Adaptive Execution Techniques for SMT Multiprocessor Architectures[*] [†]

Changhee Jung
Electronics and Telecommunication Research Institute
Yuseong-Gu, Daejeon, 305-530, Korea
chjung@etri.re.kr

Daeseob Lim
Dept. of Computer Science and Engineering
University of California, San Diego
9500 Gilman Drive, La Jolla, CA 92093-0114
dalim@cse.ucsd.edu

Jaejin Lee
School of Computer Science and Engineering
Seoul National University
Seoul 151-744, Korea
jlee@cse.snu.ac.kr

SangYong Han
School of Computer Science and Engineering
Seoul National University
Seoul 151-744, Korea
syhan@pandora.snu.ac.kr

ABSTRACT

In simultaneous multithreading (SMT) multiprocessors, using all the available threads (logical processors) to run a parallel loop is not always beneficial due to the interference between threads and parallel execution overhead. To maximize performance in an SMT multiprocessor, finding the optimal number of threads is important. This paper presents adaptive execution techniques to find the optimal execution mode for SMT multiprocessor architectures. A compiler preprocessor generates code that, based on dynamic feedback, automatically determines at run time the optimal number of threads for each parallel loop in the application. Using 10 standard numerical applications and running them with our techniques on an Intel 4-processor Hyper-Threading Xeon SMP with 8 logical processors, our code is, on average, about 2 and 18 times faster than the original code executed on 4 and 8 logical processors, respectively.

[*]This work was supported in part by the Ministry of Education under the Brain Korea 21 Project in 2005, by the Ministry of Science and Technology under the National Research Laboratory program, and by the Institute of Computer Technology at Seoul National University.

[†]Correspondence to `jlee@cse.snu.ac.kr`. Changhee Jung and Daeseob Lim did their work when they were students of Seoul National University.

Categories and Subject Descriptors

D.1 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms

Performance

Keywords

simultaneous multi-threading, adaptive execution, compilers, performance counters, performance estimation

1. INTRODUCTION

The simultaneous multithreading architecture (SMT) [9, 19, 15] allows a superscalar processor to simultaneously maintain the context of multiple threads while making the instructions from different threads execute in parallel. SMT processors can increase the performance of parallel (multi-threaded) applications by increasing utilization of the processor's internal resources. However, identical instruction streams from multiple threads created by conventional loop-level parallel programs may have poor instruction-level parallelism for SMT architectures. This results in overall performance degradation due to contention of processor's internal resources in the SMT architecture. In a multiprocessor system with multiple SMT processors, frequent execution of synchronization instructions may worsen the contention between threads. Moreover, shared caches between different threads in an SMT processor may cause conflicts and decrease the hit rate. Consequently, to obtain reasonable performance from an SMT multiprocessor system, programmers need a detailed knowledge of the underlying system and the ability to fine-tune parallel applications.

High performance optimizing and parallelizing compilers perform various optimizations to improve performance of sequential and parallel programs [2]. Performance of an algorithm that best solves a problem largely depends on

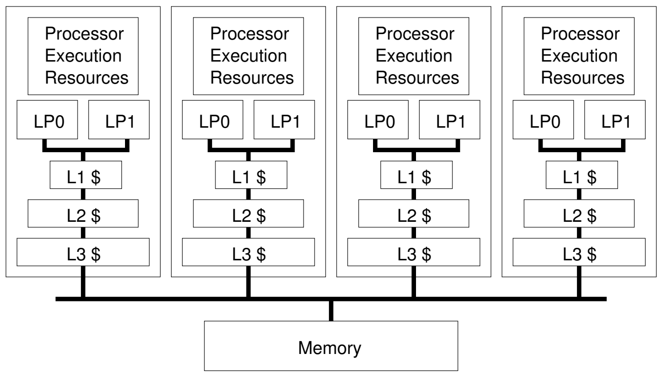

Figure 1: Target SMT multiprocessor architecture.

the combination of the input set and the hardware platform executing it. Information about the input set and the platform is difficult to obtain or unavailable at compile time. Consequently, it is difficult or sometimes impossible to statically fine-tune the application to the platform. To address this issue, recent work [1, 8, 17, 22, 26, 27] has begun to explore the feasibility of run-time fine-tuning and optimizations when complete knowledge about the input set and hardware platform is available.

Automatic parallelizing compilers analyze and transform a sequential program into a parallel program without any user intervention of the user. However, to achieve maximum performance from automatically or manually parallelized programs, the user must consider many factors related to the underlying SMT multiprocessor architecture: the amount of parallelism contained in the program, the cache locality of the program, the cache coherence mechanism, workload distribution, data distribution, false sharing, the coordination overhead incurred between threads, synchronization overhead, cache conflicts, and the processor's internal resource contention. Since these factors manifest synergistically on the performance and the effects differ from one machine to another, identifying performance bottlenecks of a parallel program is a tedious and difficult job for the programmer. Consequently, the programming cost to obtain reasonable performance from a parallel program contributes to the difficulty of developing and maintaining parallel programs.

Even though SMT is more useful for workloads that do not have enough extratable instruction-level parallelism, the performance of conventional loop-level parallel programs on SMT multiprocessors is becoming important as SMT multiprocessor systems are becoming popular and have a widening user base.

In this paper, instead of manually identifying the performance bottlenecks of parallel programs running on SMT multiprocessor architectures, we avoid executing some parallel loops in parallel or change the number of threads to run the loops optimally if the performance degradation of the loops exceeds a predefined threshold value at run time. No knowledge of the parallel program is assumed from the programmer. Such adaptive execution techniques significantly increase parallel program performance in SMT multiprocessor architectures. A compiler preprocessor generates code that automatically determines at run time the optimal number of threads for each parallel loop in the application. This decision is based on dynamic feedback from performance counters.

Using a set of standard numerical applications and running them with our technique on a real SMT multiprocessor machine [19] with 8 hardware contexts (a symmetric multiprocessor machine with four Intel Hyper-threading Xeon MP processors), our code is, on average, about 2 and 18 times faster than the original code executed on 4 and 8 logical processors (threads), respectively. We used a quad Intel Xeon multiprocessor system [19] as our target. However, the same technique will also work with other SMT multiprocessor systems, such as the IBM Power5 [15], as long as the adaptive execution parameters are tuned to the target machine.

