performance of the dynamic processor allocation scheme were performed for both the C and the Java implementations of the speculative multithreading model analyzed in Section 5.3. Note that the problem sizes of the test programs in C and Java are not necessarily the same for the same experiments due to the difference in the execution model for C and Java.

5.4.1 Dynamic Processor Allocation Overhead

The implementation of the dynamic processor allocation scheme results in some execution-time overhead for the run-time system. First, there is the overhead for system initialization, which we

		C Implementation	**Java Implementation**
Initialization Overhead	incurred once	8.1580 ms	1400.46 ms
Profiling Overhead	incurred each time a loop is profiled	0.0015 ms	0.1045 ms
Decision-making Overhead	incurred each time a loop is invoked	0.0198 ms	0.5464 ms
Load Determination Overhead	incurred each time the load sampling timer expires	43.8050 ms	1108.45 ms

Table 5.4: Dynamic processor allocation overhead (in milliseconds) on the 6-processor Sun SparcServer multiprocessor system.

call the *initialization overhead*. This overhead is due to the initialization of different data structures. The *initialization overhead* is incurred only once during program execution. The profiling of the original sequential loops results in additional *profiling overhead*, which happens each time a target loop is profiled. The decision-making process to determine the number of processors to allocate to the loop results in the *decision-making overhead*. This overhead is incurred once each time the target loop is invoked. Finally, the *load determination overhead* is incurred every time the load sampling timer expires. The impact of the *load determination overhead* can be reduced by increasing the sampling timer interval, although at the expense of obtaining less precise load information.

Table 5.4 shows these overheads for both the C and the Java implementations of our dynamic processor allocation scheme measured on the 6-processor Sun SparcServer system. The overheads for the C implementation are considerably smaller than those of the Java implementation. Specifically, the *initialization* and the *load determination* overheads are quite high in the Java implementation. The initialization of the Java run-time system requires an external file access to obtain the loop parameters as well as the parallelization overhead for the target system. The file access time increases the *initialization* overhead significantly. In Java, the system load is determined using the *top* Unix utility, which causes significant overhead each time the load is sampled. In the C implementation, on the other hand, the initialization parameters are part of the program code and, thus, the initialization does not require any external file access. Also, the system load is determined using a Unix system call

that makes the load determination a light-weight process. The low overhead for load determination in the C implementation allows us to sample the system load quite frequently during program execution without incurring a significant increase in the overall execution time.

5.4.2 Dynamically Adapting to a Program's Characteristics

For parallel application programs with dynamically varying program behavior, there may not be any fixed processor configuration that maximizes performance speedup. For instance, the parallelization overhead will dominate the parallel execution time for very small problem sizes, and, thus, parallelization may actually result in a performance degradation. A different number of processors may provide the best performance depending on the problem size. To demonstrate this varying behavior, Figure 5.12 shows the execution times of different iterations of the *forward elimination* loop of the Java version of *gauss* on 1, 3, and 6 dedicated processors of the Sun SparcServer system. The first iteration has a problem size of N=100 with the problem size decrementing by 1 in each successive invocation. The timings in this figure clearly show that the processor configuration that provides the best performance for this loop changes as the problem size changes. Towards the end with the smaller problem sizes, the single processor execution gives the best performance as the parallel execution times are dominated by the parallelization overhead.

Our dynamic processor allocation scheme is able to determine the number of processors that can most efficiently be used under a program's current behavior. Dynamically reallocating the number of processors can thus improve performance over one particular processor configuration. In the following, we use three of our test applications with varying workload to demonstrate the effectiveness of our dynamic processor allocation scheme in allocating the number of processors that will result in the best performance.

For any given problem size, N, the workload of *gauss* dynamically changes during program execution as discussed earlier. The workload of *matrix multiplication* (*mat_mul*) can be varied by changing the matrix dimension N. Similarly, the workload for *wc* is determined by the input file used.

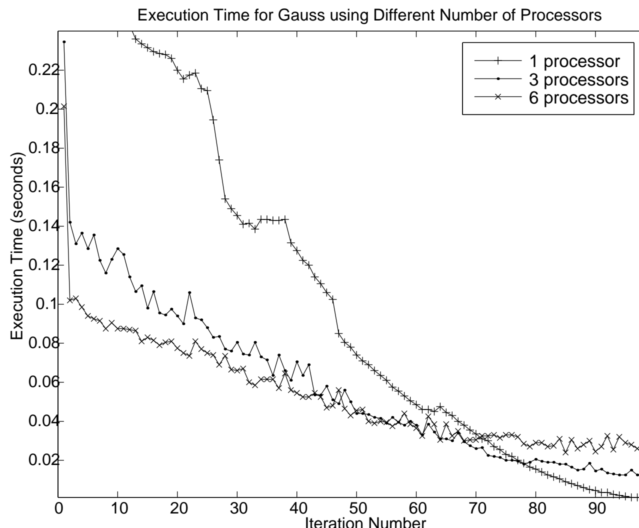

Figure 5.12: Execution time of the Java version of *gauss* (N=100) on 1, 3, and 6 dedicated processors of the Sun SparcServer 1000 system.

The workload of both the *MDG* and the *183.equake (quake)* benchmarks are fixed since they use a fixed data file during execution. Thus, we only considered *gauss*, *mat_mul*, and *wc* in the following experiment. In *mat_mul* and *wc*, we simulated the workload variation by using a combination of different workloads by using different values of N for *mat_mul* and different input files for *wc*.

5.4.2.1 C Implementation

We executed *gauss* with a problem size of N=200. For a given N, the *forward elimination* loop is executed N-1 times. The iteration count for the first invocation is N and is reduced by 1 in each successive invocations. The execution profile of individual iterations of the outer-most loop of *forward elimination* as shown in Figure 5.12 revealed that the loop can efficiently utilize fewer processors as problem size decreases.

Mat_mul was executed with 6 different problem sizes of N=2, 5, 10, 15, 20, and 50. The matrix multiplication loop was executed 10 times for each workload and the entire set was executed a total of 5 times. Execution profile of *mat_mul* showed that the first 5 problem sizes do not have sufficient workload for parallel execution while the last one (N=50) provides the best performance

Application Program	**Dedicated 6-processor**		**Dynamic Processor Allocation**	
	Speedup	Average # of processors	Speedup	Average # of processors
Gauss	1.15	6	1.53	2.90
mat_mul	1.29	6	1.61	2.23
wc	2.44	6	3.16	3.23

Table 5.5: Performance of C applications with a dynamically changing program workload.

when executed with 6 processors.

Wc was executed using a buffer size of 1000 bytes on two different input files. The smaller file is 717 bytes and the larger file is 0.6 Mbytes. The loop was invoked a total of 175 times, 100 of which are on the smaller input file. The smaller file required sequential execution and the larger file required a 6-processor run for best performance.

Table 5.5 shows the performance of these three application programs using parallel execution on a dedicated 6-processor system and a dynamic processor allocation based execution on the Sun SparcServer multiprocessor system. The application programs using the dynamic processor allocation scheme were executed in the system with no other processes running. The load sampling timer interval was set to 5 seconds for all three applications.

The 6-processor speedup of *gauss* is 1.15. When the program is executed with the dynamic processor allocation scheme, the performance speedup obtained is 1.53, which is about 33% better than the parallel performance on the dedicated system. The average number of processors used by the application in this case is 2.9. *Mat_mul* and *wc* show improvements of about 25% and 30% in performance, respectively, when executed using our dynamic processor allocation scheme. The average number of processors used by these two programs are 2.23 and 3.23, respectively. As a result, the dynamic processor allocation scheme provides better performance than the dedicated parallel runs using a lower average number of processors.

Application Program	Dedicated 6-processor		Dynamic processor allocation	
	Speedup	Average # of processors	Speedup	Average # of processors
Gauss	1.90	6	2.03	3.66
mat_mul	1.55	6	2.29	2.33
wc	1.78	6	2.23	3.35

Table 5.6: Performance of Java applications with dynamically changing program workload.

5.4.2.2 Java Implementation

In the Java implementation, *gauss* was executed with a problem size of N=100. *Mat_mul* used problem sizes N=2, 3, 4, 5, 10, and 20. The matrix multiplication loop was executed 10 times for each workload and the entire set was executed a total of 5 times, as in the C implementation. The first four data sets resulted in the best performance with a sequential execution while the last two did so with 6 processors. *Wc* was applied on three input files using a buffer size of 100 bytes. The input files were 21 bytes, 5 Kbytes and 8 Kbytes in size. The loop was invoked a total of 175 times with 75 on the smallest file, 50 on the medium-sized file, and the remaining 50 on the largest file. *Wc* performed best on a single processor for the first file and on 6 processors for the other two files. The application programs were executed in a dedicated system using a 10 seconds load sampling interval for the dynamic processor allocation scheme.

As shown in Table 5.6 the dynamic processor allocation scheme again provides better performance for the Java application programs (with varying workloads) compared to the 6-processor dedicated system run. *Gauss* shows about 7% performance improvement using an average of 3.66 processors. The speedup of *mat_mul* is about 50% better than the 6-processor run with 2.33 processors, on average. *Wc* shows an improvement of about 25% using 3.35 processors, on average.

The dynamic processor allocation scheme was able to improve the performance by dynamically changing the processor configuration to the one that would provide the best performance for each invocation of the loop in the given program as the problem size changed. Furthermore, by allocating only the number of processors that would be best utilized, the dynamic processor allocation scheme can make processors available to other processes in the system thereby improving the overall system

performance. The results presented in Table 5.5 and 5.6 demonstrate that it is important to adapt to a program's varying characteristics even on a dedicated system.

5.4.3 Dynamic Processor Allocation versus Time-Sharing in a Multiprogrammed System

In this section, we show that the dynamic processor allocation scheme improves the performance of parallel application programs compared to using a standard time-sharing mechanism in a multiprogrammed system. A parallel application that uses all of the processors in the system while other applications are running simultaneously will have to time-share some of the processors with these applications. Time-sharing will slow down the execution of the application programs compared to the execution on a dedicated system. The amount of slowdown will depend on the amount of computation in each thread and the time quantum of the scheduler. Since the dynamic processor allocation scheme allocates only the processors that are available, application programs executing under this mechanism do not have to share processors. Thus, there will not be any performance degradation due to resource sharing. The slowdown in this case will be completely determined by the number of processors available to be allocated to an application, which is a direct function of the system load.

To show the effectiveness of the dynamic processor allocation technique, we executed each parallel application simultaneously with carefully controlled background loads on the 6-processor Sun SparcServer 1000 system. All the test applications were executed using a fixed problem size to eliminate the effect of workload variation in the overall performance. Thus, the performance will be affected by the system load only. Since workload variation is inherent in *gauss*, its performance will be determined by both the program workload and the system load, however. The problem size of each application programs were chosen to ensure that the 6-processor parallel execution time on a dedicated system is the best performance for each application.

We use *slowdown factor* as a metric to measure the performance of the parallel application pro-

grams under system load. The slowdown factor of a parallel application, compared to the 6-processor run, in the presence of system load is defined as follows:

$$slowdown = \frac{execution\ time\ with\ system\ load}{parallel\ execution\ time\ on\ a\ dedicated\ system\ with\ 6\ processors}$$

A slowdown factor greater than one indicates that the parallel application is slowed down by the simultaneously executing system load. Note that a slowdown factor closer to one is better.

To measure the slowdown factors, each of the test programs was first executed simultaneously with one background sequential application program. The application programs were then executed simultaneously with other parallel application programs. In the following, we first analyze the performance of the C implementation of the dynamic processor allocation scheme compared to standard time-sharing mechanism. We then discuss the results for the Java implementation.

5.4.3.1 C Implementation

Table 5.7 shows the program characteristics used in the C implementation. The second column in the table shows the number of times the target loop in each program is executed in the program. The last column shows the overall execution time of the program when executed on a single processor of the Sun SparcServer multiprocessor system. To measure the slowdown factors, we executed each application program in parallel on a dedicated 6-processor system. This parallel execution time was used as the basis for determining the slowdown factors in the presence of system load.

Slowdown with one sequential background load:

Each application program was executed simultaneously with one sequential application program using both time-sharing and the dynamic processor allocation scheme. The dynamic processor allocation based run-time system used four different load sampling intervals, 5, 10, 25 and 50 seconds,

Application Program	**Number of times loop executed**	**Sequential execution time** (seconds)
Guass (N=300)	5980	131.38
mat_mul (N=200)	20	200.53
MDG	30	212.00
wc (0.6 Mbytes input)	500	159.92
quake	4	224.00

Table 5.7: Characteristics of C programs used in measuring slowdown factors in the presence of system load.

to show the effect of the interval length on the execution times of the application programs.

