JavaSpMT: A Speculative Thread Pipelining Parallelization Model for Java Programs*

Iffat H. Kazi ihkazi@ece.umn.edu
David J. Lilja lilja@ece.umn.edu
Department of Electrical & Computer Engineering
Minnesota Supercomputing Institute
University of Minnesota, Minneapolis, MN 55455

Abstract

This paper presents a new approach to improve performance of Java programs by extending the superthreaded speculative execution model [14, 15] to exploit coarse-grained parallelism on a shared-memory multiprocessor system. The parallelization model, called Java Speculative MultiThreading (JavaSpMT), combines control speculation with run-time dependence checking to parallelize a wide variety of loop constructs, including do-while loops, that cannot be parallelized using standard parallelization techniques. JavaSpMT is implemented using the standard Java multithreading mechanism and the parallelization is expressed using a Java source-to-source transformation. Thus, the transformed programs are still portable to any shared-memory multiprocessor system with a Java Virtual Machine implementation that supports native threads.

1. Introduction

Java source code is compiled into the platform-independent bytecodes of the *Java Virtual Machine* (JVM) to make Java programs portable across all hardware platforms. The standard interpreted mode of Java execution is much slower than the execution of compiled languages, however. Just-In-Time (JIT) compilers and traditional compilers have been developed to improve Java performance, although these approaches sometimes restrict the portability of Java programs.

We present the *Java Speculative MultiThreading* (JavaSpMT) approach to improve Java performance. JavaSpMT executes Java programs on a shared-memory multiprocessor system using a coarse-grained speculative thread pipelining parallelization technique based on the fine-grained thread pipelining model proposed for the *superthreaded* processor architecture [14, 15]. JavaSpMT allows concurrent execution of potentially dependent loop iterations in a pipelined fashion with run-time data-dependence checking and control speculation. This model makes it possible to parallelize traditionally sequential constructs, such as *do-while* loops. Since JavaSpMT is implemented using Java's language-level multithreading, the transformed parallel Java code is still portable to any shared-memory multiprocessor system with a JVM implementation that supports native threads.

*This work was supported in part by National Science Foundation grant No. MIP-9610379 and the IBM Corporation.

2. JavaSpMT execution model

2.1. Thread pipelining execution model

The superthreaded architecture [14, 15] exploits task-level parallelism using multiple threads of control. Each thread runs on a separate thread processing unit, each with its own program counter and instruction execution data path. The execution of a program starts from its entry thread which can then fork a successor thread with or without control speculation on another thread processing unit. The forking of threads continues until all thread processing units are busy. The oldest thread in the sequential order is called the *head thread*. If a thread forks a successor thread with control speculation and later the speculation evaluates to be false, it must abort all of its successor threads.

2.2. Thread pipelining in JavaSpMT

JavaSpMT extends the basic thread pipelining model of the superthreaded processor shown in Figure 1(a) to speculatively parallelize coarse-grained Java applications on a shared-memory multiprocessor system. JavaSpMT is implemented entirely in software using the same four thread pipeline stages as in the fine-grained superthreaded

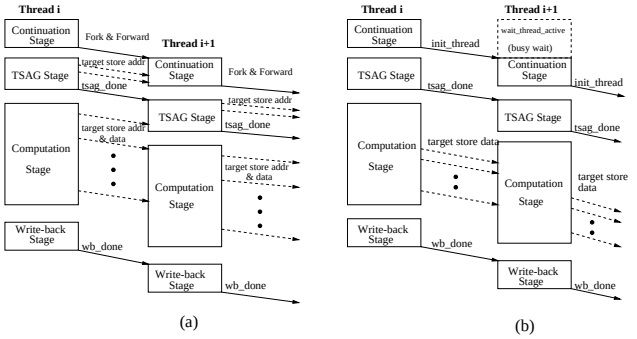

Figure 1. Pipelined execution of threads in (a) superthreaded processor, (b) JavaSpMT.

model, except concurrent threads execute on multiple processors using the standard Java multithreading mechanism. 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.

2.2.1. Continuation stage. Each thread begins its execution with the *continuation stage* to compute and forward the recurrence variables, such as loop index variables, needed to fork the next thread. This stage ends with a *fork* instruction that initiates the successor thread. In the Java implementation, all of the Java threads are created simultaneously at the beginning of the parallel loop's execution (Figure 1(b)). We introduce the global variable *thread_active* to indicate that a thread on a particular processor can proceed. Initially, *thread_active* points to the first processor so that only the first thread enters the continuation stage. When the first thread completes this stage, it updates *thread_active* so that its successor thread on the next processor can begin its continuation stage. When this thread completes its continuation stage, it will then update the variable for its successor and so on to sequentially start successive threads on different processors.