The rest of the paper is organized as follows: Section 2 describes our target architecture and parallel programming and execution model; Section 3 presents our adaptive execution schemes; Section 4 describes the evaluation environment; Section 5 evaluates our technique; and Section 6 discusses related work, and Section 7 concludes the paper.

2. TARGET ARCHITECTURE AND OUR FRAMEWORK

Figure 1 shows our target SMT multiprocessor system. The architecture is similar to the quad Intel Xeon with hypertheading[19]. The machine contains four SMT processors, each of which has two copies of the architectural state (LP0 and LP1). Thus, the system appears to have eight logical processors. There are L1, L2, and L3 data caches in each physical processor and the two logical processors in a physical processor share them.

We use the OpenMP parallel programming model, which is master-slave thread model, for our parallel execution [21, 6, 12, 13]. Each thread is mapped to one logical processor in the system at run time. We denote the thread-processor mapping with the notation mPnL, where m are n are the number of physical processors and the number of logical processors used in the mapping, respectively. In addition, n is the number of all threads used. For example, 2P2L means that two threads are mapped to two physical processors and each thread is mapped to LP0 in each physical processor. 2P4L means that four threads are mapped to

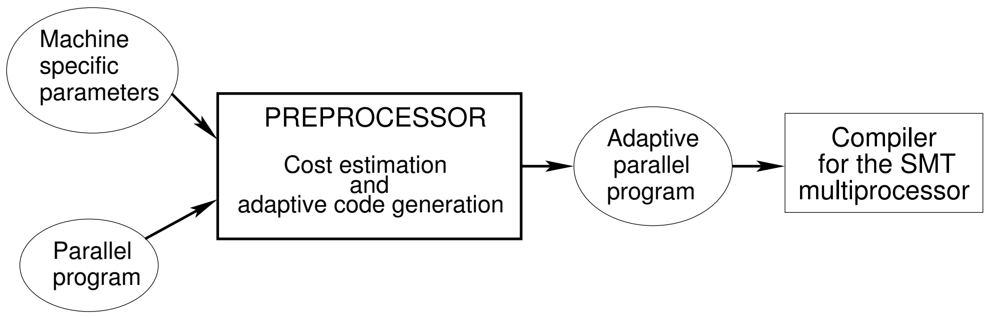
Machine specific parameters
PREPROCESSOR
Cost estimation and adaptive code generation
Parallel program
Adaptive parallel program
Compiler for the SMT multiprocessor

Figure 2: Our framework.

two physical processors and there is a one-to-one correspondence between the threads and logical processors in the two processors. In 4P1L execution mode, only one thread is running on only one logical processor in the system (i.e., a parallel loop is executed sequentially).

When loops are run in parallel, an overhead is incurred due to the creation of slave threads, the distribution of loop iterations between threads, and the synchronization between threads at the end of a loop. In addition, because two logical processors in a physical processor share cache hierarchies and other internal resources, interference and cache conflicts exist between the two logical processors. We call these types of overhead *parallel loop overhead* in an SMT multiprocessor system.

Figure 2 shows our framework. An automatically or manually parallelized program is fed into the compiler preprocessor. The preprocessor inserts performance estimation and adaptive execution code into the parallel program using target machine specific parameters. The output program from the preprocessor is compiled with a compiler that generates code for the SMT multiprocessor system.

The preprocessor inserts instrumentation code in each parallel loop in order to measure, at run time, the execution time and the numbers of graduated instructions and cache misses during an invocation of the loop. The instrumentation code uses performance counters in the processor. The invocations where the performance counters are read are called *decision runs.* Based on the measurements taken during the decision runs, the adaptation code determines how to execute the loop (i.e., 4P1L, 4P4L, or 4P8L) in the next or remaining invocations.

The preprocessor generates two different versions of a single parallel loop: one is a sequential version that is for 4P1L mode and the other is a parallel version for 4P4L and 4P8L modes. Switching modes between 4P4L and 4P8L is done by changing the number of working threads and controlling thread affinity to logical processors with system calls.

Since most parallel loops in numerical applications are invoked many times, adaptive execution with decision runs is feasible and fairly effective. A drawback of our adaptive execution technique is that we may have some performance penalty caused by decision runs and run-time instrumentation.

3. ADAPTIVE EXECUTION STRATEGIES

An overview of our adaptive execution scheme is shown in Figure 3. It consists of several filtering steps. First, a static filtering scheme filters parallel loops that contain a smaller workload than the parallel loop overhead at compile time using a cost estimation model. An *efficient parallel loop* is the parallel loop whose speedup is greater than 1; otherwise, it is an *inefficient parallel loop.* The first step filters highly inefficient parallel loops at compile time. Then, the run-time cost estimation model filters highly inefficient parallel loops that cannot be handled at compile time. These loops are executed in 4P1L mode. Among the remaining loops, those loops that are highly efficient in 4P4L mode and 4P8L mode are selected next. Then, the loops that are suitable for switching between 4P4L and 4P1L modes are selected. Finally, the remaining loops are executed by switching between 4P4L and 4P8L modes depending on their execution environment. As an alternative to switching between 4P4L and 4P8L modes, we can identify the loops that are suitable for 4P8L mode using a cost model and execute the remaining loops in 4P4L mode.

3.1 Compile-time Cost Estimation

The compile-time cost estimation model identifies parallel loops that are highly inefficient due to an insufficient workload in the loop. It is not beneficial to run this type of loop in parallel because the workload in the loop is fairly small compared to the parallel loop overhead. We define a threshold value for the workload contained in the loop. If the workload is smaller than the threshold value, we run the loop in 4P1L mode (i.e., the loop runs on only one logical processor in the system). We use a fairly simple cost estimation model. The workload (W) in a loop can be estimated as a function of the numbers of iterations(n_i), assignments (n_a), floating point addition ($n_{f_{add}}$), floating point multiplication ($n_{f_{mul}}$), floating point subtraction ($n_{f_{sub}}$), floating point division ($n_{f_{div}}$), intrinsic function calls (n_{fi}), system function calls (n_{fs}), and user defined function calls (n_{fu}). Consequently, the estimated amount of work (W_i) in an iteration is given by the following formula:

$$\begin{aligned} W_i \;=\; & n_a \cdot c_a + n_{f_{add}} \cdot c_{f_{add}} + n_{f_{mul}} \cdot c_{f_{mul}} \\ & + n_{f_{sub}} \cdot c_{f_{sub}} + n_{f_{div}} \cdot c_{f_{div}} \\ & + n_{fi} \cdot c_{fi} + n_{fs} \cdot c_{fs} + n_{fu} \cdot c_{fu} \end{aligned}$$