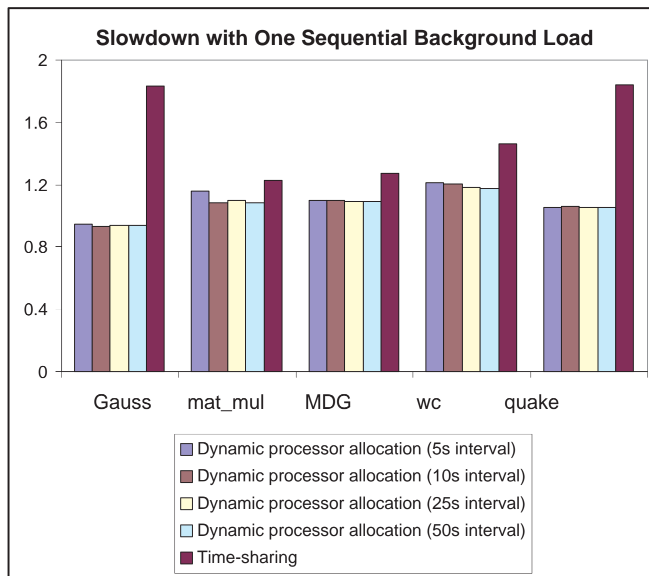

Figure 5.13: Slowdown factors of the C application programs when using time-sharing and dynamic processor-allocation with one sequential application running.

Figure 5.13 shows the slowdown factors of dynamic processor allocation and time-sharing in the presence of one sequential system load. All five application programs slowed down more with time-sharing than with dynamic processor allocation. The execution times with dynamic processor allocation increased by as much as 1.2 times whereas the increase in case of time-sharing is as much as 1.86 times the 6-processor performance. The slowdowns of *gauss* and *quake* with time-sharing were quite large in particular. Time-shared parallel execution of *gauss* is affected by the system load

as well as its varying workload. The amount of concurrency in *quake* is limited by cross-iteration data-dependences. In time-sharing, when a thread is put to sleep by the scheduler its successor will not be able to proceed because of its dependence on the predecessor thread, blocking its successor in turn. Thus, even with one system load, the parallel execution will suffer greatly. With dynamic processor allocation, the parallel loop is allocated the 5 other idle processors and, thus, threads do not have to be blocked for resource-sharing.

The run-time system used four different load sampling intervals for the dynamic processor allocation scheme. As shown in Figure 5.13, the load sampling interval does not have a significant effect on the performance of the application programs. Because of the very small overhead associated with determining the system load in the C implementation, the overall effect of load sampling is quite small. Thus, the load sampling interval can be made reasonably small in the C implementation to obtain accurate load information without incurring a high overhead.

Slowdown with concurrent parallel applications:

We executed the parallel applications concurrently with other parallel applications in different combinations to evaluate how the dynamic processor allocation scheme performs in the presence of other parallel applications. For all five test programs, we first concurrently executed the programs in pairs. Next, we executed them in groups of three, four and five in separate runs. The dynamic processor allocation scheme used a load sampling interval of 25 seconds in these experiments. Figures 5.14 to 5.17 show the slowdown factors associated with the dynamic processor allocation scheme and time-sharing when the application programs are executed in groups of 2, 3, 4, and 5.

Figure 5.14 shows the slowdown factors of the application programs when they are executed concurrently in pairs. For all 10 combinations of the pair-wise runs, the application programs performed better with the dynamic processor allocation scheme than with time-sharing. When there are two parallel programs running in a 6-processor system, each program will ideally be able to use half the cpu processing power. Thus, ideally the slowdown should be 2. The dynamic processor

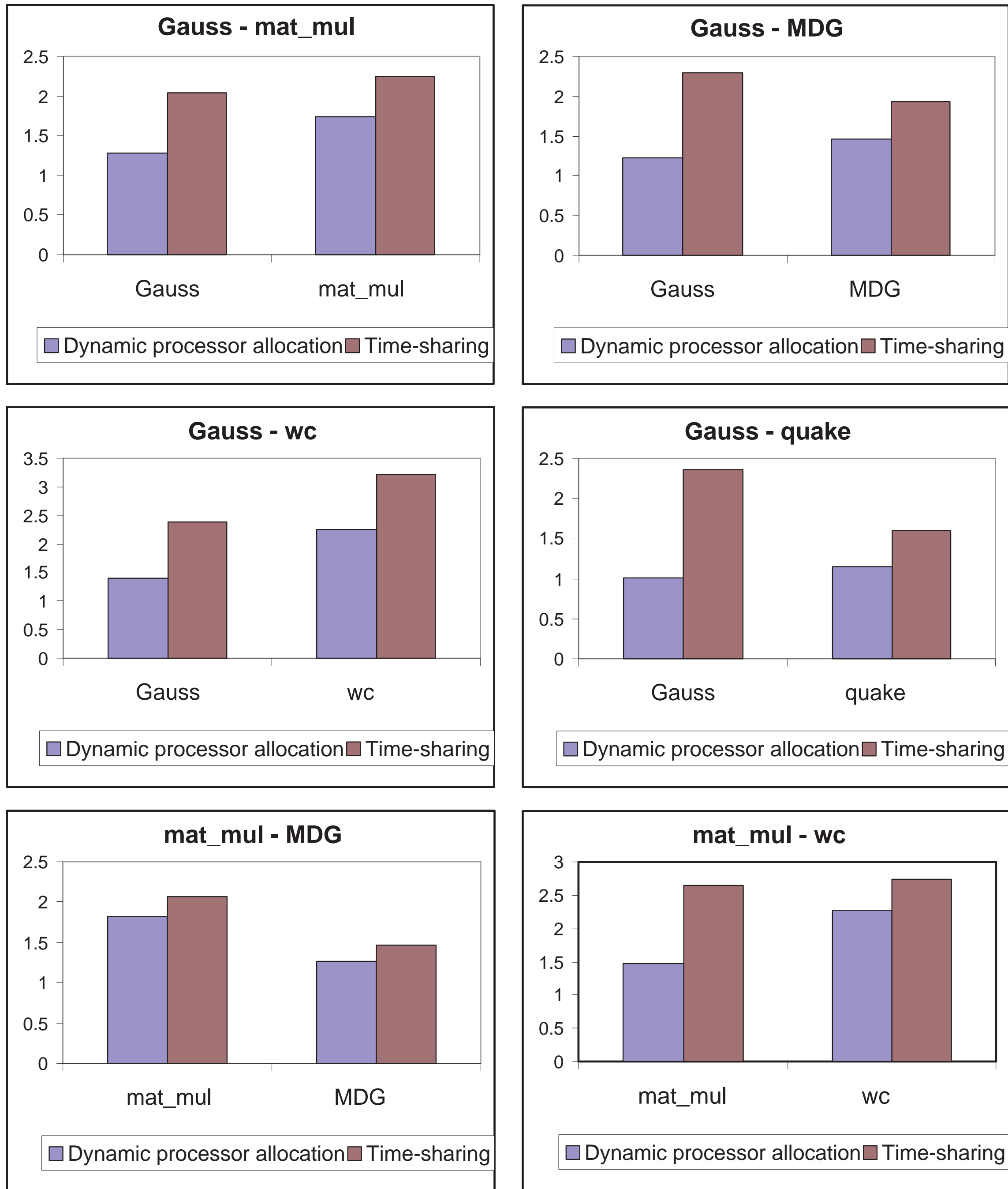

Figure 5.14: Slowdown factors of the C application programs when using time-sharing and dynamic processor-allocation with two concurrently executing parallel C applications.

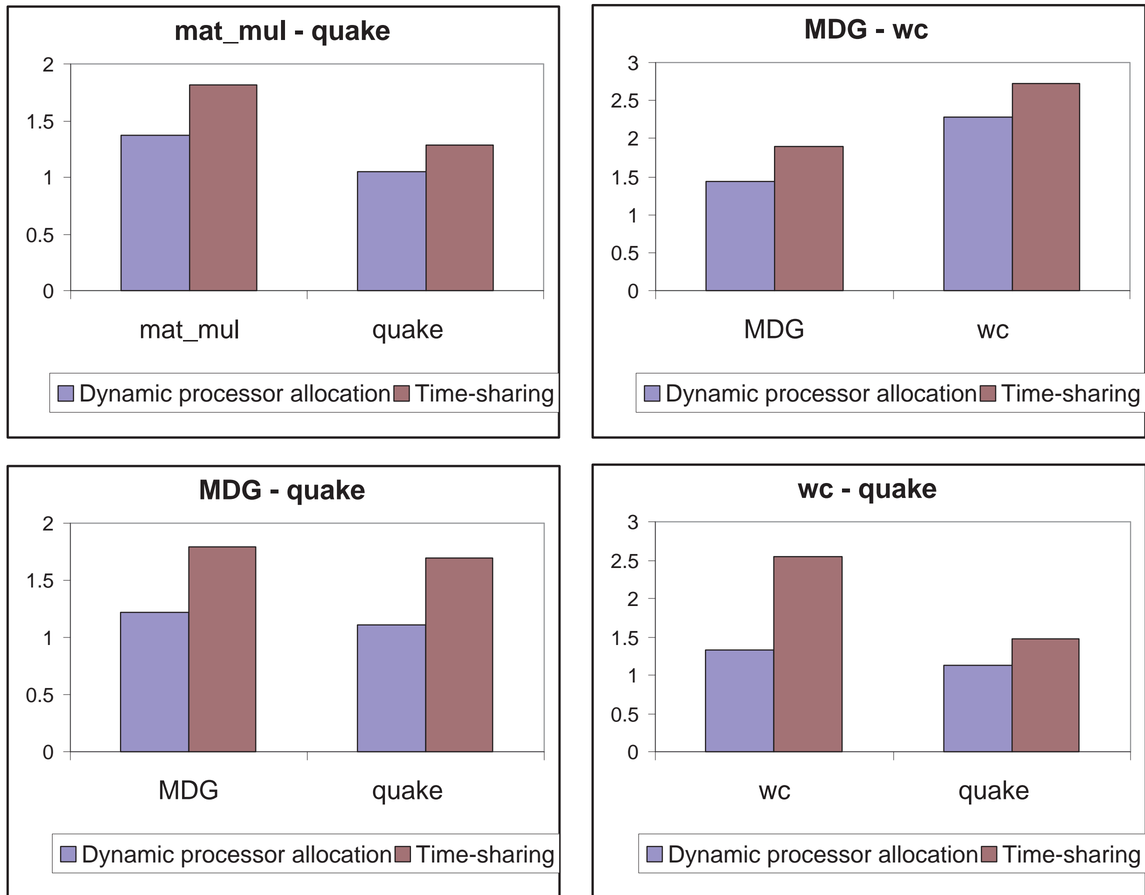

Figure 5.14: (Contd..) Slowdown factors of the C application programs when using time-sharing and dynamic processor-allocation with two concurrently executing parallel C applications.

allocation mechanism allocates the processors based on the program requirement and the system load. Thus, the run-time system divides the 6 processors between the two programs so that the programs do not have to contend for processors. From the results shown in Figures 5.14, we find the slowdown factors of the programs with the dynamic processor allocation scheme to be no more than 2.28, which is within a reasonable factor of the ideal slowdown. The average slowdown, in this case, is 1.46. With time-sharing, however, the application programs slowed down by as much as a factor of 3.2 with an average slowdown of 2.12. The added overhead of context-switching and poor cache performance resulted in poor performance with time-sharing.

The slowdowns of the application programs when executed in groups of three are shown in Figure 5.15. With three programs running together (each using one third the cpu resource), the ideal slowdown factor should be 3. From the slowdown results we see that all the applications performed better with dynamic processor allocation than with time-sharing. With dynamic processor allocation, the average slowdown factor of the application programs is about 1.9. The maximum slowdown is about 3.36 which is again within a reasonable factor of the ideal slowdown. Time-sharing resulted in an average slowdown of approximately 3.8. The slowdown factor is as much as 10.37 for *wc* (versus 2.92 with dynamic processor allocation) when executed together with *gauss* and *quake*.

When there are four parallel applications running in the system, each application will be able to utilize only one fourth of the cpu resources resulting in an ideal slowdown of 4. The slowdown factors in Figure 5.16 show that the application programs perform better with dynamic processor allocation than with time-sharing except for *mat_mul* when executed together with *MDG, wc*, and *quake*. In this case, time-sharing performs about 11% better than dynamic processor allocation, but at a cost to the other applications in the system. For this same run, *wc* slowed down by a factor of 8.75 with time-sharing as opposed to a factor of 2.99 with dynamic processor allocation. The maximum slowdown with dynamic processor allocation is 4.76 with an average slowdown of 2.4. Due to severe contention for processors, time-sharing resulted in a slowdown of as much as 15.49 with an average of 5.8 for all the applications.