2.2.2. Target-Store-Address-Generation (TSAG) stage. The *TSAG stage* of a thread computes the addresses of the write operations, called the *target store addresses*, upon which successor threads may be data-dependent. 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 with the *wait_tsag_done* and the *release_tsag_done* instructions synchronizing on the *tsag_done* flag. This mechanism effectively implements run-time data-dependence checking. Since data values are allocated in shared-memory in the JavaSpMT model, there is no need to actually forward the data values between the threads. Instead, data-dependences among threads are enforced through flags on dependent data items. When a thread is done with a dependent data item, it releases the corresponding flag to thereby allow successive threads to proceed. The TSAG stages are synchronized using the *wait_tsag_done* and the *release_tsag_done* library calls, analogous to the original superthreaded processor instructions.

2.2.3. Computation stage. The main computation of a thread's execution is performed in the *computation stage*. If a thread executes a load operation during the computation stage whose address matches that of a target store entry in its memory buffer, 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. In JavaSpMT, dependent data items are loaded using the synchronization flags described in the TSAG stage. A speculative thread performs the writes of values that are not needed by subsequent threads in local memory. It later updates the shared memory during the write-back stage to maintain correct memory state. Non-speculative threads directly update shared memory to eliminate the write-back overhead.

2.2.4. Write-back stage. A thread that was initiated speculatively may have to be aborted during its computation stage if the speculation turns out to be incorrect. If a thread completes normally without being aborted by a predecessor thread, it will end with a *stop* instruction. The thread waits until it becomes the head thread and then performs its *write-back stage*. If a thread determines that the control speculation is incorrect, however, it aborts all successor threads by setting the *ABORT_FLAG*. Each thread checks this 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 which has already started its computation stage, but which needs to be aborted because a predecessor thread has set the flag, will either 1) also execute the instruction that causes the abort and thereby exit the computation stage, or 2) will complete its computation stage and will then check for the abort signal before it begins its write-back stage. Since data updates for speculative threads are performed in local memory, a thread which is supposed to be aborted does not produce incorrect program state even if it completes its computation stage.

2.3. Handling Java language features

The following discusses how Java's unique language features are handled by JavaSpMT.

2.3.1. Exception handling. Exceptions that may be thrown during the parallel execution of the JavaSpMT code are handled by catching the exception in the corresponding Java thread's *run* method [1]. This caught exception may be thrown back to the main thread of execution if necessary. However, if multiple exceptions occur in different threads of the same loop, only one such exception will be thrown back to the main thread.

2.3.2. 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.

2.3.3. Garbage Collection. Java's automatic garbage collection does not affect the semantics of JavaSpMT parallel codes. It may affect JavaSpMT performance, however. 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. Performance evaluation

We used a 196 MHz IP25 SGI Challenge shared-memory multiprocessor system to evaluate the performance of the JavaSpMT technique. The system consists of 8 MIPS R10000 processors with 32 Kbytes of separate data and instruction caches, 2 Mbytes of secondary unified cache, and 768 Mbytes of 2-way interleaved shared memory. The system uses the IRIX 6.2 implementation of JDK 1.1.6 as its JVM. The Java class files were executed using JIT compilation. Furthermore, the Java threads were mapped onto physical processors using the *-native* flag.

3.1. Parallelization overhead

The implementation of the speculative thread pipelining model in Java requires the use of some additional library classes. The methods in these classes are used to initiate JavaSpMT threads and to enforce the pipelined execution of concurrent threads. This parallelization overhead has two components. First, native Java threads must be created on the processors to execute each of the parallel threads of the JavaSpMT thread pipelining model. Since JavaSpMT relies on the Java thread package for thread creation, this *thread creation overhead* is a function of the specific Java thread package. The thread pipelining stages are implemented through a number of library calls. Each thread must execute these library calls once as it goes through the different pipeline stages. The effect of the *library call overhead* is somewhat minimized due to the pipelined execution of threads.

Table 1. Parallelization overhead of the JavaSpMT model on the test system.

	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 1 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.

3.2. Application-level performance

3.2.1. Test programs. The performance of JavaSpMT was evaluated using three Java application programs. The first is the *Gaussian Elimination* program (*Gauss*) for solving an NxN system of linear equations. The second one is a Java version of the *MDG* program from the Perfect Club Benchmarks [5], which performs molecular dynamic simulation of flexible water molecules. The other is a Java version of the Unix utility program *wc* that counts lines, words, and characters in the input files.