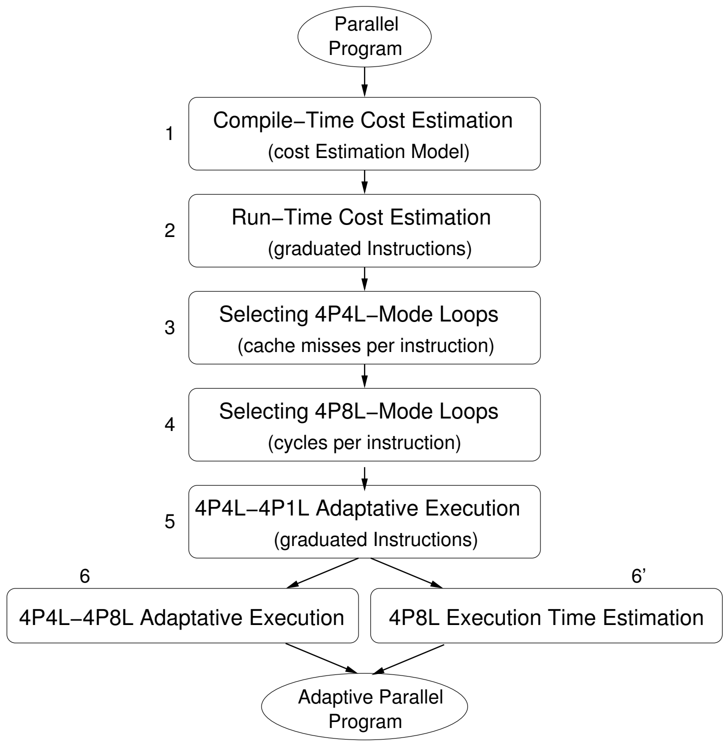
Parallel Program
1
Compile−Time Cost Estimation
(cost Estimation Model)
2
Run−Time Cost Estimation
(graduated Instructions)
3
Selecting 4P4L−Mode Loops
(cache misses per instruction)
4
Selecting 4P8L−Mode Loops
(cycles per instruction)
5
4P4L−4P1L Adaptative Execution
(graduated Instructions)
6
4P4L−4P8L Adaptative Execution
6'
4P8L Execution Time Estimation
Adaptive Parallel Program

Figure 3: Adaptive execution schemes.

where c_{op} is the cost of performing a single operation with type *op*. Thus, the total estimated workload in an invocation of the loop is

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

Because we cannot determine at compile time the actual number of iterations in a loop in general, this formula is parameterized by n_i. When we estimate the workload in a loop that has branches, we give equal weight to each branch.

We determine the threshold value heuristically. First, we run several representative micro-benchmark programs that contain different types of parallel loops. After measuring sequential execution time in 4P1L mode and parallel execution time of each loop in 4P4L mode, we plot the speedup of each loop on the Y-axis and the estimated cost (the estimated workload) on the X-axis. Then, we draw a line from the point that has the lowest estimated cost (p_a in Figure 4) to the point that has the lowest estimated cost among those whose speedup is greater than 1 (p_b). We choose the cost at the intersecting point (p_T) of this line and the horizontal line of speedup 1.0 as our threshold value.

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

3.2 Run-time Cost Estimation

The run-time cost estimation model handles inefficient loops that cannot be handled by the compile-time cost model due to the parameters determined at run time. Because the number of instructions executed in a loop is proportional to its workload contained in it, we use the number of instructions executed (graduated) in the first invocation of a parallel loop as its cost estimation. The number of graduated instructions from the main thread is used as the estimation when the loop runs in 4P4L mode. For this model to be effective, the parallel loop must be invoked at least once in the program. If the estimated cost is smaller than our threshold value, we run the loop in 4P1L mode for the remaining invocations. The threshold value is determined heuristically and the process is similar to the compile-time cost estimation model. The remaining loops are given to the next execution-mode switching stage.

3.3 Loops for 4P4L mode and 4P8L mode

The parallel loops remaining after run-time filtering with graduated instructions may be effected heavily by the cache conflicts and interference between logical processors (threads). To identify those loops that are heavily effected by the cache conflicts between the logical processors, we use cache misses per instruction (MPI) as our selection criterion. A loop with a high MPI in 4P4L mode is most likely to have a high MPI in 4P8L mode because the effective cache size for each thread in 4P8L mode is half the size in 4P4L mode. Thus, it is better not to run this type of loop in 4P8L mode. We run each loop in 4P4L mode in its first invocation and measure the MPI. If the measured