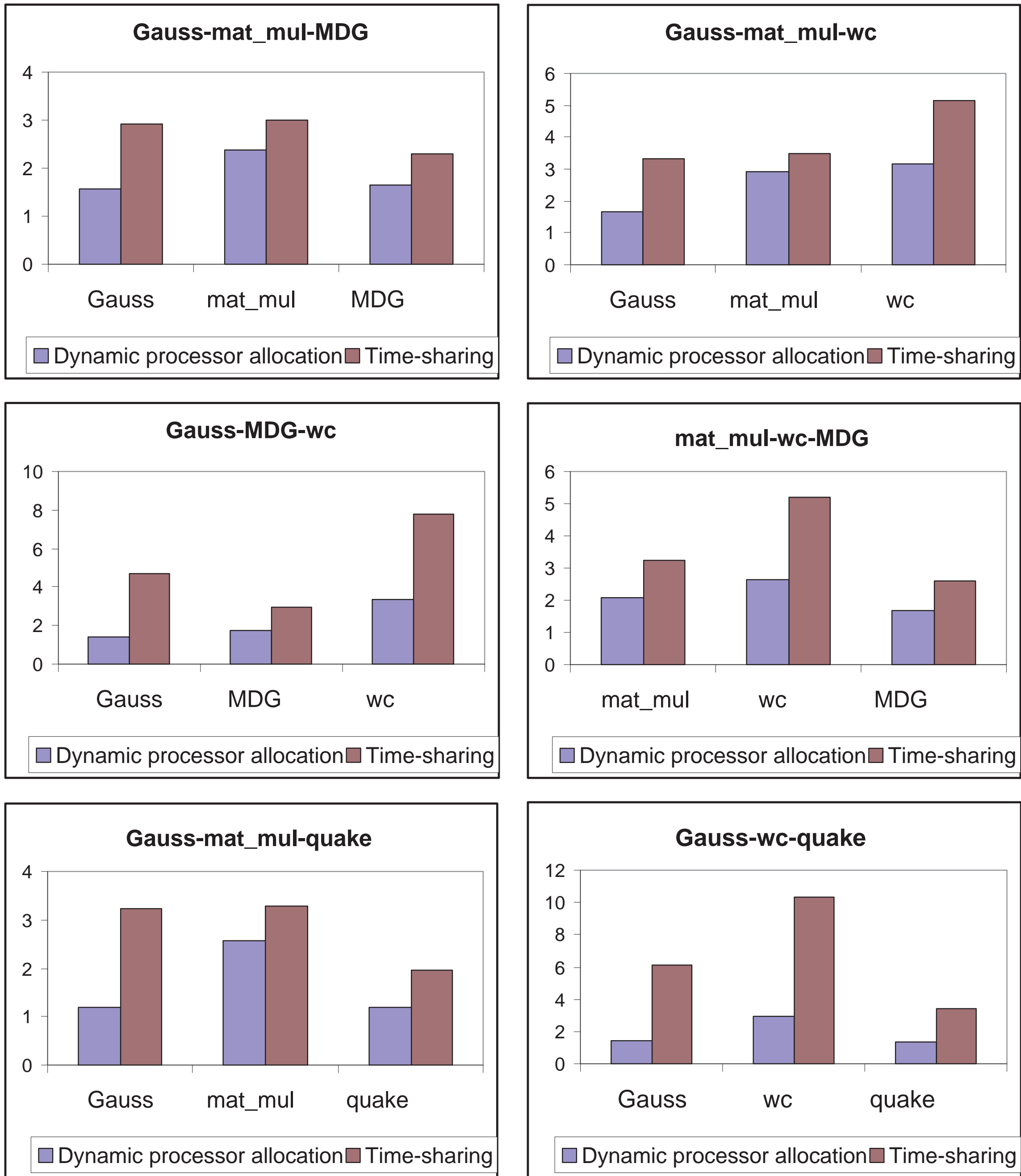

Figure 5.15: Slowdown factors of the C application programs when using time-sharing and dynamic processor-allocation with three concurrently executing parallel C applications.

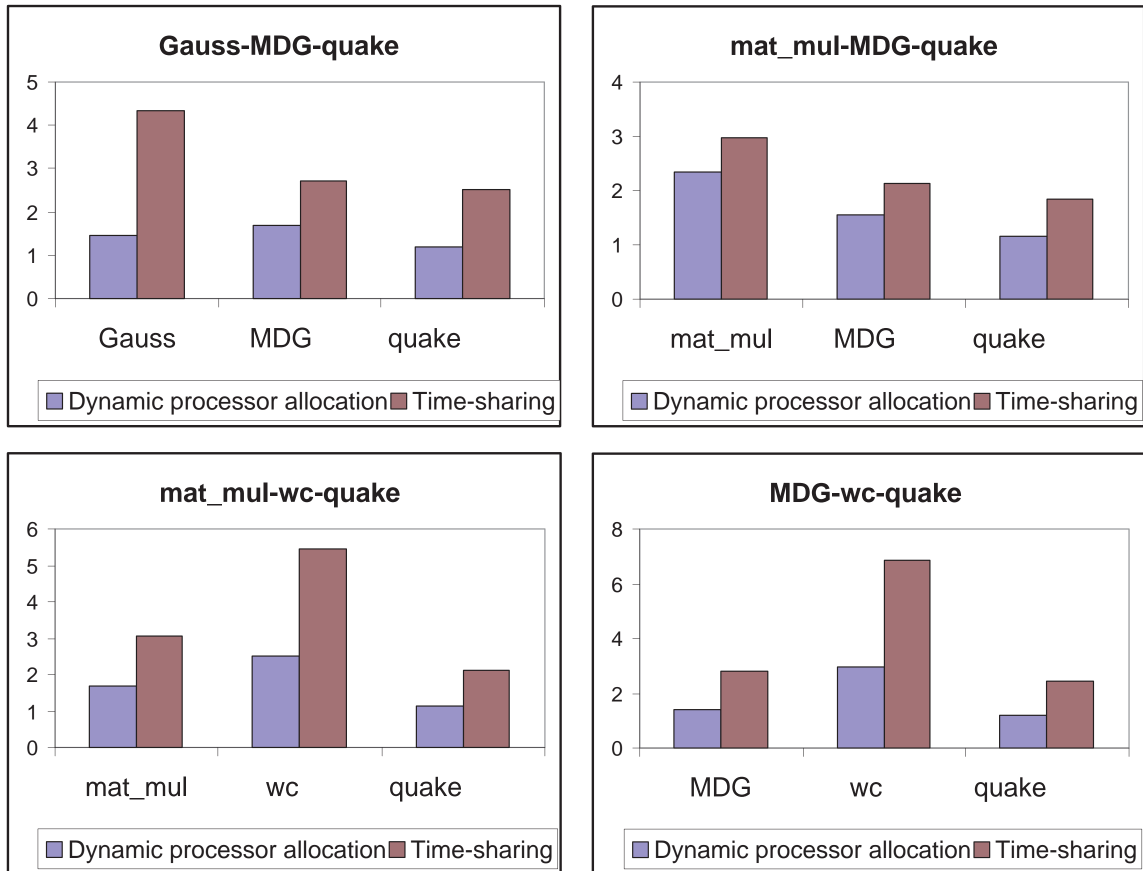

Figure 5.15: (Contd..) Slowdown factors of the C application programs when using time-sharing and dynamic processor-allocation with three concurrently executing parallel C applications.

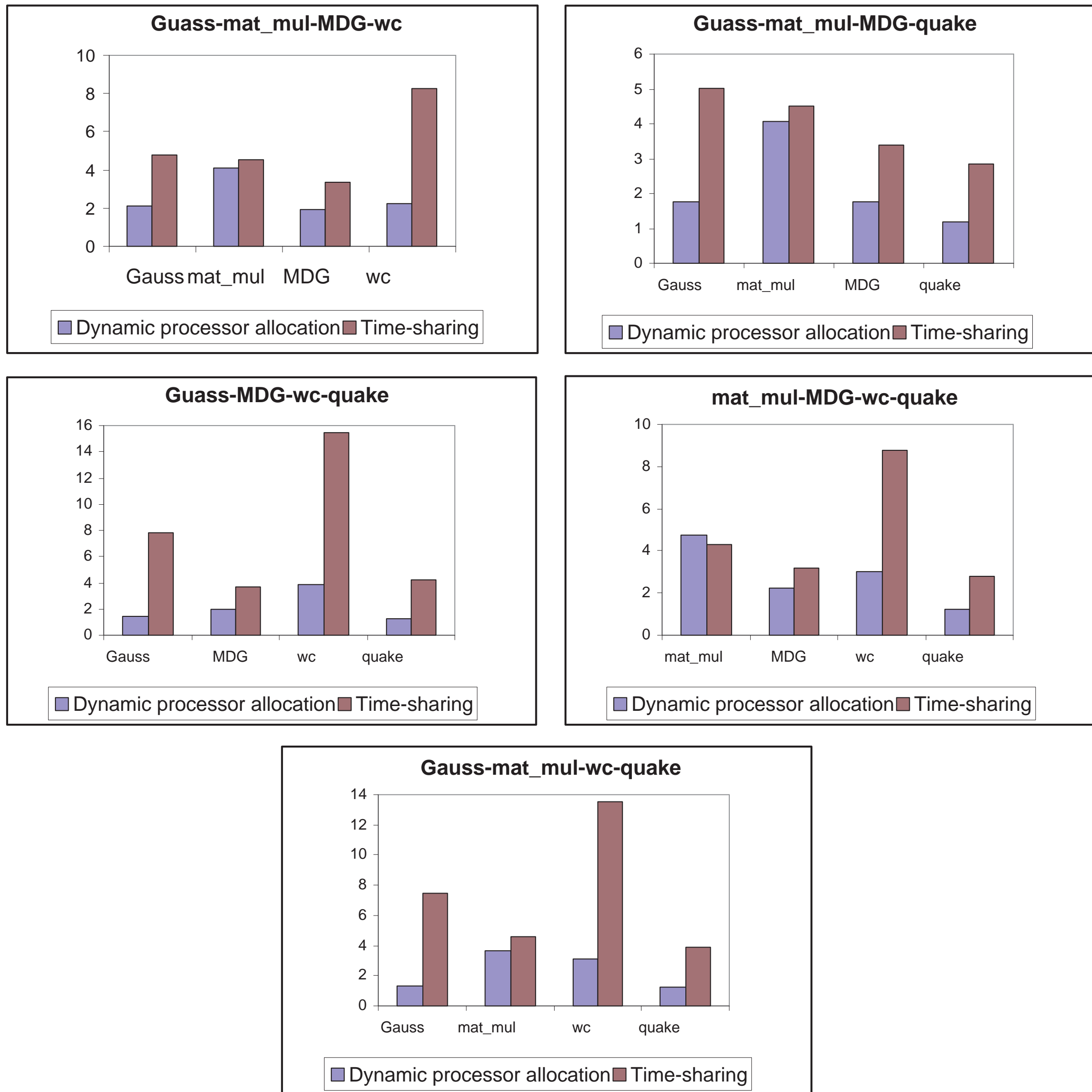

Figure 5.16: Slowdown factors of the C application programs when using time-sharing and dynamic processor-allocation with four concurrently executing parallel C applications.

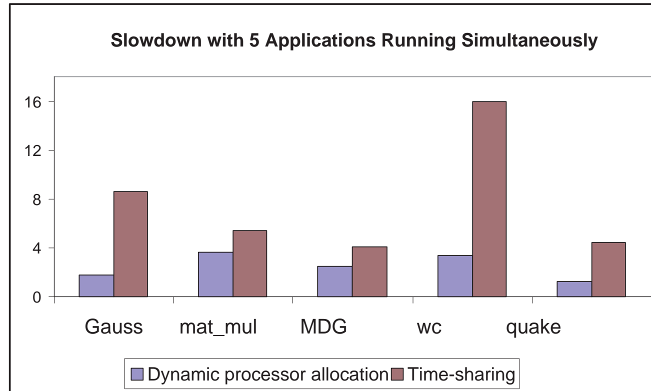

Figure 5.17: Slowdown factors of the C application programs when using time-sharing and dynamic processor-allocation with five concurrently executing parallel C applications.

Finally, with five applications running, each application should ideally be slowed down by a factor of 5. Since the system has 6 processors and 5 applications, it would be more profitable to execute the programs sequentially. The dynamic processor allocation scheme was able to allocate the processors efficiently and resulted in an average slowdown of 2.5 (with a maximum of 3.6) as shown in Figure 5.17. The application programs had to go through severe contention when executed in the time-shared mode and slowed down by as much as 15.9 with an average slowdown factor of 7.7.

We observed significantly large slowdowns for *wc* and *gauss* with time-sharing. In some cases, the parallel execution-time performance was much worse than the corresponding sequential execution. Both of these applications have a large number of parallel loops with considerably smaller granularity than the other applications. In time-sharing, the processors would be allocated to other applications in between these smaller parallel loops. The processors would not be allocated again to the application until the time-quantum for the other executing applications are over. Thus, the parallel loops in these programs had to incur additional overhead in parallel execution waiting for the operating system scheduler to schedule the processors to the loop. The dynamic processor allocation scheme allocates processor based on program workload and system load to eliminate contention for processors. Thus, the nature of the parallel loops did not make a significant difference in the

performance of these loops compared to the other applications.

