The Pensieve Project: A Compiler Infrastructure for Memory Models

Chi-Leung Wong [*] Zehra Sura [*] Xing Fang [†] Samuel P. Midkiff [‡] Jaejin Lee [†] David Padua [*]

[*] *University of Illinois*
Urbana, IL 61801
{cwong1,zsura,padua}@cs.uiuc.edu

[†] *Michigan State University*
East Lansing, MI 48824
{fangxing,jlee}@cse.msu.edu

[‡] *IBM T.J. Watson Research Center*
Yorktown Heights, NY 10598
smidkiff@us.ibm.com

Abstract

The design of memory consistency models for both hardware and software is a difficult task. It is particularly difficult for a programming language because the target audience is much wider than the target audience for a machine language, making usability a more important criteria. Adding to this problem is the fact that the programming language community has little experience designing programming language consistency models, and therefore each new attempt is very much a voyage into uncharted territory. A concrete example of the difficulties of the task is the current Java Memory Model. Although designed to be easy to use by Java programmers, it is poorly understood and at least one common idiom (the "double check idiom") to exploit the model is unsafe. In this paper, we describe the design of an optimizing Java compiler that will accept, as either input or as an interface implementation, a consistency model for the code to be compiled. The compiler will use escape analysis, Shasha and Snir's delay set analysis, and our CSSA program representation to normalize the effects of different consistency models on optimizations and analysis. When completed, the compiler will serve as a testbed to prototype new memory models, and to measure the differences of different memory models on program performance.

1. Introduction

Memory consistency model design, both for hardware systems and programming languages, has long been a difficult and contentious topic [1]. The difficulty of reaching a consensus on an ideal memory model is exacerbated by the fact that the quality of a model depends both on its ease of use (i.e. how hard it is to write correct programs using the model) and the performance the model can deliver to programs.

In the area of software memory models, even comparisons of ease of use and performance are difficult. This is because widely available implementations of different consistency models are targeted to a specific domain (e.g. release consistency is used in the Treadmarks system for technical computing [2], and the Java memory model specified for the Java language is used mostly for non-numerical computing [6]). Also, compiler optimizations do not presently attempt inter-thread analysis, so any performance numbers gathered may not be representative of the performance that the consistency model could obtain in that application if a more aggressive compiler were used.

Into this information vacuum steps the widespread availability and adoption of multithreaded programming via Java. Java is the first widely used programming language to define a multithreaded consistency model as an integral part of the language[1]. Java multithreading is used for interactive event driven applications, multiprocessor hardware, and networked applications that cannot freeze when waiting for a network event. Because of such widespread influence, it is important to analyze the Java Memory Model (JMM) [6]. This model has tried to provide both ease of use and performance using current compiler technology. The result has been a memory consistency model with many problems, both in usability and implementation [10]. It is our belief that attempting to design usable, efficient consistency models within the current information vacuum is difficult, if not impossible. Moreover, given the subtlety of the trade-offs involved, producing a satisfactory consistency model without extensive prototyping and testing of various versions will not only be extremely difficult, but runs the risk of alienating the larger programming community before a satisfactory model is deployed. Finally, there is no reason to believe that "one size fits all", i.e. that there is one best consistency model for all application domains.

This paper describes a flexible compilation system we are developing that will allow different consistency models to be easily prototyped, and their effect on performance measured both with and without advanced inter-thread anal-

[1] Other languages, including Ada, have done this, but they have never achieved the programmer base of Java. This is of great practical importance, because as the user base for a language expands, usability becomes more important.

Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'02)
1087-4089/02 $17.00 © 2002 IEEE

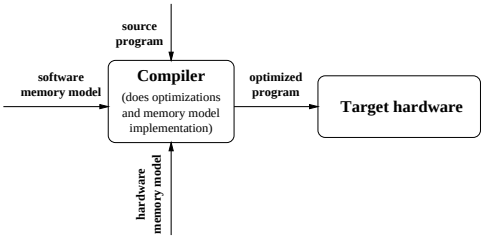

Figure 1. The compiler

ysis algorithms. This approach will make the testing of new consistency models easier. It will allow the performance benefits of consistency models to be measured relative to one another, and will enlarge the design space of practical consistency models by letting optimizations obtain acceptable performance from more usable models. In short, our system harnesses compiler technology to bring performance and ease of use to memory models. This is analogous to the use of advanced compiler technology for realizing the software engineering benefits of object oriented programming and data abstraction.

Existing Java programs are to be used as the basis for building a benchmark suite because well-synchronized Java programs yield sequentially consistent execution outcomes, and the synchronization required is similar to that for other popular consistency models.

The outline of the rest of the paper is as follows. Our approach is discussed in detail in Section 2. Some of the analyses and optimizations required to support programmable memory models are described in Section 3. Some examples of our approach are illustrated in Section 4. The current implementation is discussed in Section 5, and Section 6 presents the conclusion.

2. Our approach

It is hard to design memory consistency models as is evident from the problems of the Java Memory Model [10]. Great care and thought must be exercised when considering the effects of different memory consistency models because of the difficulty in understanding them. This is true even for people with much experience in parallel computing [7]. Although the discussion in [7] refers to hardware models, the same caveats apply even more to consistency models for languages because of the larger user community, and the greater variation in skill levels of practitioners.