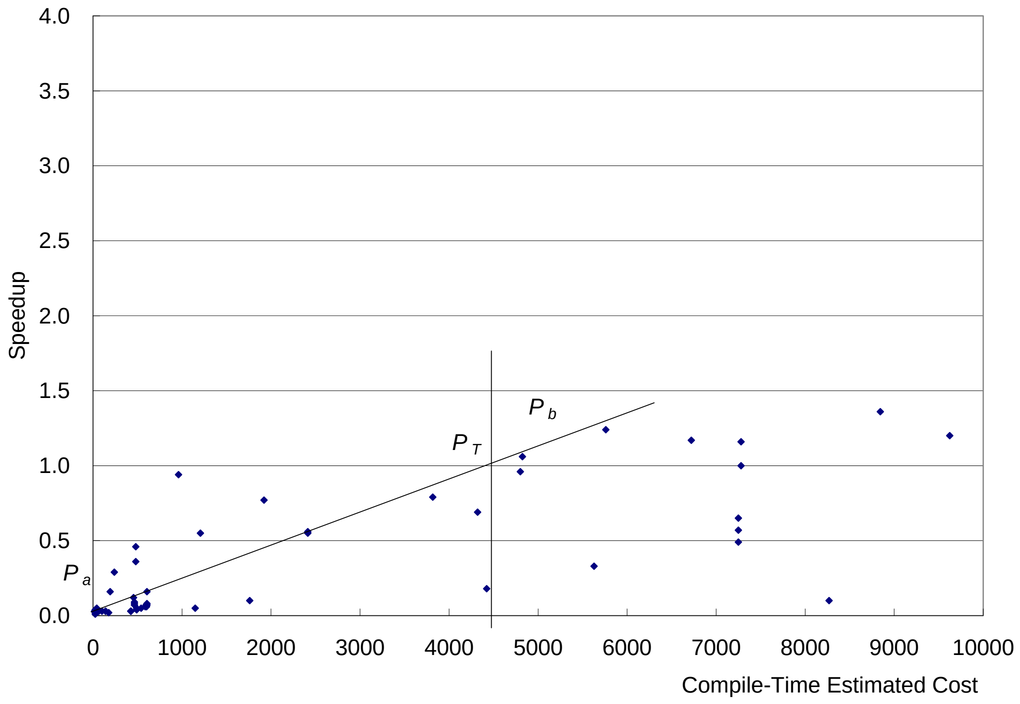

Figure 4: Determining the threshold value with the compile-time cost estimation model.

MPI is larger than our threshold value, we run the loop in 4P4L mode for the remaining invocations. Again, the threshold value is determined heuristically. We run several representative benchmark programs that contain different types of parallel loops. For each loop in the benchmark program, we measure the execution time and the number of cache misses in 4P4L mode, and the execution time in 4P8L mode. Then, we plot the speedup (i.e., the ratio of 4P4L mode to 4P8L mode) of each loop on the Y-axis and MPI on the X-axis. We choose the rightmost point (P in Figure 5) where its speedup is greater than 1.2. The MPI value at this point is used as the threshold value.

Some loops that are heuristically determined to run in 4P4L mode may perform better in 4P8L mode. To filter these loops, we use cycles per instruction (CPI). Even though CPI is partially dependent on MPI, it is a good measure of interference between instruction mixes from the two logical processors in a single physical processor. The higher the interference between logical processors, the bigger the CPI. If the CPI of a loop in 4P4L mode is smaller than our threshold CPI value, we run the loop in 4P8L in the remaining invocations. Obtaining the threshold value is similar to the process of obtaining the threshold value of MPI.

The remaining loops are given to our next adaptive execution scheme called *Most Recent with Timing.*

3.4 Most Recent with Timing (MRT)

When a parallel loop is invoked for the first time, it is executed in 4P4L mode. Its number of graduated instructions and execution time are measured. Depending on the number of graduated instructions, there are two adaptive execution schemes: 4P1L-4P4L MRT and 4P4L-4P8L MRT. If the number of graduated instructions is smaller than our threshold value, the loop is executed in 4P1L-4P4L MRT for the remaining invocations. Otherwise, it is executed in 4P4L-4P8L MRT. Determining the threshold value is similar to the run-time cost estimation model.

The MRT scheme can adapt to changes in the workload of the loop across invocations. MRT uses the recent past behavior of a loop to predict its future behavior. Consequently, if the workload of the loop changes gradually, this strategy works well. However, sudden changes may cause this strategy to work poorly. The drawback of this scheme is that the loop is executed sub-optimally at least once due to decision runs.

3.4.1 4P1L-4P4L MRT

In the case of 4P1L-4P4L MRT, the workload contained in the loop may be small, but not small enough for sequential execution. When it is invoked for the second time, it is executed sequentially (4P1L) and timed again. Then we determine whether we run this loop in 4P4L or 4P1L in the next invocation by comparing the two measurements. However, the mode in which we execute the loop for the remaining invocations is not fixed at this point. Instead, every time the loop is executed, we time it and compare the execution time to its most recent execution time in the other mode. If the latter is lower, we change its running mode.

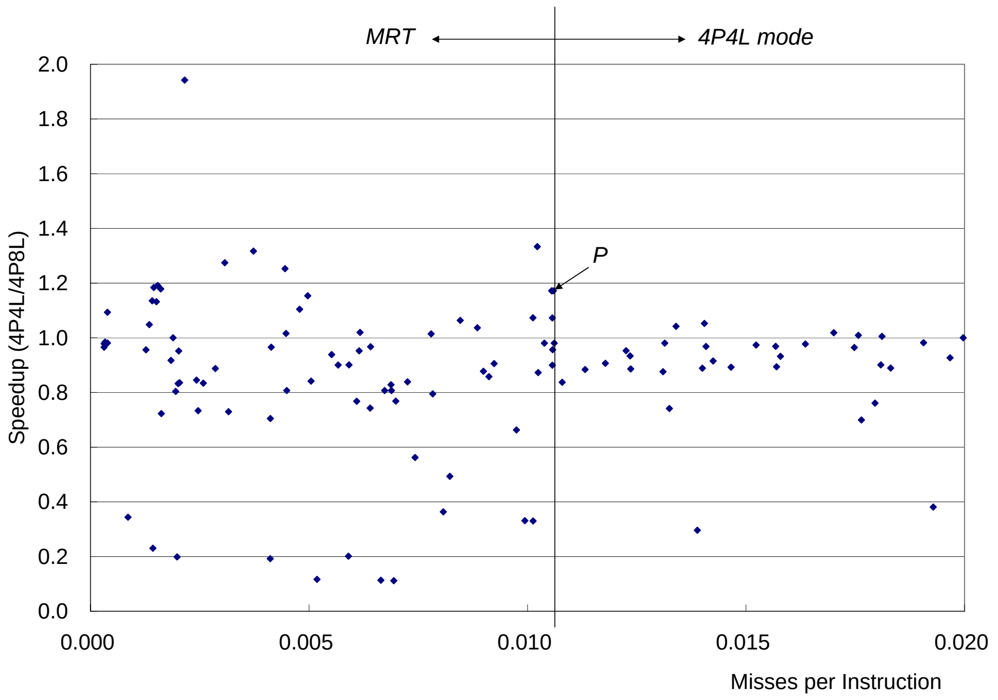

Figure 5: Determining the threshold value with the number of cache misses per instruction.

3.4.2 4P4L-4P8L MRT

In the case of 4P4L-4P8L MRT, we know that the loop contains enough workload but we do not know the degree of interference between the logical processors in 4P8L mode. Thus, when it is invoked for the second time, it is executed in 4P8L and timed again. Whether we run this loop in 4P4L or 4P8L in the next invocation is determined by comparing the two measurements. The rest is similar to 4P1L-4P4L MRT.