5.4.3.2 Java Implementation

We considered four of the test application programs to measure the slowdown factors in the Java implementation. We excluded *quake* from the Java experiments for the following reason. Before the parallel loop in *quake* starts, the program reads an external file of size 1.6 Mbytes to initialize its data structures. In the interpreted mode of Java execution the initialization code (which is sequential) and the first execution of the *simulate* loop takes about 2665 seconds. If we execute *quake* concurrently with other parallel Java applications, those other applications would complete execution before *quake* even started its parallel loop. So, the effect of running *quake* would be the same as a sequential load instead of a concurrent parallel load. (The effects of sequential load are studied separately in this section.)

Table 5.8 shows the Java program characteristics used in the following experiments. The sequential execution time is measured on a single processor of the 6-processor Sun SparcServer system using Sun JDK 1.2 in the interpreted mode. To measure the slowdown factors, we executed each Java application program in parallel on a dedicated 6-processor system with standard JDK interpretation. The measured parallel execution times were used as the basis for calculating the slowdown factors. As the dynamic processor allocation based Java run-time system, we used our modified Sun JDK 1.2 interpreter. The standard JDK interpreter was used for the parallel runs in time-shared execution.

Slowdown with one sequential background load:

As with the C implementation, each Java application program was first executed simultaneously with a sequential application program using both time-sharing and the dynamic processor allocation scheme. For the dynamic processor allocation scheme, we used two different load sampling intervals,

Application Program	**Number of times loop executed**	**Sequential execution time** (seconds)
Guass (N=200)	1990	872.177
mat_mul (N=150)	10	1356.51
MDG	5	1850.00
wc (5 Kbytes input)	1200	280.74

Table 5.8: Characteristics of Java programs used in measuring slowdown factors in the presence of system load.

10 seconds and 50 seconds, to show the effect of the interval length on the execution times of the application programs. Unlike the C implementation, the load determination process in the modified JVM results in a significant overhead on program execution time.

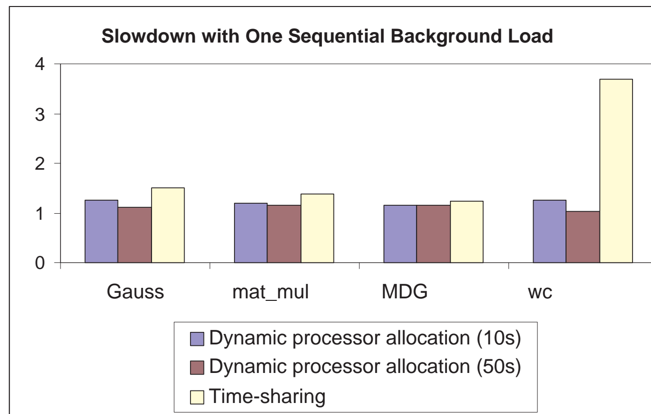

Figure 5.18: Slowdown factors of the Java application programs when using time-sharing and dynamic processor-allocation with one sequential application running.

Figure 5.18 shows the slowdown factors of the Java application programs when executed with one background sequential load. *Gauss*, *mat_mul* and *wc* slowed down more when using a 10-second sampling interval than with a 50-second interval due to the added overhead of checking the system load more frequently. The interval length did not have any effect on the overall execution time of *MDG* since the *interf* loop executed sufficiently longer than 50 seconds making it necessary to check the load before each invocation independent of the sampling interval. All four applications slowed down more with time-sharing than with dynamic processor allocation for both intervals. The execution time with dynamic processor allocation increased by at most a factor of 1.26 whereas with

time-sharing, the increase in execution time was as much as 3.69 times the 6-processor execution time.

Slowdown with concurrent parallel applications:

The parallel Java application programs were executed in groups of 2,3, and 4 using both dynamic processor allocation and time-sharing. With dynamic processor allocation, a 100 second load sampling interval was used. Figures 5.19 to 5.21 show the measured slowdown factors for the concurrently executing parallel Java application programs.

Figure 5.19 shows the slowdown factors of the Java application programs when executed in groups of two. The applications slowed down more with time-sharing than with dynamic processor allocation in all cases except for *MDG* in the *mat_mul* - *MDG* pair. For this pair, *MDG* in time-sharing showed slightly better (about 1%) performance than with the dynamic processor allocation scheme, possibly due to measurement error. *Wc* performed poorly with time-sharing in all three cases showing a slowdown of as much as 8.69 compared to a slowdown factor of 1.79 with dynamic processor allocation. The average slowdown with dynamic processor allocation is about 1.5. Application programs slowed down by a factor of 3, on average, with time-sharing.

As shown in Figure 5.20, all programs performed better with dynamic processor allocation when the programs were executed in groups of three. The programs slowed down by a factor of 2.1, on average, with a maximum slowdown of 3. With time-sharing, the same applications showed an average slowdown factor of 4.13. The slowdown with time-sharing was as much as 8.6 for *wc* (versus 2.69 with dynamic processor allocation).

Figure 5.21 shows the slowdown factors when all four programs were executed simultaneously. The programs slowed down 2.7 times, on average, with the dynamic processor allocation scheme. In this case, *wc* slowed down the most, by a factor of 3.14. With time-sharing, the application programs showed an average slowdown of 5.67 with *wc* slowing down by as much as 10.59.

As in the C implementation, the smaller loops in *wc* victimized it in the time-shared execution. Due to the smaller loop granularity, *gauss* also slowed down more than the other two applications,

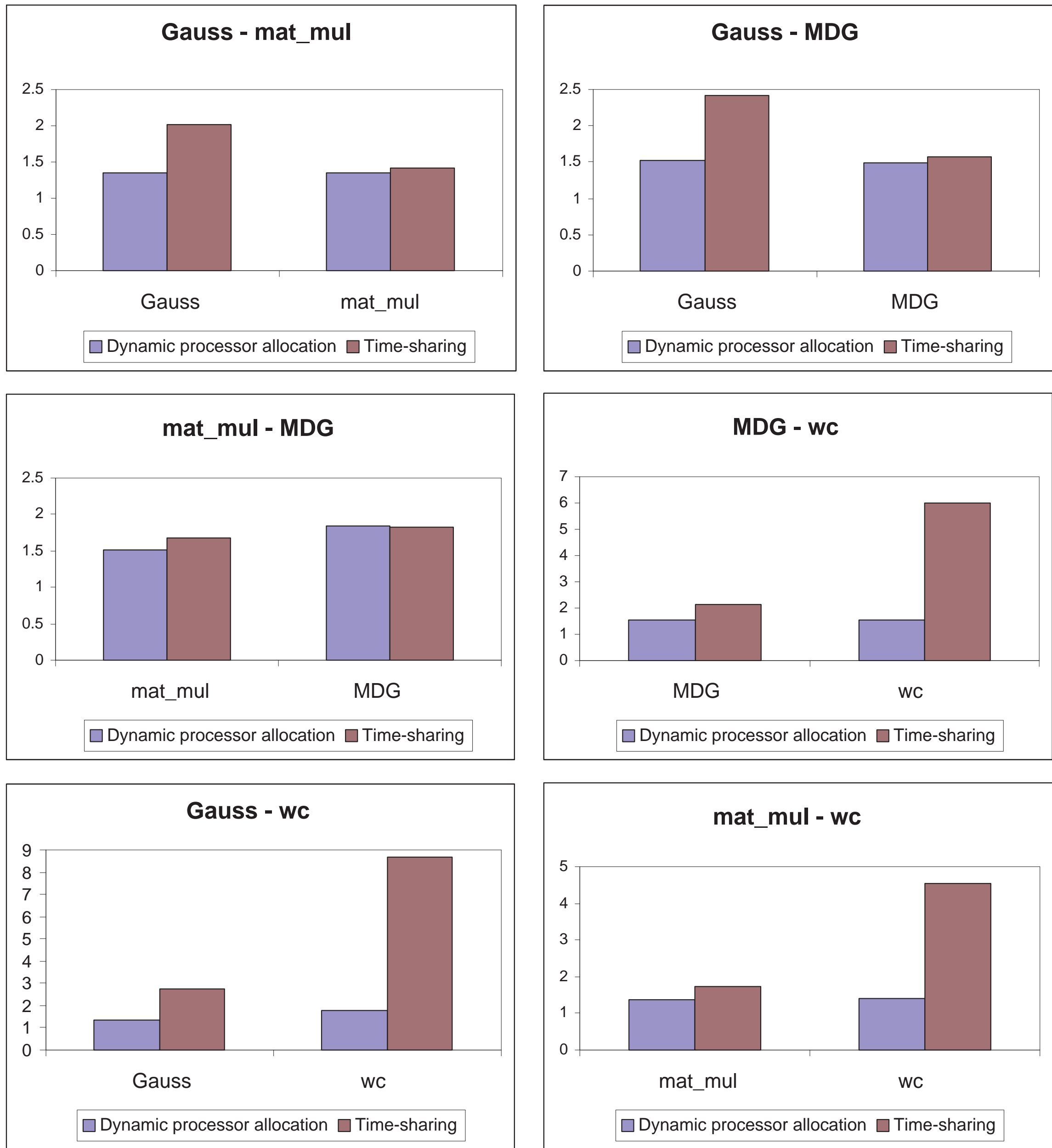

Figure 5.19: Slowdown factors of the Java application programs when using time-sharing and dynamic processor-allocation with two concurrently executing parallel Java applications.

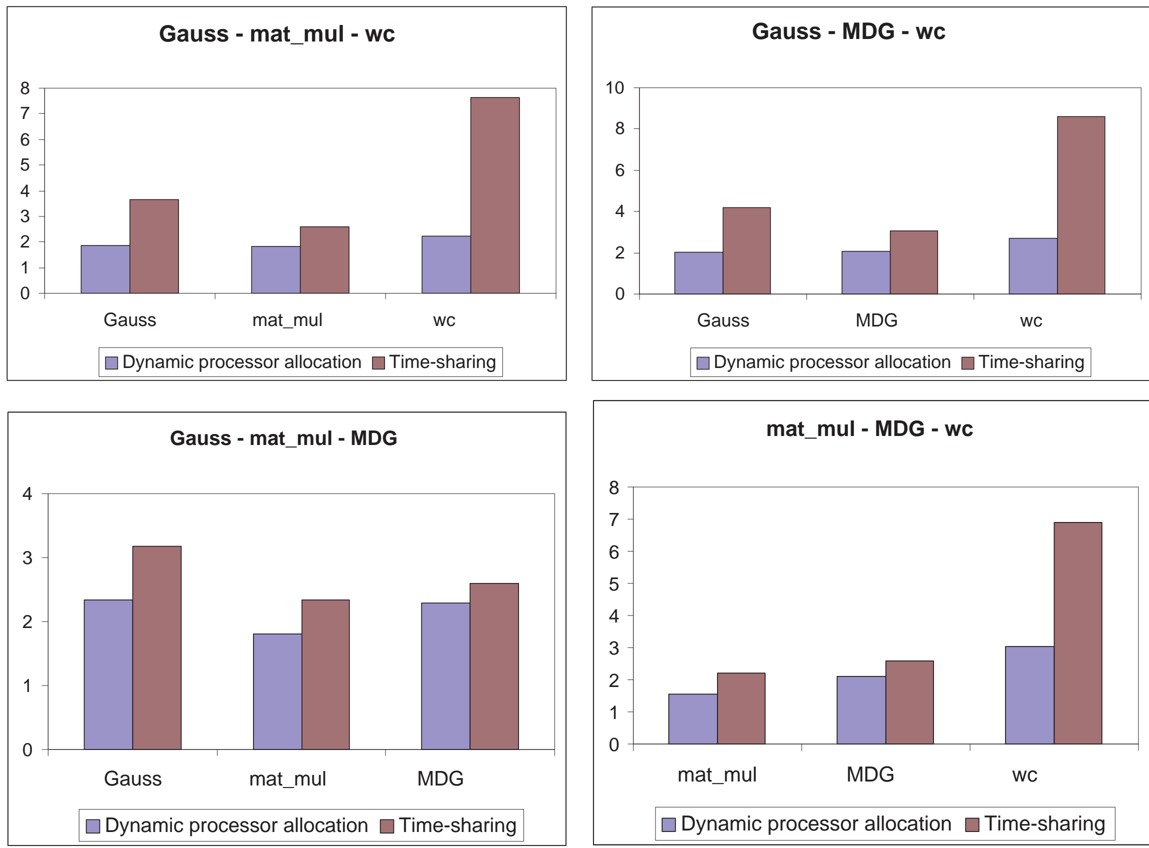

Figure 5.20: Slowdown factors of the Java application programs when using time-sharing and dynamic processor-allocation with three concurrently executing parallel Java applications.