The *forward elimination* loop of *Gauss* is a three-level loop nest. While the outermost loop is sequential, the two innermost loops can be fully parallelized. *Loop 1000* of the *INTERF* routine in *MDG* is a two-level loop that has a loop-carried dependence due to reduction operations on

shared data variables. We parallelized the outer loop of *INTERF* since it has a much larger granularity. The main computation of *wc* consists of a *while* loop that reads and processes a block of bytes from the input file. 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 our JavaSpMT model allowed this *while* loop to be easily parallelized, however.

3.2.2. Speedup results. Each of the three serial Java application programs was manually transformed to the corresponding parallel source code. The resulting speedup for each program was computed by using as the basis the execution time of the original serial version of the program when executed on a single processor of the same system.

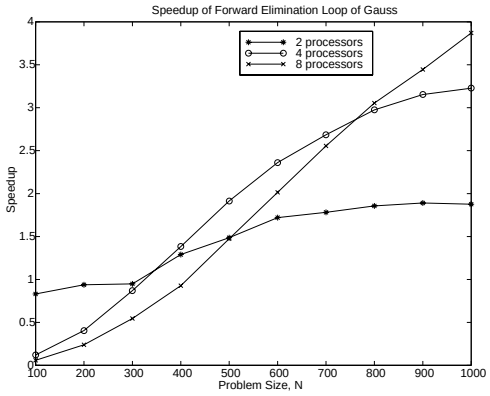

Figure 2. Speedup of *Gauss*.

The problem size for *Gauss* was varied from 100 to 1000 with an increment of 100. To obtain an appropriate grain size, 8 iterations were combined in each thread. The speedup of the *forward elimination* loop is shown in Figure 2. The speedup 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.

Due to the increasing parallelization 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.

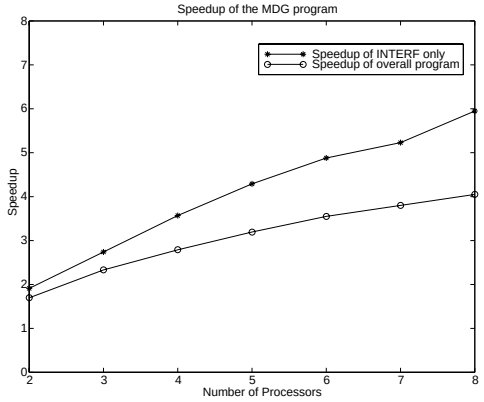

Figure 3. Speedup of *MDG*.

Figure 3 shows the speedup of the parallelized *INTERF* method and the overall speedup of the entire *MDG* program for 2 to 8 processors. Eight iterations per thread were again used to provide an adequate amount of work per thread. The speedups scale nearly linearly with the number of processors. Due to the relatively large granularity of the loop iterations, the speedups are quite high even though the loop has some cross-iteration dependences. Since about 90% of the program is parallelizable, the overall speedup also is quite high.

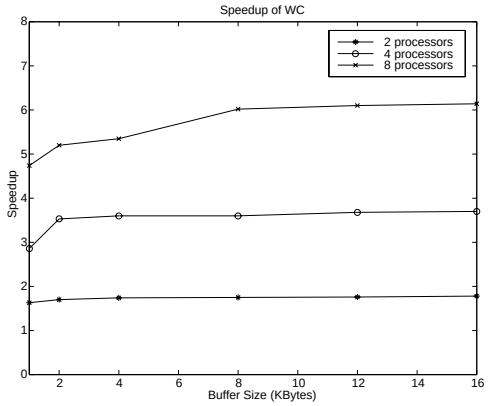

Figure 4. Speedup of *wc*.

The *wc* program was executed while varying the input buffer size using an input file of 2 Mbytes. The buffer size determines the number of bytes read from the input file for each iteration of the *while* loop. The buffer size was varied from 1 to 16 Kbytes. Each thread was assigned one iteration of the parallel loop. The speedup of the *wc* method of the program with 2, 4, and 8 processors is shown in Figure 4. Each iteration of the while loop had sufficient computation to produce significant speedups for all three processor configurations. The speedup of this program increases as the buffer size is increased since increasing the buffer size increases the granularity of each thread, until the performance saturates.

3.2.3. Cost of misspeculation. When threads are speculated incorrectly a penalty to abort and restart the threads

will be incurred that adds to the overall execution time. 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. The cost of misspeculation is determined by the work done within each thread and the total number of threads initiated. The misspeculation overhead might be reduced by forcing an exception when an abort condition is encountered, although this mechanism has not been incorporated into our current implementation.

4. Comparison to related work

C language implementation of the speculative multithreading model. We have previously developed a version of the speculative multithreading parallelization model for applications written in the C language [9]. We found that the performance of the JavaSpMT model, as measured by the speedups, is somewhat lower than that of the C implementation, especially with higher numbers of processors. As an example, Figure 5 shows the speedup of the *INTERF* routine of MDG using both C and Java implementations of the speculative multithreading model. These results indicate that, for the same problem size, the speedup of the Java implementation is lower than the C implementation. Furthermore, this difference increases as the number of processors is increased.