3.4.3 Estimating the Execution Time in 4P8L Mode

There are many factors that affect the execution time of a parallel loop. Among those, the number of graduated instructions is proportional to the synchronization instructions executed. This is due to busy waiting synchronization at the implicit barriers inserted by OpenMP for parallel loops. Consequently, the larger the number of graduated instructions, the higher the interference between the two logical processors in a physical processor due to synchronization instructions. In addition, L2 and L3 cache misses affect the execution time significantly in 4P8L mode due to the cache conflicts between logical processors. Therefore, we can estimate the execution time (T_{4P8L}) of a parallel loop in 4P8L mode with the following:

$$T_{4P8L} = a \cdot N^{grad}_{4P8L} + b \cdot N^{L2}_{4P8L} + c \cdot N^{L3}_{4P8L} + d$$

where a, b, c, and d are some constants. N^{grad}_{4P8L}, N^{L2}_{4P8L}, and N^{L3}_{4P8L} are the numbers of graduated instructions, L2 cache misses, and L3 cache misses in 4P8L mode, respectively.

Moreover, the numbers of graduated instructions, L2 cache misses, and L3 cache misses in 4P8L mode are proportional to the ones in 4P4L mode. That is,

$$N^{grad}_{4P8L} = a^{grad} \cdot N^{grad}_{4P4L} + b^{grad}$$

$$N^{L2}_{4P8L} = a^{L2} \cdot N^{L2}_{4P4L} + b^{L2}$$

$$N^{L3}_{4P8L} = a^{L3} \cdot N^{L3}_{4P4L} + b^{L3}$$

Therefore,

$$\begin{aligned} T_{4P8L} &= a \cdot a^{grad} \cdot N^{grad}_{4P4L} + b \cdot a^{L2} \cdot N^{L2}_{4P4L} \\ &\quad + c \cdot a^{L3} \cdot N^{L3}_{4P4L} + a \cdot b^{grad} + b \cdot b^{L2} + c \cdot b^{L3} d \\ &= a' \cdot N^{grad}_{4P4L} + b' \cdot N^{L2}_{4P4L} + c' \cdot N^{L3}_{4P4L} + d' \end{aligned}$$

Later, in the evaluation section, we will verify this fact using the regression analysis with experimental data and obtain the constants a', b', c', and d'. Using this model, we can estimate the execution time of a parallel loop in 4P8L mode with the numbers obtained in 4P4L mode. This scheme is an alternative to 4P4L-4P8L MRT and, consequently, will reduce the penalty caused by MRT adaptive execution scheme.

4. EVALUATION ENVIRONMENT

Compiler. We have implemented the compiler and adaptive execution algorithms described in Section 3 in our compiler preprocessor. The preprocessor is written in Perl. To identify parallel loops in a program, we used the

Table 1: Applications used.

Application	Source	Number of Lines	Data Size and Number of Iterations
BT	NAS OpenMP	3673	Class A
CG	NAS OpenMP	1507	Class A
FT	NAS OpenMP	1167	Class B
MG	NAS OpenMP	1815	Class B
SP	NAS OpenMP	3119	Class A
APPLU	SPECfp2000	3980	Test
HYDRO2D	SPECfp95	4303	Reference
MGRID	SPECfp2000	489	Reference
SWIM	SPECfp2000	435	Reference input with 200 iterations
TOMCATV	SPECfp95	201	Reference

Table 2: Machine specification.

Architecture	Symmetric multiprocessor (SMP)
Processor type (Clock speed)	Intel Xeon MP (1.5GHz)
Number of physical processors	4
Number of hardware contexts per processor	2
Total Memory	4 GB
Data caches	8KB L1 data, 512KB L2 unified, 1MB L3 unified

Polaris parallelizing compiler [2]. The parallelization information (OpenMP directives) together with the original program is fed into our compiler preprocessor. The preprocessor inserts adaptive execution code and appropriate directives in the original program to direct the compiler of the target SMT multiprocessor machine. The output adaptive program from the preprocessor is compiled by Intel Fortran Compiler 8.0 for Windows [12, 13] to generate an executable.

To obtain the values of performance counters, we implemented our own performance counter library using Microsoft Windows system calls [14, 3]. The performance counters of the main thread are used for the instrumentation because the threads created for a parallel loop are symmetric. We also used Microsoft Windows system calls to set the affinity of each thread to a logical processor.

Applications. We evaluated the effectiveness of our schemes using scientific applications written in Fortran77. We selected applications that are highly parallel. Table 1 shows the problem sizes and number of iterations used for the applications.

Target Machine. The code generated by our system is targeted to an SMP with four Intel Xeon MP Hyper-Threading processors. Table 2 shows the parameters of the SMP.

5. EVALUATION

Before we evaluate our adaptive execution strategies, we first examine the parallel loop overhead and characteristics of the parallel loops in each application (Section 5.2). We then evaluate the performance of our strategies (Section 5.4).

5.1 Parallel Loop Overhead

Table 3 shows the parallel loop overhead in each execution mode. We executed the following code 1000 times and took the averages of the execution time and the number of graduated instructions in the main thread.

```
!$OMP PARALLEL DO
      DO I=1, 4096
      ENDDO
```

We see that the number of graduated instructions in 1P2L is significantly greater than 2P2L, and the same is true for 2P4L and 4P4L. These extra instructions are mostly from the barrier synchronization at the end of the loop. This tells us that the extra instructions in 1P2L, 2P4L, and 4P8L modes most likely worsen the interference between threads because they consume the processor's internal resources. This results in more waiting time at the barrier and inefficient computation due to the interference. Thus, it is important to find a proper execution mode for a parallel loop to maximize its performance.

5.2 Characteristics of the Parallel Loops

Table 4 shows the characteristics of the parallel loops in each application. The table gives us the rationale of our adaptive execution strategies. The first section is for all the parallel loops, the second for inefficient parallel loops in each application, and the third for the parallel loops that are efficient in 4P8L mode when compared to 4P4L mode instead of 4P1L.