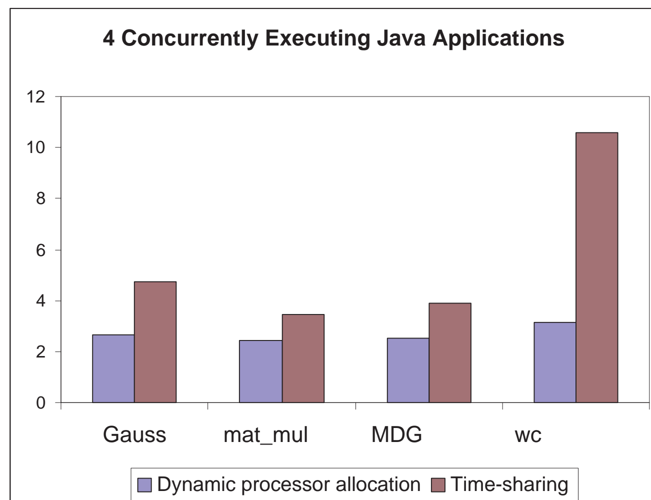

Figure 5.21: Slowdown factors of the Java application programs when using time-sharing and dynamic processor-allocation with four concurrently executing parallel Java applications.

although not as much as *wc*.

5.4.4 Effect of Load Sampling Interval on the Performance of the Dynamic Processor Allocation Scheme

With the dynamic processor allocation scheme, the performance of application programs in the presence of system load is affected by the ability of the run-time system to obtain an accurate

```
/* sequential code for sequential loop */
void seq_loop(int n1){

   int i,j;

   for(i=0;i<n1;i++){
       /* dummy loop simulating useful work */
       for(j=0;j<work;j++);
   }
}

/* sequential code for parallel DOALL loop */
void par_loop(int n2){

   int i,j;

   for(i=0;i<n2;i++){
       /* dummy loop simulating useful work */
       for(j=0;j<work;j++);
   }
}

void main(String argv[]){

   int k;
   int N, n1, n2;

   ......
   ......
   for(k=0;k<N;k++){
       seq_loop(n1);
       par_loop(n2);
   }
}
```

Figure 5.22: Synthetic Java program to evaluate the effect of the load sampling interval on parallel execution performance.

estimate of the current system load. In our current implementation, the run-time system checks the system load only when the load sampling interval timer expires. A smaller load sampling interval allows the run-time system to check the system load frequently and, thus, to use up-to-date load information in its processor allocation algorithm. However, because of the overhead associated with load determination, checking the load too frequently may result in significant overhead. If the load is sampled after a long interval, on the other hand, the previously sampled load may no longer be valid between the two sampling points. If the load increases significantly before the load is sampled again, the performance of the parallel application program will suffer due to resource contention. If the load is reduced considerably, on the other hand, the application program will not be able to fully utilize the available resources.

As discussed earlier in Section 5.4.1, the load determination overhead in the C implementation of the dynamic processor allocation scheme is quite small. The slowdown results presented in Figure 5.13 further revealed that the length of the load sampling interval did not result in any significant difference in the overall performance of the C application programs when executed with one background load. Thus, we can use a relatively small interval to guarantee up-to-date system load information while not introducing a large overhead in the program execution time. The sampling interval in the Java implementation, however, had a significant impact on the performance. The slowdown factors with one background load shown in Figure 5.18, improved by as much as 21% when the load sampling interval was increased from 10 seconds to 50 seconds. Thus, it is important that we do not use too small an interval in the Java implementation to avoid introducing too much overhead in the execution time.

We used a synthetic program to analyze the effect of the load sampling interval on the program's performance as the amount of parallelism in the program, as well as the system load, changes. The synthetic program shown in Figure 5.22 consists of a sequential loop (*seq_loop*) and a DOALL parallel loop (*par_loop*). The main method (*main*) of the program alternately executes *seq_loop* and *par_loop*.

Each iteration of both loops performs a fixed amount of work that is simulated by the inner

loop. The *work* parameter determines the actual workload per iteration. The total workload of the sequential and the parallel loops are defined by the outer loop parameters *n1* and *n2*, respectively. For a given total workload defined by $(n1+n2)$ iterations, the amount of parallelism in the program is defined as $\frac{n2}{(n1+n2)}$. For example, the amount of parallelism with 70 parallel loop iterations and 30 sequential loop iterations will be $\frac{70}{(70+30)} = 0.7$ or 70%. For the same total number of iterations $(70+30 = 100)$, a 90% parallelism will result in 90 iterations for the parallel loop and the remaining 10 iterations for the sequential loop.

For the following experiments, we defined $n1 + n2 = 100$ and executed the main loop 40 times $(N = 40)$. We used two different values of the parameter *work* to simulate two different total program workloads. For each workload, the $n1$ and $n2$ parameters were varied to vary the amount of parallelism (or the percentage of parallel code) to be 100%, 90%, 70%, and 50%. The program was executed with different numbers of background loads. The system load was varied from 1 to 3. Each load corresponds to a sequential program that computes for 5 seconds and then sleeps for a duration which is uniformly distributed with mean sleep time of 5 seconds and 10% variance. Due to the variance in the sleep time, with 3 background processes running, the total system load can vary anywhere between 0 to 3, for example. For a given combination of total program workload, percentage of parallel code, and background system load, the program was executed with the dynamic processor allocation based Java run-time system with six different load sampling intervals of 5, 7, 10, 15, 25, and 50 seconds.

We normalized both the total program workload and the load sampling interval with respect to the load determination overhead in the Java implementation. The normalization of the load sampling interval is determined as follows. For a given load sampling interval, T_i, the number of times the load is sampled in a program of duration T_{total} is at most $\frac{T_{total}}{T_i}$. The total overhead due to load determination is then $\frac{T_{total}}{T_i} \times T_{load}$, where T_{load} is the load determination overhead . The ratio of this total overhead to the execution time T_{total}, which is $\frac{T_{load}}{T_i}$, tells us how significant the overhead effect is. For example, with an interval length of 5 seconds the ratio is 0.22 on the Sun

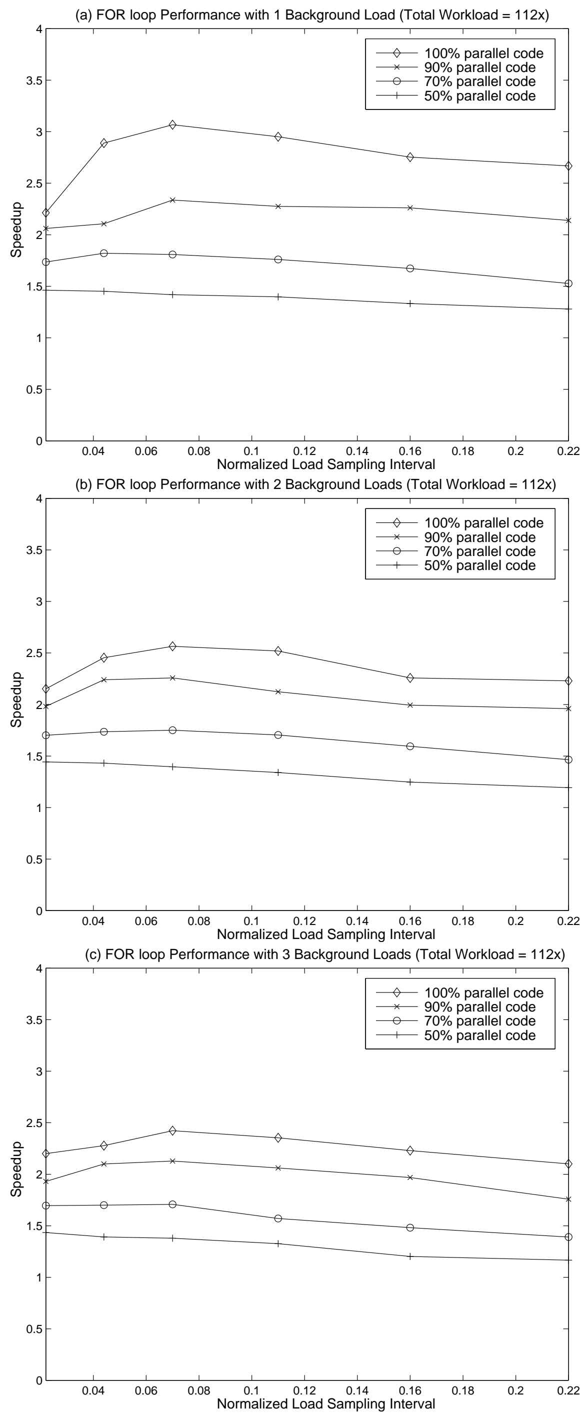

Figure 5.23: Effect of load sampling interval on parallel execution performance of a *for* loop with a normalized workload of 112x.

SparcServer system where the overhead $T_{load} = 1.1$ seconds for the Java implementation (discussed in Section 5.4.1). The interval length of 25 seconds results in a ratio of 0.044. Thus, a smaller ratio means a less significant overhead effect. The total workload of the program was normalized by taking a ratio of the total sequential execution time of the program to the overhead T_{load}.

Figures 5.23 and 5.24 show the speedup of the synthetic program for a normalized total workload of 112x and 223x, respectively, on the 6-processor Sun SparcServer multiprocessor system. The speedups were calculated using the execution time of the sequential code of the program on a single processor of the test system as the basis. The normalized load sampling intervals were 0.022, 0.044, 0.07, 0.11, 0.16, and 0.22 representing a gradually decreasing interval length from 50 seconds to 5 seconds.

Figure 5.23(a) shows the speedup of the program with a workload of 112x when executed simultaneously with one background load. When the program has a 100% parallel code (i.e., the sequential loop has zero workload), the change in the load sampling interval has significant impact on the program performance. The longest interval with a normalized value of 0.022 results in about 38% performance degradation compared to the best performance (corresponding to a ratio of 0.11). The smallest interval (0.22) results in a 15% performance degradation. The longer intervals suffered by using inaccurate estimates of the system load due to infrequent sampling. The excessive sampling overhead due to frequent sampling with the smaller intervals, on the other hand, degraded the performance even though the run-time system had an up-to-date system load information.

The interval length that provided the best performance is in the middle of these two extremes. This interval length ensured the load to be sampled frequently enough to have a reasonably accurate estimate of the system load, but long enough to reduce the total overhead effect. As the interval length is increased or decreased beyond this point, the performance degrades either due to inaccurate load information or due to too much overhead. As the percentage of parallel code in the program is reduced, the load sampling interval has less and less impact on the performance. As the percentage of parallel code is reduced, the percentage of serial code is increased which means that there are longer

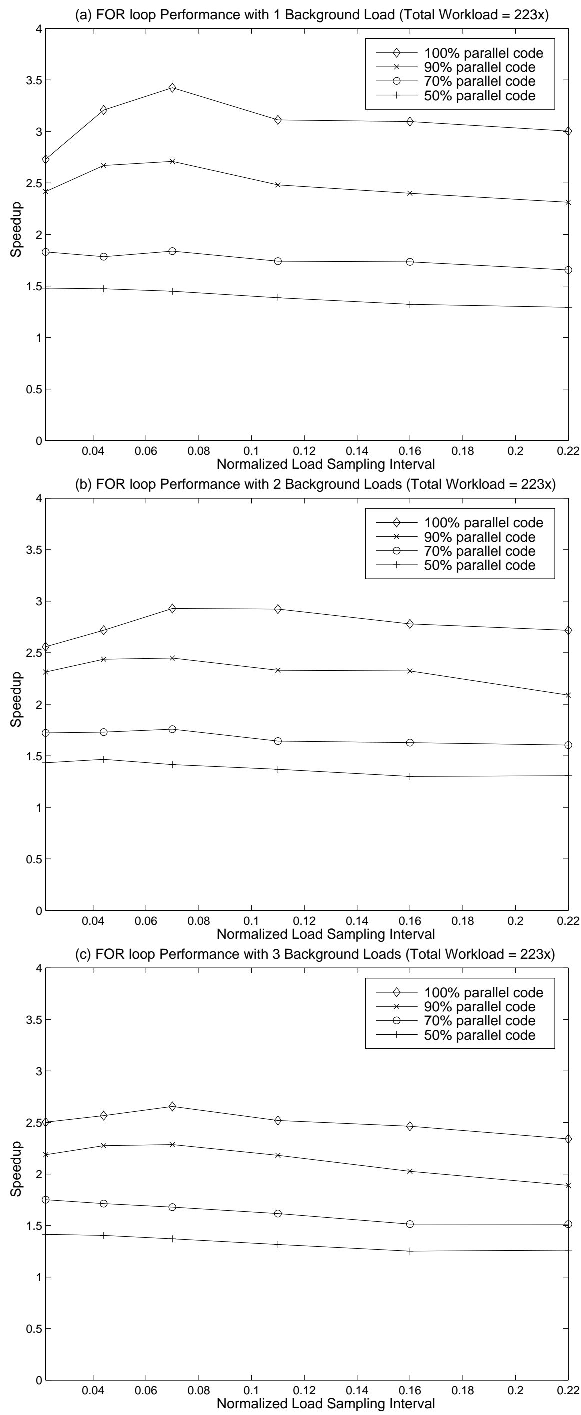

Figure 5.24: Effect of load sampling interval on parallel execution performance of a *for* loop with a normalized workload of 223x.

intervals in between successive parallel executions. For longer sampling intervals, the longer intervals between successive execution increase the number of times the system load is actually sampled before the parallel execution begins. This allows the run-time system to use a more up-to-date load for each parallel loop even with larger sampling intervals, thereby improving the performance.