The performance degradation of the Java implementation is due to its relatively high parallelization overhead compared to the overhead of the C implementation. The higher overhead is primarily due to the underlying implementation of the JVM. In particular, 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

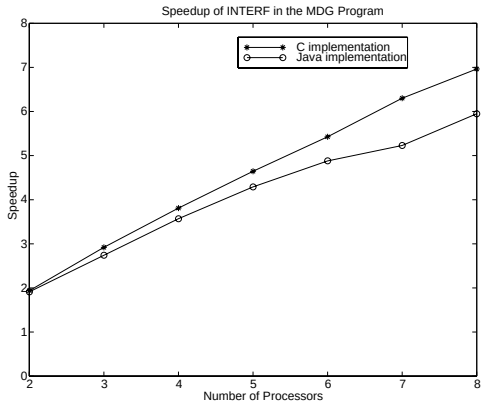

Figure 5. Comparison between the C and the Java implementations of the speculative multithreading parallelization model.

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.

JAVAR parallelization tool. JAVAR [1] parallelizes serial 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 a matrix multiplication program. Each parallel thread computed the inner products of a strip of the result matrix. Figure 6 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 our 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.

Other parallelization tools. Among the different existing Java parallelization techniques [1, 2, 3, 6, 7, 8, 10, 11], High Performance Java [1], Do! [10], and the Tiny Data-Parallel language [6] target parallelization on shared-memory architectures. The other approaches [2, 3, 7, 8, 11] attempt to improve performance of Java programs in a distributed en-

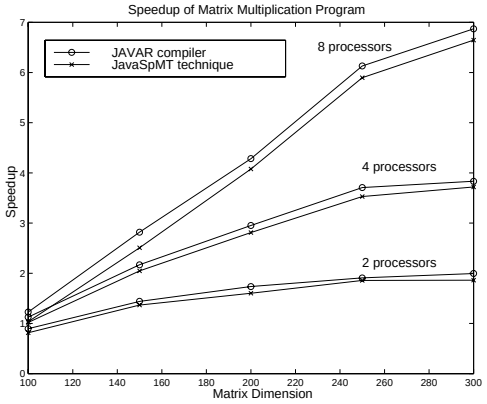

Figure 6. Comparison between the JAVAR and the JavaSpMT execution techniques.

vironment. JavaSpMT targets shared-memory architectures and is similar to the JAVAR tool [1] in that the parallelization is achieved by a Java source code transformation using Java multithreading and a library that is implemented entirely in Java. However, the speculative execution combined with run-time data-dependence checking make our approach unique. Our parallelization model can parallelize a wide variety of loop structures, including *do-while* loops and loops with cross-iteration dependences that cannot be resolved statically.

Although there are no other existing Java parallelization techniques that support run-time data-dependence checking or speculative execution, a number of *inspector-executor* schemes [4, 12, 13, 16, 17] have been developed for parallelization of compiled languages, such as Fortran and C. These techniques exploit medium to coarse-grained loop-level parallelism in programs in which the parallelism cannot be detected at compile-time.

5. Conclusion

We have extended the superthreaded speculative parallelization technique [14, 15] to Java programs to exploit coarse-grained parallelism on shared-memory multiprocessor systems. Programs parallelized using JavaSpMT remain portable across any shared-memory multiprocessor system that has a JVM implementation supporting native threads. Comparing the performance to the JAVAR tool, which uses standard parallelization techniques to parallelize loops, showed that our JavaSpMT model incurs slightly more overhead. This additional overhead is due to the thread library calls needed to implement pipelined execution. However, a variety of additional program constructs, such as *do-while* loops, can be parallelized with JavaSpMT. Thus, this extra overhead is not significant when considering the performance improvement that was obtained by the parallel execution of loops that would not otherwise be parallelizable by standard techniques.

References

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

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

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

[4] 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.

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

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

[7] 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/.

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

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

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

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

[12] 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.

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

[14] J.Y. Tsai and P.C. Yew, *The Superthreaded Architecture: Thread Pipelining with Run-time Data Dependence Checking and Control Speculation*, Conference on Parallel Architectures and Compilation Techniques, Oct. 1996, pp. 35-46.

[15] J. Y. Tsai et al., *The Superthreaded Processor Architecture*, IEEE Transactions on Computers, Special Issue on Multithreaded Architectures and Systems, Sept. 1999, pp. 881-902.

[16] 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.

[17] 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.