The first row in the first section shows the total number of parallel loops in each application and their percentage of sequential execution time relative to the entire application. The second row shows the average number of invocations for each individual parallel loop in the application. The first row in the second section shows the number of inefficient loops in each application for modes 4P4L and 4P8L. It also shows their percentage of sequential execution time relative to the entire application and their percentage of parallel execution time relative to the parallel execution time of the entire application. The

Table 3: Parallel Loop Overhead.

	1P1L	1P2L	2P2L	2P4L	4P4L	4P8L
Number of Clock Cycles	14730	16175	11785	11735	9398	116956
Number of Graduated Instructions	16416	22907	11787	13167	6632	18622

Table 4: Characteristics of parallel loops in the applications.

		BT	CG	FT	MG	SP	APP	HYD	MGR	SWM	TOM	Ave.
Parallel Loops (% sequential)		26	20	8	11	30	61	92	12	16	6	28.2
		(99.8)	(93.5)	(99.9)	(94.6)	(99.9)	(72.3)	(97.7)	(91.1)	(99.9)	(87.6)	(93.6)
Number of Invocations		79	85	14	160	188	522K	697	56K	125	128K	71K
	Mode	BT	CG	FT	MG	SP	APP	HYD	MGR	SWM	TOM	Ave.
Inefficient Loops (% sequential) (% parallel)	4P4L	2	7	1	1	2	29	24	3	4	2	7.2
		(0.1)	(0.4)	(0.1)	(2.0)	(0.1)	(21.8)	(0.3)	(0.4)	(0.1)	(0.1)	(2.5)
		(0.1)	(1.2)	(1.3)	(4.0)	(0.1)	(86.9)	(1.6)	(17.9)	(0.1)	(0.1)	(11.3)
	4P8L	2	13	1	1	2	48	41	6	5	4	12.0
		(0.1)	(0.5)	(0.1)	(2.0)	(0.1)	(22.2)	(0.5)	(4.7)	(19.6)	(30.7)	(8.0)
		(0.1)	(1.9)	(1.3)	(4.1)	(0.1)	(98.1)	(3.8)	(80.9)	(39.2)	(73.3)	(30.3)
Number of Invocations	4P4L	2	58	20	151	2	1099K	682	215K	100	750	132K
	4P8L	2	35	20	151	2	664K	599	112K	240	192K	97K
Efficient Loops in 4P8L compared to 4P4L (% in 4P4L) (% in 4P8L)		4	6	2	3	8	4	28	6	2	–	6.3
		(0.1)	(85.9)	(6.0)	(30.8)	(28.7)	(4.0)	(69.3)	(58.8)	(20.8)	–	(30.4)
		(0.1)	(82.6)	(3.9)	(26.2)	(19.8)	(0.3)	(55.1)	(14.5)	(16.1)	–	(21.9)
Number of Invocations		2	69	11	106	151	921	1007	938	101	–	331

next row shows the average number of invocations for each individual inefficient parallel loop for different modes. The last section shows the statistics for efficient loops in 4P8L mode when the speedup is calculated by the execution time in 4P4L mode divided by the execution time in 4P8L.

We see that the applications are highly parallel and that the parallel loops account for an average of 93.6% of the sequential execution time. APPLU, MGRID, and HYDRO2D contain many inefficient parallel loops both in 4P4L and 4P8L modes. In addition, many loops in CG become inefficient in 4P8L mode. These inefficient loops account for more than 10% of parallel execution time in 4P4L mode and 30% in 4P8L mode. However, the percentage of sequential execution time of these loops is at most 8%. The percentage of parallel execution time of inefficient parallel loops increases when the mode is 4P8L. This is due to the parallel loop overhead incurred by the inefficient parallel loops. Consequently, we see that inefficient parallel loops are good targets for performance optimization.

Most of the parallel loops are invoked many times (71K times on average). Inefficient parallel loops are invoked many more times than the other parallel loops (132K in 4P4L and 97K in 4P8L). Thus, adaptive execution of these loops using decision runs is feasible and effective.

About 20% of parallel loops have better performance in 4P8L mode than 4P4L mode. Thus, it is beneficial to filter these loops and run them in 4P8L mode by switching the number of threads from 4P4L mode. The techniques for this were described in Section 3.3 and 3.4.

5.3 Regression Analysis

Table 5 shows the coefficients a', b', c', and d' in Section 3.4.3 and the relationship between the numbers of graduated instructions, L2 misses, and L3 misses in 4P4L and 4P8L modes. These are obtained empirically using a regression analysis for the loops remaining after the 4P1L-4P4L MRT selection in Section 3.4.3. We see that the number of graduated instructions in 4P8L mode is directly proportional to the number in 4P4L mode. The same is true for the numbers of L2 and L3 cache misses. R^2 values that are very close to 1.0 means that our estimation of the execution time in 4P8L mode using the formulas with the numbers obtained in 4P4L mode is quite accurate.

5.4 Evaluation Results

Figure 6 shows the speedup of the applications when our techniques are used. For each application, there are seven bars. The leftmost bar (Sequential) corresponds to the base case where the applications are executed in 4P1L mode. The second (4P4L) and third (4P8L) bars correspond to the cases where the applications are executed in 4P4L and 4P8L modes respectively. The fourth bar (Static(4P4L)) and the fifth bar (Static(4P8L)) correspond to the cases where only compile-time cost estimation is applied to the applications in 4P4L and 4P8L respectively. The sixth bar (ADP) corresponds to the speedup obtained using our adaptive execution strategy described in Section 3 (steps 1,2,3,4,5, and 6 in Figure 3). Similarly, the last bar (EST) corresponds to the our adaptive execution strategy in Section 3, but it estimates the execution time in 4P8L mode with the numbers obtained in 4P4L mode instead of using MRT (steps 1,2,3,4,5, and 6' in Figure 3). The speedup for each scheme is obtained by dividing the execution time of

Table 5: The result after linear and multiple regression analyses.