The performance with two and three background loads in Figures 5.23(b) and 5.23(c) show performance similar to the one-load performance. The overall performance decreases with increasing load. The effect of the load sampling interval shows similar characteristics in all three cases.

The performance of the program with an increased total workload of 223x in Figure 5.24 also shows characteristics similar to the smaller workload performance. The speedup values with higher workload is higher - the one load speedup for 100% parallel code is 3.4 versus 3 for the smaller workload. As with the smaller workload case, too large or too small load sampling intervals resulted in performance degradation. The load sampling interval had less impact on the performance as the percentage of parallel code was reduced.

We repeated the experiments with *do-while* parallel loops using the same values for all the parameters. Due to the different loop constructs used, the total normalized workloads were a little different in this case, even with the same parameter values. The two normalized workloads were 107x and 209x. As shown in Figures 5.25 and 5.26, the load sampling intervals had effects similar to the *for* loop performance as the percentage of parallel code and the system load were varied.

5.4.5 Comparison of Different Parallelization Heuristics for *Do-While* Loops

In this section, we show the effectiveness of the dynamic processor allocation algorithm in classifying loops as sequential or parallel based on the loop workload. The parallelization heuristic used in the dynamic processor allocation scheme (discussed in Chapter 4) compares the estimated sequential execution time of the loop with the parallelization overhead on the target system to determine if the loop workload is large enough to amortize the overhead. Estimating the sequential execution time for the current instance of a loop depends on the type of loop being considered. Execution time of *for*

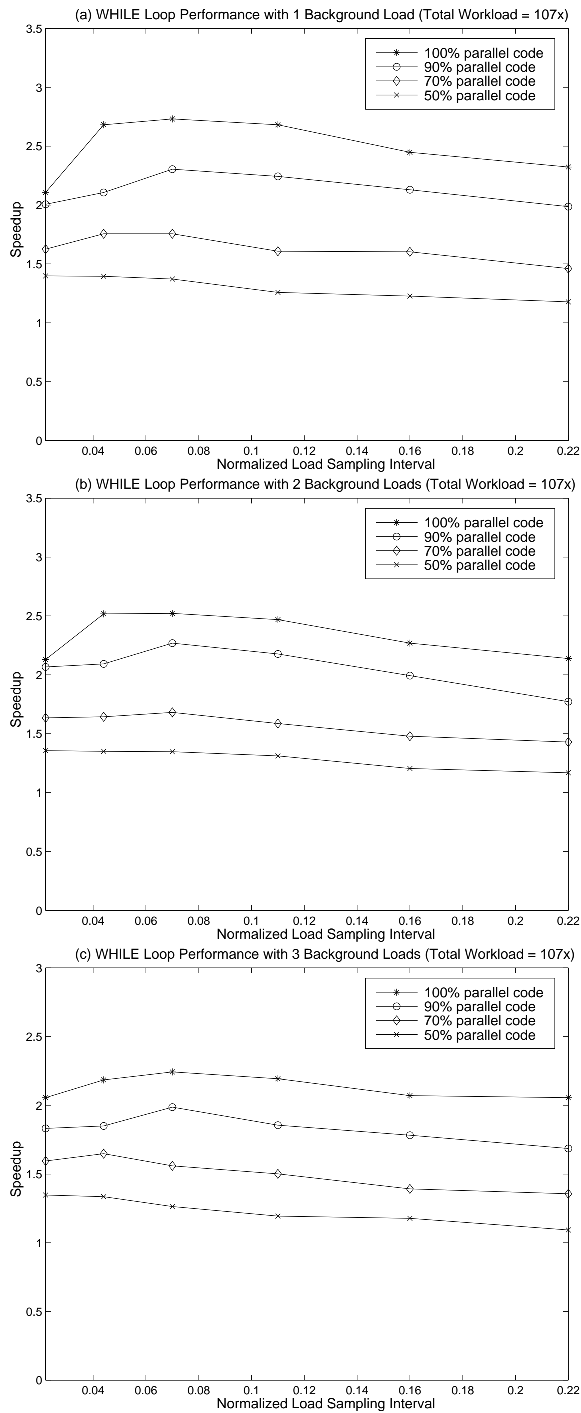

Figure 5.25: Effect of load sampling interval on parallel execution performance of a *while* loop with a normalized workload of 107x.

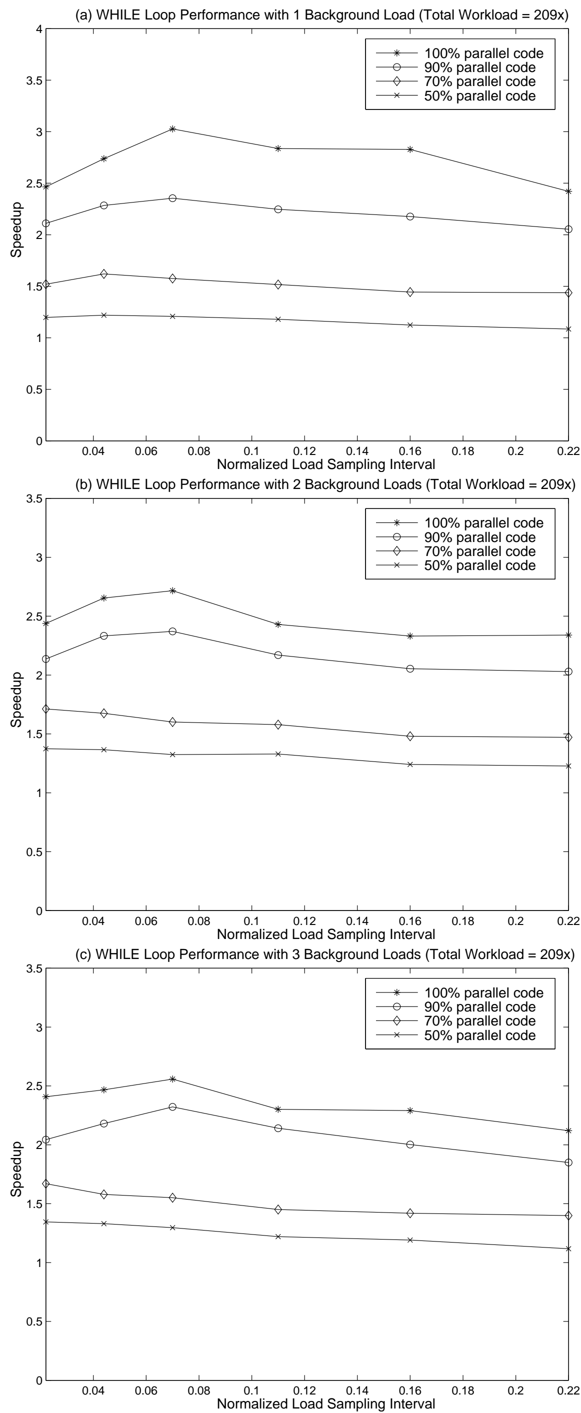

Figure 5.26: Effect of load sampling interval on parallel execution performance of a *while* loop with a normalized workload of 209x.

type loops are estimated using the current loop iteration count (which is always available before the loop is actually executed) and the profiled per iteration execution time. Thus, the run-time system always has a reasonably accurate estimate of the sequential execution time for each invocation of the loop.

As discussed in Chapter 4, the estimation of sequential time for *do-while* type loops cannot be done in such a straight-forward manner. In the absence of the actual value for the number of times such a loop will be executed, the parallelization heuristic makes its decision based on execution profiles of the loop in previous invocations. Specifically, if the profiled loop is parallelizable, the current invocation is also considered parallelizable. The execution profile must be updated if there is a significant change in loop workload to reflect the change in future decisions. Now, we can update the loop profile as soon as there is a significant change in execution time. Let us call this approach the *immediate-update* heuristic. Changing the loop profile immediately, however, is likely to degrade the overall performance of loops that exhibit spurious changes in their loop workload, specifically changes from a large workload to a very small workload. Thus, instead of updating the loop profile immediately after such a change, if the run-time system waits for a few executions of the loop before updating the profile, it will be able to avoid performance penalty due to spurious changes. Let us call this approach the *delayed-update* heuristic. In our current implementation of the dynamic processor allocation scheme, the run-time system updates the profile only after two such consecutive loop behavior.

A synthetic program with dynamically varying program characteristics was used to show the effectiveness of the delayed-update based parallelization heuristic used in dynamically allocating processors to *do-while* type loops. The synthetic program consists of a *do-while* loop, shown in Figure 5.27, whose loop termination is determined by a conditional statement on the ith element of the array `A`. By setting an appropriate element in `A` to `test_val`, we can control the number of times the loop will be executed and, thus, can control the loop workload.

In the following experiments, we control the loop workload explicitly in a way so that either the

```
i = 0;
while(A[i] == test_val){

   /* dummy loop to simulate useful work */
   for(j=0;j<W;j++);

   i++;
}
```

Figure 5.27: Synthetic loop to compare the performance of different parallelization heuristics for *do-while* loops.

6-processor parallel loop execution is the most profitable execution, or the sequential loop execution is the most profitable execution. We show the execution-time performance of the synthetic program using four different approaches. First, the program is executed sequentially on a single processor of the Sun SparcServer system. The program is then executed using the dynamic processor allocation approach on a dedicated 6-processor system. We used the immediate-update and the delayed-update heuristics separately in the dynamic processor allocation scheme. Finally, we executed a hand-tuned version of the program, where we execute the loop in parallel on 6 processors if it has a large workload and execute it sequentially if the loop workload is small. Since the loop workload is controlled explicitly, we know beforehand which invocation will be parallel and thus, can tune the program accordingly to generate the optimized code.

First, we executed the synthetic program with the loop invoked 500 times in the program. For each invocation, the loop had either a sequential workload (loop iteration count = 5) or a parallel workload (loop iteration count = 100). The loop workload was selected to be randomly parallel (using a uniform random number generator) with a given probability, p. There were five different runs of the program with $p = 0.9$, 0.7, 0.5, 0.3, and 0.1. A probability of 0.7, for example, would result in about 70% of the loop invocations to have a parallel workload. The remaining 30% of the invocations would correspond to a sequential workload.

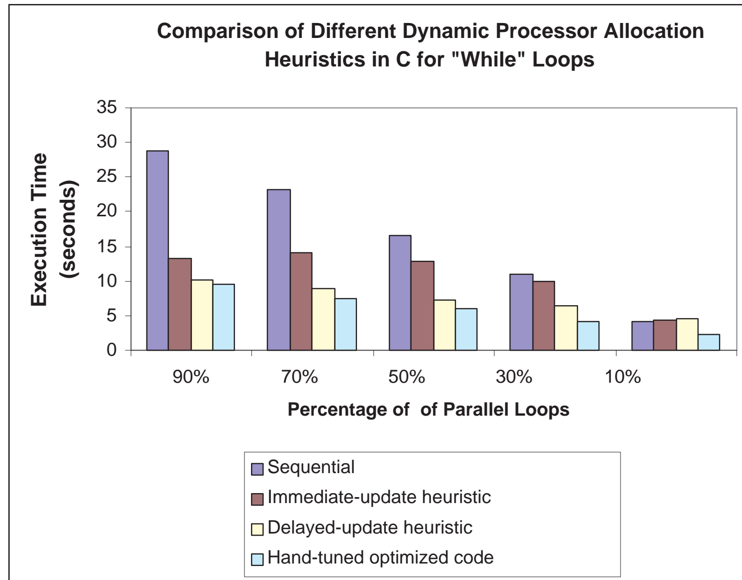

Figure 5.28: Comparison of different parallelization heuristics in the C implementation of the dynamic processor allocation scheme.