Formulas	R^2
$T_{4P8L} = 0.773 \cdot N_{4P8L}^{grad} - 23.258 \cdot N_{4P8L}^{L2} + 393.006 \cdot N_{4P8L}^{L3} + 2293277$	0.9994
$N_{4P8L}^{grad} = 1.000 \cdot N_{4P4L}^{grad} - 760536$	0.9999
$N_{4P8L}^{L2} = 3.206 \cdot N_{4P4L}^{L2} - 150422$	0.9253
$N_{4P8L}^{L3} = 1.623 \cdot N_{4P4L}^{L3} - 15684$	0.8905
$T_{4P8L} = 0.773 \cdot N_{4P4L}^{grad} - 74.573 \cdot N_{4P4L}^{L2} + 637.781 \cdot N_{4P4L}^{L3} - 960399$	–

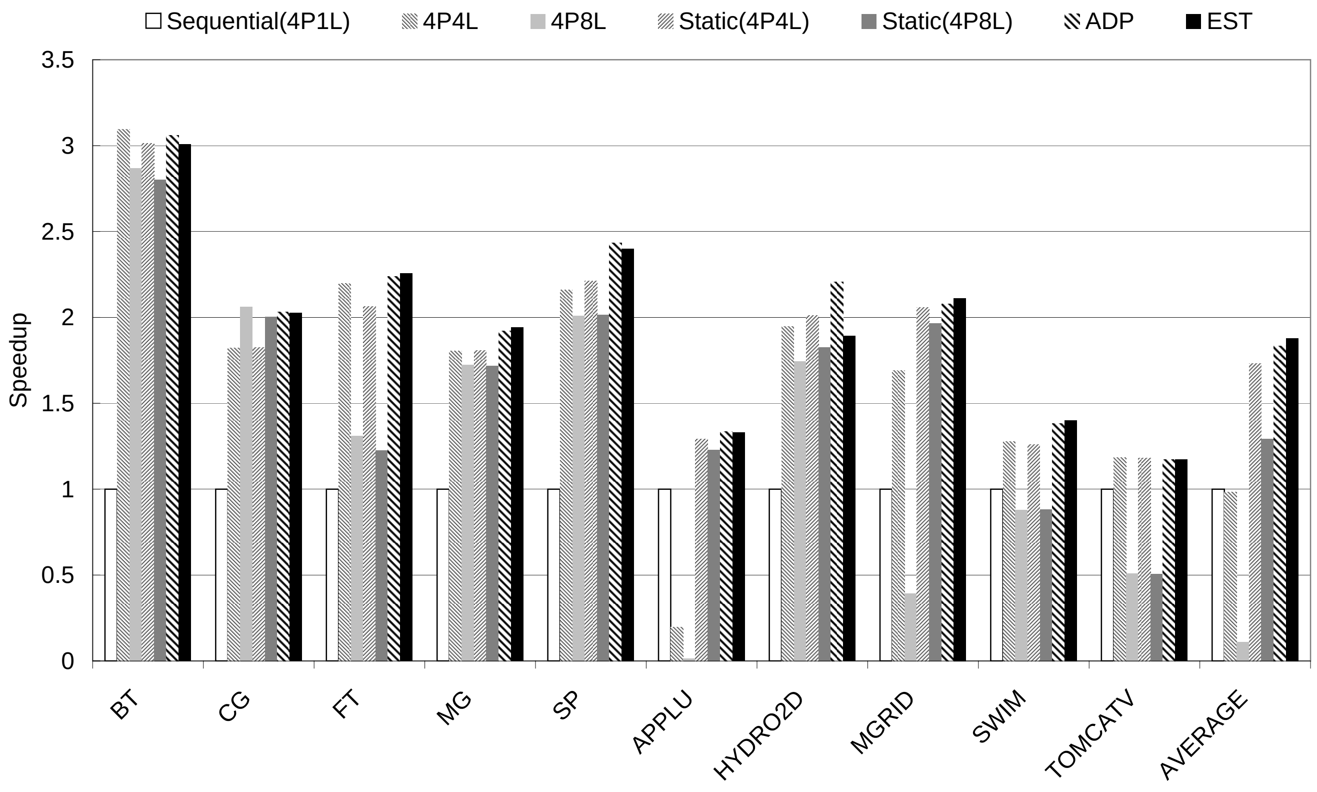

Figure 6: Speedup

Sequential by the execution time of the scheme. Thus, the taller the bar for a scheme, the better its performance.

We see that 4P4L performs much better than 4P8L. This tells us that cache conflicts and interference between threads are significant slow-down factors in an SMT multiprocessor machine. APPLU has many parallel loops that contain smaller workloads than the parallel execution overhead. Thus, its performance in 4P4L and 4P8L is very poor.

Static(4P4L) performs better than 4P4L, and Static(4P8L) performs better than 4P8L in APPLU, HYDRO2D, and MGRID. These applications contain highly inefficient parallel loops whose workload is very small. Therefore, the conventional parallel loop sequentialization technique (compile-time cost estimation) works well for these applications. However, it does not work well with the remaining applications. The performance is comparable to 4P4L and 4P8L.

ADP and EST are comparable to or better than 4P4L and 4P8L in all the applications. This implies that our adaptive execution technique is effective at finding an optimal for running the parallel loops. EST performs the best except in HYDRO2D, in which case ADP performs better than EST. This is because step 6' in Figure 3 in EST is not accurate enough for HYDRO2D. The same is true for BT and SP. Step 6' estimates the performance of a parallel loop in the 4P8L mode using a decision run in the 4P4L mode.

As mentioned in Section 1, the factors that affect program performance manifest synergistically. Each filter mentioned in Section 3 conservatively filters different inefficient parallel loops according to its own relatively simple criterion. The remaining loops are passed to the next filter. The overall effect of all the simple filters on the performance is synergistic. For example, the performance of MRT without any filters is much worse than the case with filters due to decision run overhead and some malicious workload patterns in the MRT execution scheme (we did not show the performance in Figure 6 because the performance was very poor). The filters select loops that are suitable for the MRT execution time and achieve the performance of ADP and EST.

Each instrumentation function call for reading a performance counter takes about 300 CPU cycles, and most of instrumentation calls occur in decision runs. The overhead of such calls is negligible compared to the performance improvement by our adaptive execution technique.

Overall, running parallel applications with our adaptive execution technique is, on average, about 2 and 18 times faster than the original code executed in 4P4L and 4P8L modes, respectively.

6. RELATED WORK

Many different types of adaptive optimization techniques have been proposed recently in the literature. Some approaches [10, 24, 7, 23] are based on parameterization of the code at compile time to restructure it at run time. Gupta and Bodik [10] dealt with the complexity of loop transformations, such as loop fusion, loop fission, loop interchange, and loop reversal, done at run time. Saavedra *et al.* [24] proposed an adaptive prefetching algorithm that can change the prefetching distance of individual prefetching instructions. Their adaptive algorithm uses simple performance data collected from hardware monitors at run time. Another adaptive optimization technique based on replication of objects in object-oriented programs is proposed by Rinard *et al.* [22]. To avoid synchronization overhead incurred in updating a shared object, the object is replicated adaptively. Multiple versioning of a loop for run-time optimization was first proposed by Byler *et al.* [4], and modern compilers still use this technique. Diniz et al. [8] used multiple versioning with dynamic feedback to automatically choose the best synchronization optimization policy for object-based parallel programs. Holzle *et al.* [11] proposed a dynamic type-feedback technique for improving the performance of object-oriented programs. These approaches are similar to our work in that the program dynamically adapts to the environment using run-time information. However, we neither restructure the code at run time nor deal with object-oriented programs. We focus on shared memory parallel programs with loop-level parallelism, and use different adaptation strategies with dynamic feedback to improve performance in an SMT multiprocessor architecture. In particular, our technique controls the number of active threads during execution.

Lee *et al.*[16, 17] proposed a serialization technique of small parallel loops using static performance prediction and some heuristics. They also proposed adaptive execution techniques of parallel programs for conventional SMP and distributed shared memory machines. A sophisticated static performance estimation model based on the stack distance[20] was proposed by Cascaval *et al.* [5]. A generic compiler-support framework called ADAPT was proposed by Voss and Eigenman [26, 27] for adaptive program optimization. Users can specify optimization types and heuristics in ADAPT language. The ADAPT compiler generates a complete run-time system by reading these heuristics and applying them to the target application. Voss and Eigenman [25] also proposed two run-time test schemes to identify unprofitable parallel loops. Because they used a profiling technique and the execution time of the first invocation of a parallel loop for the tests, their schemes are partially adaptive during the entire execution of an application. Even though we used a fairly simple static performance estimation model in this paper, our run-time filters compensate for the inaccuracy caused by the static model. Moreover, our scheme is fully adaptive during the entire execution of an application. Our work is also related to adaptive compilers for heterogeneous Processing-In-Memory systems [18], where heterogeneity of the system is exploited adaptively.

7. CONCLUSION

We presented performance estimation models and techniques for generating adaptive execution code for SMT multiprocessor architectures. The adaptive execution techniques determine an optimal number of threads using dynamic feedback and run-time decision runs. The adaptation strategies and performance estimation models are in the code and are inserted into the original parallel program by the compiler preprocessor. Using 10 standard numerical applications and running them with our techniques on an Intel 4-processor Hyper-Threading Xeon SMP with 8 logical processors, our code is about 2 and 18 times faster on average than the original code executed on 4 and 8 logical processors, respectively. The results indicate that our adaptive execution techniques are promising and effective at speeding up parallel programs running on SMT multiprocessor architectures.

8. REFERENCES

[1] Bowen Alpern et al. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[2] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[3] John Borozan. Microsoft Windows-Based Servers and Intel Hyper-Threading Technology. *Microsoft Corporation*, April 2002.

[4] Mark Byler, James Davies, Christopher Huson, Bruce Leasure, and Michael Wolfe. Multiple Version Loops. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 312–318, August 1987.

[5] Calin Cascaval, Luise DeRose, David A. Padua, and Daniel Reed. Compile-Time Based Performance Prediction. In *Proceedings of the 12th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 365–379, August 1999.

[6] Rohit Chandra, Leo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Manon. *Paralle Programming in OpenMP*. Morgan Kaufmann Publisher, 2001.

[7] Alan L. Cox and Robert. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th International Symposium on Computer Architectur*, pages 98–108, May 1993.

[8] Pedro Diniz and Martin Rinard. Dynamic Feedback: An Effective Technique for Adaptive Computing. In *Proceedings of the ACM SIGPLAN Conference on Program Language Design and Implementation*, pages 71–84, June 1997.

[9] Susan J. Eggers, Joel S. Elmer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, September/October 1997.

[10] Rajiv Gupta and Rastislav Bodik. Adaptive Loop Transformations for Scientific Programs. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, pages 368–375, October 1995.

[11] Urs Holzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 326–336, June 1994.

[12] Intel. *Intel Fortran Compiler User's Guide*, 2002.

[13] Intel. *Intel Fortran Programmer's Reference Manual*, 2002.

[14] Intel. *IA-32 Intel Architecture Software Developer's Manual*, 2004.

[15] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, pages 40–47, March-April 2004.

[16] Jaejin Lee. *Compilation Techniques for Explicitly Parallel Programs*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1999. Department of Computer Science Technical Report UIUCDCS-R-99-2112.

[17] Jaejin Lee and H. D. K. Moonesinghe. Adaptively Increasing Performance and Scalability of Automatically Parallelized Programs. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, July 2002.

[18] Jaejin Lee, Yan Solihin, and Josep Torrellas. Automatically Mapping Code in an Intelligent Memory Architecture. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA)*, pages 121–132, January 2001.

[19] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Kaufaty, J. Alan Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1), February 2002.

[20] R. L. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, December 1970.

[21] OpenMP Standard Board. *OpenMP Fortran Interpretations*, April 1999. Version 1.0.

[22] Martin Rinard and Pedro Diniz. Eliminating Synchronization Bottlenecks in Object Based Programs Using Adaptive Replication. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 83–92, June 1999.

[23] Theodore H. Romer, Dennis Lee, Brian N. Bershad, and Bradley Chen. Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 255–266, November 1994.

[24] Rafael H. Saavedra and Daeyeon Park. Improving the Effectiveness of Software Prefetching with Adaptive Execution. In *Proceedings of the Conference on Parallel Algorithms and Compilation Techniques*, October 1996.

[25] Michael J. Voss and Rudolf Eigenmann. Reducing Parallel Overheads through Dynamic Serialization. In *Proceedings of the International Parallel Processing Symposium*, pages 88–92, April 1999.

[26] Michael J. Voss and Rudolf Eigenmann. ADAPT: Automated De-Coupled Adaptive Program Transformation. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, page 163, August 2000.

[27] Michael J. Voss and Rudolf Eigenmann. High-level Adaptive Program Optimization with ADAPT. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 93–102, June 2001.