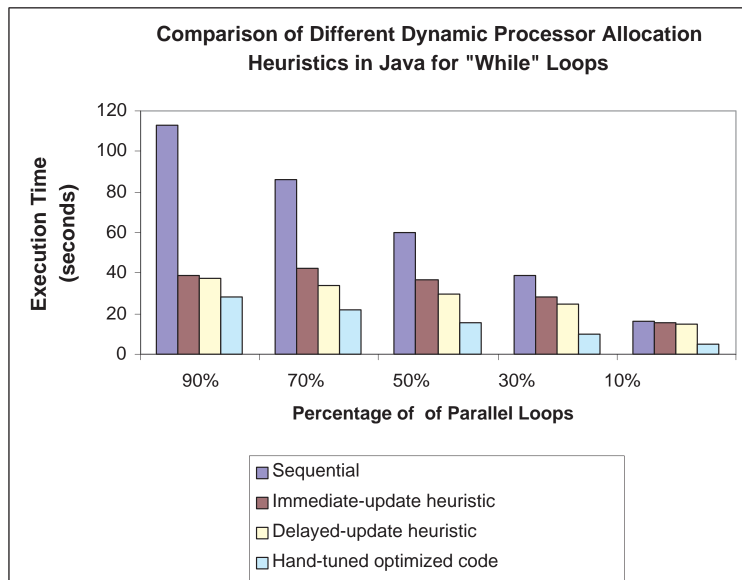

Figure 5.29: Comparison of different parallelization heuristics in the Java implementation of the dynamic processor allocation scheme.

Figure 5.28 and 5.29 show the execution time performance of the C and the Java versions of the synthetic program, respectively. In the C implementation, the dynamic processor allocation scheme with the delayed-update heuristic performs significantly better than with the immediate-update heuristic, except for the last case where the loops are mostly serial ($p = 0.1$). For the mostly parallel case ($p = 0.9$), delayed-update performed within 6.1% of the hand-tuned optimized code whereas immediate-update showed a 37% increase in execution time compared to the hand-tuned code. As the probability of the loop being parallel decreases, the performance difference between the dynamic processor allocation scheme and the hand-tuned code increases. For example, with a 0.5 probability of the loop being parallel, delayed-update resulted in a 23% increase while immediate-update resulted in 117% increase in execution time over the optimized code. As more sequential loops get interleaved with the parallel loops, the dynamic processor allocation scheme makes more frequent updates in the loop execution profile. This results in more of the parallel loops being executed sequentially as a result of previous changes in execution profile. Thus, the performance degrades as the probability of the loop being parallel decreases. In the mostly serial loop ($p = 0.1$) case, both heuristics performed equally well resulting in a performance close to the sequential execution performance. The results from the Java implementation in Figure 5.29 showed similar performance with the delayed update approach performing better than immediate-update approach in most cases.

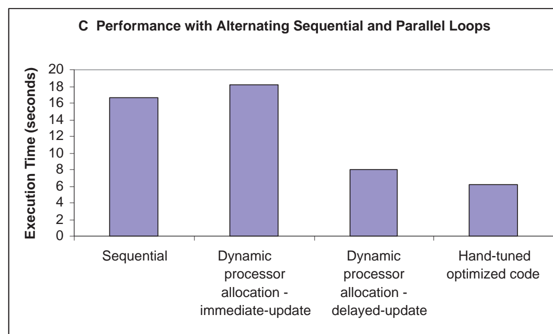

Figure 5.30: Comparison of different parallelization heuristics in the C implementation with alternating parallel and sequential loop workload.

To further show the effectiveness of the delayed-update heuristic over the immediate-update

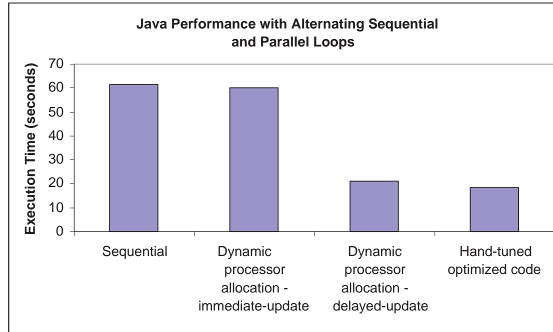

Figure 5.31: Comparison of different parallelization heuristics in the Java implementation with alternating parallel and sequential loop workload.

heuristic, we considered an extreme case where the loop workload alternately changed between parallel and sequential. Figure 5.30 shows the performance of this particular case with the C implementation. The delayed-update heuristic was able to correctly classify the parallel loops since it did not change the profile in response to a single sequential run in between parallel runs. Thus, it performed very well with an execution time comparable to the execution time of the optimized code. The immediate-update heuristic, however, showed poor performance. This approach resulted in sequential execution of the loop for all the invocations and, thus, the execution time is almost the same as the sequential execution time. The results from the Java implementation shown in Figure 5.31 also prove the superiority of the delayed-update heuristic over the immediate-update heuristic.

5.5 Conclusion

In this chapter, we have evaluated the performance of the speculative multithreading parallelization model using some real application programs. Using this parallelization technique, we were able to parallelize standard DOALL and DOACROSS loops as well loops with indeterminate termination conditions and loops with run-time data dependences. Performance results on the SGI Challenge shared-memory multiprocessor system presented in Section 5.3 showed that application programs

(both C and Java) with sufficiently large granularity compared to the parallelization overhead can exhibit significant performance gains when parallelized using the speculative multithreading model.

Due to the underlying difference in language-specific implementations of thread packages, the C implementation showed relatively better performance than the Java implementation. Comparison with an existing run-time parallelization technique for compiled languages, specifically, the Privatizing DOALL Test [37], demonstrated that the speculative multithreading parallelization model can improve performance by allowing concurrent threads to execute in a pipelined fashion. Further comparison of the Java implementation of the speculative multithreading model with a traditional Java parallelization technique, specifically, the JAVAR parallelization tool [4], showed slightly lower performance for JavaSpMT. This lower performance is acceptable since JavaSpMT can extract parallelism from a wide variety of loop constructs that the traditional techniques are unable to do.

We applied the dynamic processor allocation scheme to the programs parallelized using the speculative multithreading model on a 6-processor Sun SparcServer 1000 system. The performance results presented in Section 5.4 show that the comprehensive dynamic processor allocation scheme is quite effective in dynamically reallocating the number of processors to a parallel application to achieve the best performance for both a program's varying behavior and the varying system load. Comparisons with the time-shared mechanism in a multiprogrammed environment further showed that parallel applications executed with the dynamic processor allocation-based system provides better performance than with time-sharing, especially when several applications are executing in the system.

Chapter 6

Conclusion

A wide variety of application programs exhibit a large amount of coarse-grained loop-level parallelism, which if exploited successfully, can lead to significant performance improvement. Traditional parallelizing compilers, however, often cannot extract much of the parallelism inherent in application programs due to insufficient compile-time information. This thesis addressed some of the problems associated with compiler parallelized programs through a dynamically adaptive parallelization model based on speculative multithreading. The speculative multithreading parallelization model allows us to extract loop-level parallelism from general-purpose programs which otherwise would be impossible to parallelize using traditional techniques. Furthermore, the dynamic adaptation of the parallel programs (through dynamic processor allocation) allows the programs to execute efficiently on a multiprogrammed shared-memory multiprocessor system.

Existing parallelizing compilers cannot parallelize loops if they are unable to successfully analyze the data-dependence relations across iterations or are unable to determine the number of times the loop will be executed. Thus, loops with run-time data dependences or loops based on traditionally sequential constructs (i.e., loops with an unknown iteration spaces) cannot be parallelized by these compilers. General-purpose applications, unlike scientific applications, often contain these types of loop structures and, thus, are impossible to parallelize. Furthermore, even if the available parallelism

can be extracted by a compiler, it is not guaranteed that the resulting parallel code will provide better performance over sequential execution. Parallelizing compilers usually map a parallel application to all the processors in the system. This static mapping may result in poor performance as well as wasting processor resources, even in a dedicated system, if a parallel code region cannot fully utilize all the processors due to an insufficient workload. Additionally, the performance of compiler-parallelized programs may suffer in a multiprogrammed system due to severe contention for processor resources.

6.1 Thesis Contributions

This thesis proposed a dynamically adaptive parallelization model based on speculative multithreading to address the problems of existing parallelizing compilers. A run-time parallelization model, called *speculative multithreading*, is proposed to exploit loop-level parallelism from general-purpose application programs with run-time data dependences and unknown iteration spaces. A *comprehensive dynamic processor allocation* scheme is proposed to efficiently execute parallel programs on multiprogrammed shared-memory multiprocessor systems by allocating processors based on program behavior and system load.

The speculative multithreading parallelization model [23, 24] proposed in this thesis extends the fine-grained thread pipelining model of the superthreaded processor architecture [44, 45, 46] to parallelize loops with run-time data dependences and traditionally sequential loop constructs. The parallelization model executes loop iterations concurrently in a pipelined fashion with run-time data-dependence checking and control speculation. The run-time data-dependence checking allows loops for which complete data access information is not available at compile-time to be easily parallelized. Because of the pipelined execution of concurrent threads, the data-dependence checking and execution of successive loop iterations are partially overlapped. This makes our parallelization approach better than traditional *inspector-executor* based run-time parallelization techniques that require a

separate *executor* stage to implement the run-time dependence checking. The control speculation in our model further makes it possible to parallelize traditionally sequential loop constructs such as *do-while* loops.

The comprehensive dynamic processor allocation scheme [25] allows parallel application programs to adapt to the current execution environment for the most efficient execution in a multiprogrammed environment. Run-time information about program workload, target machine configuration, and current system load are used to allocate an appropriate number of processors to a parallel application program without overloading the system. Based on the program's behavior, the dynamic processor allocation system determines the number of processors a parallel code region can profitably use. The program is allocated as many processors as are currently available, up to the maximum number it can efficiently utilize. If the code region cannot be profitably executed in parallel, or if there are no processors available to allow parallel execution, the code region is dynamically serialized. The processor allocation decision is dynamically updated as both the program behavior and the system load change during a program's execution. By allocating as many processors as a program can efficiently utilize, our processor allocation scheme guarantees the best performance for the program and eliminates wastage of processor resources at the same time. Allocation of processors based on system load further improves parallel execution performance by eliminating contention for processor resources.

Special library support for the speculative multithreading model was developed, both in C and Java, to parallelize C and Java application programs, respectively. The performance of the speculative multithreading parallelization model was evaluated using some real application programs on an 8-processor SGI-Challenge shared-memory multiprocessor system. Using this parallelization technique, we were able to parallelize standard DOALL and DOACROSS loops as well loops with indeterminate termination conditions and loops with run-time data dependences. Both C and Java application programs with sufficiently large granularity compared to the parallelization overhead resulted in significant performance gains when parallelized using the speculative multithreading

model.

Comparison of the performance of the C implementation of the speculative multithreading parallelization model to that of an existing *inspector-executor* based run-time parallelization technique showed that the proposed model performs better due to the pipelined execution of threads. Further comparison of the Java implementation of the speculative multithreading model (JavaSpMT) with a traditional Java parallelization technique showed slightly lower performance for JavaSpMT. This lower performance is acceptable considering the additional overhead required in JavaSpMT to extract parallelism from a wide variety of loop constructs that are impossible to parallelize using the traditional techniques.

The comprehensive dynamic processor allocation scheme was applied to programs parallelized using the speculative multithreading model. A library, implemented on the Solaris operating system, is provided to allow parallel programs to dynamically adapt to the current execution environment. We evaluated the performance of the dynamic processor allocation scheme on a 6-processor Sun SparcServer 1000 system with the same application programs used in evaluating the speculative multithreading parallelization model. The performance results showed the effectiveness of our dynamic processor allocation scheme in dynamically reallocating the number of processors to a parallel application program to achieve the best performance as both a program's workload and the system load vary during program execution. Comparison with the time-shared mechanism in a multiprogrammed environment further showed that parallel applications executed with our dynamic processor allocation-based system resulted in much better performance than with time-sharing when multiple applications executed simultaneously in the system.

6.2 Future Work

To further improve the performance of application programs parallelized using the speculative multithreading parallelization model, a number of alternative design approaches for the thread execution

model can be explored in the future. One alternative is to use data speculation to increase the degree of concurrency in thread execution. In the current implementation of the parallelization model, run-time data dependences are enforced through synchronization flags based on the outcome of the run-time data-dependence test performed in each thread's TSAG stage. A thread cannot proceed until its predecessor thread has completed its TSAG stage. While this approach works well if data dependences actually exist among the threads, it limits the available parallelism if there are no data dependences. To improve performance in the latter case, the threads can be executed with data speculation. A test for data-dependence violation must be done before the write-back stage. In case of a violation, the thread must be re-executed with the updated values of the memory locations. Because of the overhead associated with data misspeculation, this approach may result in poor performance if data dependences actually occur most of the time.

The threads in the speculative multithreading model retire in the original sequential order. This execution model with in-order completion of threads works well provided the workload among the threads is well-balanced. In the current implementation, it is assumed that a balanced workload can be guaranteed by the compiler or the programmer (if parallelization is done manually). If the workload among different threads varies significantly, however, performance may suffer as successor threads that finish early may be idle for a long time, thereby wasting processor resources. One approach to address this problem is to allow processors to switch between threads, as is done in the thread level data speculation (TLDS) scheme [43]. When a processor reaches the write-back stage of the current thread, it can suspend the thread and begin executing a new thread. When the thread for which the processor was waiting retires, it can switch back to the suspended thread to complete its write-back stage. To allow thread switching, the state of a thread must be saved upon suspension and restored when it is resumed again. Since the thread switching will be associated with additional overhead, this approach may not be feasible unless the thread waiting time due to load imbalance is quite significant.

The comprehensive dynamic processor allocation scheme uses an estimation of the original se-

quential execution time of a loop to make its processor allocation decision. In the current implementation, this estimation is based on an execution profile generated at run-time. This run-time profiling may result in performance loss if the target loop has a sufficiently large workload to benefit from parallel execution. As discussed in Section 4.1.3, a number of alternative approaches to profiling can be used to improve performance in such cases. One approach to generating an execution profile is to execute the program once sequentially on a base machine and profile the target loops, as is done in the dynamic serialization approach [48, 49]. The base machine timings then can be scaled to the target machine by using an appropriate scaling factor for the target machine. A scaling factor for the target machine has to be determined by executing some benchmark code on both the base machine and the target machine. Another approach is to determine the instruction count in each loop iteration by performing a static analysis of the loop. The per-iteration execution time of the loop can be obtained by multiplying this instruction count by an estimate of the average instruction execution time on the target machine. The run-time system for the dynamic processor allocation scheme can be extended in the future to incorporate these alternative profiling techniques.

The Java implementation of the comprehensive dynamic processor allocation scheme currently uses the Java interpreter as its run-time system. Since Java applications are frequently executed using a just-in-time (JIT) compiler, the dynamic processor allocation scheme should be incorporated in the JIT compiler code. Java codes execute much faster (about 2-10 times) with a JIT compiler than with an interpreter [26]. The load sampling mechanism in the current Java implementation (using the `top` Unix utility) may result in too much overhead compared to the execution time of the JIT-compiled codes. In this case, the system load should be determined using direct system calls, as is done in the C implementation.

6.3 Summary

The dynamically adaptive parallelization model based on speculative multithreading exploits loop-level parallelism from general-purpose application programs and allows the transformed parallel programs to dynamically adapt to a varying execution environment for the best performance. The speculative multithreading parallelization model is a software-based approach to exploit parallelism from loops with run-time data dependences and unknown iteration spaces that are difficult to parallelize using traditional parallelization techniques. Furthermore, the dynamic processor allocation scheme allows parallel programs to execute efficiently, under dynamically varying program and system behavior, in a multiprogrammed system. Application-level performance, in both C and Java, showed the effectiveness of the parallelization model in improving performance by exploiting loop-level parallelism from programs with a variety of loop constructs. The dynamic processor allocation scheme allowed the parallel application programs to dynamically adapt to varying program behavior and varying system load. By allocating an appropriate number of processors to a parallel task, based on program workload and system load, this processor allocation scheme resulted in better performance than a standard time-shared approach.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass. 1986.

[2] G.R. Andrews, *Concurrent Programming Principles and Practice*, Benjamin/Cummings Publishing Company Inc., 1991.

[3] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad, *Fast, Effective Dynamic Compilation*, Proceedings of the SIGPLAN '96 Conference on Program Language Design and Implementation, May 1996, pp. 149-159.

[4] A. Bik and D. Gannon, *Automatically Exploiting Implicit Parallelism in Java*, Concurrency: Practice and Experience, 9(6), June 1997, pp. 579-619.

[5] A. Bik and D. Gannon, *Javab - A Prototype Bytecode Parallelization Tool*, ACM Workshop on Java for High-Performance Network Computing, 1998.

[6] W. Blume et. al., *Effective Automatic Parallelization with Polaris*, International Journal of Parallel Programming, May 1995.

[7] M.G. Burke et. al., *The Jalapeno Dynamic Optimizing Compiler for Java*, Proceedings of the ACM 1999 Conference on Java Grande, 1999, pp. 129 - 141.

[8] M. Byler, J.R.B. Davies, C. Huson, B. Leasure, and M. Wolfe, *Multiple Version Loops*, International Conference on Parallel Processing, August 1987, pp. 312-318.

[9] D. Caromel et. al., *Towards Seamless Computing and Metacomputing in Java*, Concurrency: Practice and Experience, Sept.-Nov. 1998, pp. 1043-1061.

[10] B. Carpenter et. al., *HPJava: Data Parallel Extensions to Java*, Concurrency: Practice and Experience, 1998.

[11] D.K. Chen, P.C. Yew, and J. Torrellas, *An Efficient Algorithm for the Run-time Parallelization of Doacross Loops*, Supercomputing, Nov. 1994, pp. 518-527.

[12] G. Cybenko, *Supercomputer Performance Trends and the Perfect Benchmarks*, Supercomputing Review, April 1991, pp. 53-60.

[13] L. Dagum and R. Menon, *OpenMP: An Industry-Standard API for Shared-Memory Programming*, IEEE Computational Science and Engineering, Vol. 5, No. 1, January/March 1998.

[14] P.K. Dubey et. al., *Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading*, Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, June 1995, pp. 109-121.

[15] David Griswould, *The Java HotSpot Java Virtual Machine Architecture*, http://java.sun.com/products/hotspot/whitepaper.html.

[16] A. Gupta, A. Tucker, and S. Urushibara, *The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications*, Conference on the the Measurement and Modeling of Computer Systems, 1991, vol. 19, pp. 120-132.

[17] Mary W. Hall and Margaret Martonosi, *Adaptive Parallelism in Compiler-Parallelized Code*, Proceedings of the 2nd SUIF Compiler Workshop, August 1997.

[18] M.W. Hall et. al., *Maximizing Multiprocessor Performance with the SUIF Compiler*, IEEE Computer, Vol. 29, No. 12, Dec. 1996.

[19] Y. Ichisugi and Y. Roudier, *Integrating Data-parallel and Reactive Constructs into Java*, France-Japan Workshop on Object-based Parallel and Distributed Computation, 1997.

[20] V. Ivannikov et. al., *DPJ: Java Class Library for Development of Data-parallel Programs*, Institute for Systems Programming, Russian Academy of Sciences, 1997, http://www.ispras.ru/˜ dpj/.

[21] M. Johnson, *Superscalar Microprocessor Design*, Prentice-Hall Inc., 1991.

[22] L. Kale et. al., *Design and Implementation of Parallel Java with Global Object Space*, Conference on Parallel and Distributed Processing Technology and Applications, 1997.

[23] I.H. Kazi and D.J. Lilja, *Coarse-grained Speculative Execution in Shared-Memory Multiprocessors*, International Conference on Supercomputing, July 1998, pp. 93-100.

[24] I.H. Kazi and D.J. Lilja, *JavaSpMT: A Speculative Thread Pipelining Parallelization Model for Java Programs*, Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), May 2000, pp. 559-564.

[25] I.H. Kazi and D.J. Lilja, *A Comprehensive Dynamic Processor Allocation Scheme for Multiprogrammed Multiprocessor Systems*, to be published in the International Conference on Paralell processing (ICPP), August 2000.

[26] I.H. Kazi, H. Chen, B. Stanley, and D.J. Lilja, *Techniques for Obtaining High Performance in Java Programs*, High-Performance Parallel Computing Research Group Technical Report No. HPPC-99-01, January 1999.

[27] D.E. Kuller and J.P. Singh, *Parallel Computer Architecture: A Hardwrae/Software Approach*, Morgan Kauffmann Publishers, Inc., 1999.

[28] M.S. Lam and R.P. Wilson, *Limits of Control Flow on Parallelism*, Proceedings of the International Symposium of Computer Architecture, May 1992, pp. 46-57.

[29] D.J. Lilja, *Exploiting the Parallelism Available in Loops*, IEEE Computer, Vol. 27, No. 2, February 1994, pp. 13-26.

[30] R.E. Ladner and M.J. Fischer,*Parallel Prefix Computation,* Journal of the ACM, Vol. 27, No. 4, October 1980, pp. 831-832.

[31] P. Launay and J. L. Pazat, *A Framework for Parallel Programming in Java*, IRISA, France, Technical Report 1154, Dec. 1997.

[32] V. Naik, S. Setia, and M. Squillante, *Performance Analysis of Job Scheduling Policies in Parallel Supercomputing Environments*, Supercomputing, 1993, pp. 824-833.

[33] J.A. Fisher, *Very Long Intrsuction Word Architectures and the ELI-512*, Proceedings of the 10th Annual Symposium on Computer Architectures, June, 1983, pp. 140-150.

[34] J. Oplinger et. al., *Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor*, Technical Report CSL-TR-97-715, Stanford University Computer Systems Laboratory, Feb. 1997.

[35] J. Osterhout, *Scheduling Techniques for Concurrent Systems*, In Distributed Computing Systems Conference, 1982, pp. 22-30.

[36] M. Philippsen and M. Zenger, *JavaParty - Transparent Remote Objects in Java*, Concurrency: Practice and Experience, 9(11), 1997, pp. 1125-1242.

[37] L. Rauchwerger and D. Padua, *The PRIVATIZING DOALL Test: A Run-time Technique for DOALL Loop Identification and Array Privatization*, SIGPLAN 1994 Conference on Supercomputing, July 1994, pp. 33-43.

[38] L. Rauchwerger, N. Amato, and D. Padua, *Run-time Methods for Parallelizing Partially Parallel Loops*, Supercomputing, 1995, pp. 137-145.

[39] L. Rauchwerger and D. Padua, *Parallelizing While Loops for Multiprocessor Systems*, Proceedings of the 9th International Parallel Processing Symposium, pp. 347-356, April 1995.

[40] L. Rauchwerger, *Run-Time Parallelization: A Framework for Parallel Computation*, Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[41] C. Severance et. al., *Automatic Self-Allocating Threads on the Convex Exemplar*, Proceedings of the International Conference on Parallel Processing, 1995, pp. I-24-31.

[42] G.S. Sohi et. al., *Multiscalar Processors*, Proceedings of the 22nd Annual International Symposium on Computer Architecture, June 1995, pp. 414-425.

[43] J. G. Steffan, C. B. Colohan, and T. C. Mowry, *Architectural Support for Thread-Level Data Speculation*, Technical Report CMU-CS-97-188, School of Computer Science, Carnegie Mellon University, Nov. 1997.

[44] J.Y. Tsai and P.C. Yew, *The Superthreaded Architecture: Thread Pipelining with Run-time Data Dependence Checking and Control Speculation*, Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96), Oct. 1996, pp. 35-46.

[45] J. Y. Tsai, Z. Jiang, E. Ness, and P.C. Yew, *Performance Study of a Concurrent Multithreaded Processor*, Proceedings of the Fourth International Conference on High-Performance Computer Architecture (HPCA-4), Feb. 1998.

[46] J. Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P. C. Yew, *The Superthreaded Processor Architecture*, IEEE Transactions on Computers, Special Issue on Multithreaded Architectures and Systems, September 1999, pp. 881-902.

[47] A. Tucker, *Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors*, PhD thesis, Dept. of Computer Science, Stanford University, 1993.

[48] Michael J. Voss and Rudolf Eigenmann, *Reducing Parallel Overheads Through Dynamic Serialization*, International Parallel Processing Symposium, April 1999, pp 88-92.

[49] Michael J. Voss and Rudolf Eigenmann, *Dynamically Adaptive Parallel Programs*, In International Symposium on High Performance Computing, 1999, pp. 109-120.

[50] R. Wilson et. al., *SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers*, ACM SIGPLAN Notices, 29(12), December 1994, pp 31-37.

[51] Kelvin K. Yue and David J. Lilja, *Efficient Execution of Parallel Applications in Multiprogrammed Multiprocessor Systems*, Proceedings of the International Parallel Processing Symposium, April 1996, pp. 448-456.

[52] Kelvin K. Yue and David J. Lilja, *Dynamic Processor Allocation with the Solaris Operating System*, Proceedings of the International Parallel Processing Symposium, March 1998, pp. 392-397.

[53] Y. Zhang et. al., *Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors*, Fourth International Conference on High-Performance Computer Architecture, Feb. 1998, pp. 162-173.

[54] C.Q. Zhu and P.C. Yew, *A Scheme to Enforce Data Dependence on Large Multiprocessor Systems*, IEEE Transactions on Software Engineering, SE-13(6), June 1987, pp. 726-739.

[55] *Java 2 Platform*, http://java.sun.com/java2/.

[56] *Soalris Product Line: Solaris 2.6 Operating Environment*, http://www.sun.com/software/-solaris/2.6/ds-solaris26.html.

[57] *SPEC CPU2000*, http://www.specbench.org/osg/cpu2000/.