The compiler infrastructure that we are developing contains analysis and optimization techniques for explicitly parallel programs, such as programs written in Java. The compiler presents a program level memory consistency for explicitly parallel programs to programmers by hiding the memory consistency model of the underlying machine architecture. It optimizes programs based on the semantics of the software and hardware consistency models that it accepts as inputs (Figure 1). The characteristics of the compiler are:

- it provides programmers with a programmable memory model view of the underlying architecture: it bridges the gap between the model expected by the programmer and that provided by the hardware.
- it makes it possible to apply optimization techniques to parallel programs by following the memory consistency model specified: these optimizations are those that are not handled correctly by conventional compilers because they cannot account for restrictions on memory accesses due to the memory consistency model.

Benefits of programmable memory models

There are at least four major benefits of programmable memory models. The first is to allow rapid prototyping of new memory models within a compiler that is capable of applying aggressive optimizations to programs written with that memory model. This will allow language designers to make fair comparisons among different memory models, since the compiler infrastructure is the same. It will also allow performance comparison between different memory models with and without inter-thread analysis. Measuring performance is important when evaluating memory models because the deviations from sequential consistency in all memory models known to the authors exist to improve performance. Also, measuring performance with and without inter-thread analysis and optimizations is important because it allows the value and cost of inter-thread analysis to be determined. The latter is particularly needed to determine the value of inter-thread analysis in dynamic compilation systems.

Second, optimizers can be used by different programming languages. The optimizations and analyses are not limited to a single programming language or family of languages as the language details can be abstracted away, while the ordering required by a memory model is maintained. To our knowledge, the current situation relies on stopgap measures.

Third, the user and vendor communities can be encouraged to try and accept new, improved memory models. In a system such as the one we propose, changing a language memory model from one to another can simply be an option change on the front-end. The optimization and code-generation phases take care of the change and the users do not need to modify the original source program.

Fourth, our system supports inter-operability across programs and programming languages that have different

Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'02)
1087-4089/02 $17.00 © 2002 IEEE

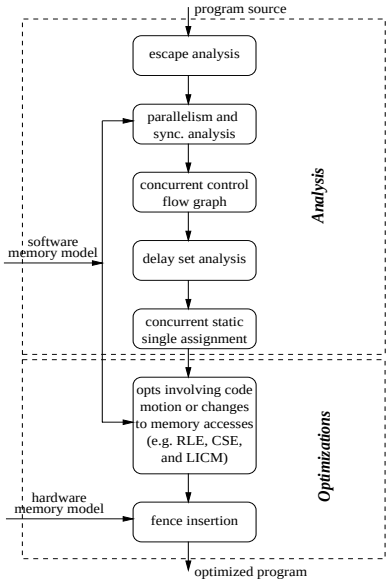

Figure 2. Compilation stages for memory model implementation

memory models. This is because we use delay set analysis [11] to determine thread local information about constraints on the ordering of memory accesses in the thread.

Programmable memory models allow the user to specify the memory model. Since Java employs dynamic compilation systems, the choice of memory model can be made during program execution. For example, when the data size is small, the user can choose to do a single-threaded computation using a weak memory model. On the other hand, when the data size is large, the user can choose a multi-threaded computation using a stronger memory model (e.g. sequential consistency model).

3. Analyses and optimizations

In order to support programmable memory models, our system uses appropriate program analyses to extract information about programs. This information is then used to perform correct optimizations. Figure 2 shows the sequence of compilation stages that are relevant.

Our system uses a *concurrent control flow graph*(CCFG) to represent the program to be analyzed, from which the *concurrent static single assignment*(CSSA) graph [9] is constructed. The concurrent control flow graph extends conventional control flow graphs with components that enable the representation of parallel programs. There are constructs to denote the creation and termination of new threads of execution, shared memory variable accesses, and the effect of synchronization operations.

The CSSA graph is an extension of the traditional SSA graph for explicitly parallel programs. It preserves the property that all uses of a variable are reached by exactly one static assignment to the variable. Multiple definitions that may reach the use of a variable are combined using a CSSA function. These functions are placed at confluence points in the program CCFG, which may be one of the following:

1. points with multiple incoming control flow edges (these are the same as for traditional SSA graphs).

2. points at the end of thread bodies, where the effect of inter-leaving of assignments to shared variables can be summarized.

3. points where shared variables are used, so that assignments that may have occurred in other threads can be accounted for.

Escape analysis [4] is used to determine the set of objects shared by different threads. It helps to make the analysis more precise and generate more accurate CCFG and CSSA graphs. Using these two intermediate graph forms, the statements that might execute in parallel can be determined (*may happen in parallel analysis*) and *delay set analysis* [11] can be performed. Delay set analysis finds the minimum set of constraints that are required to preserve the inter-thread properties demanded by the memory model. Each constraint (or *delay*) specifies the execution order that must be enforced for a pair of accesses to shared memory locations within a program segment. Delays are obtained by analyzing *conflict cycles* in the control flow graph of all the threads in the program. A conflict cycle is one that includes at least one *conflict* edge, i.e. an edge between two accesses in different threads that are to the same memory location and at least one is a write access.

Program transformations that involve code motion or elimination of memory accesses are sensitive to the memory model of the program. These include redundant load elimination (RLE), common sub-expression elimination (CSE), loop invariant code motion (LICM), and fence insertion. A *fence* is a synchronization instruction that imposes an ordering between memory operations that precede it and those that follow it. Due to out-of-order execution architectures, fences must be inserted such that they enforce all the delays determined by delay set analysis. Optimization of fence insertion is discussed in [8].

Dynamic class loading

Due to Java's dynamic class loading feature, it is possible that a method being analyzed contains calls to other unresolved methods. This could result in incomplete information for analysis and the compiler must make conservative assumptions. For example, when an object is passed to an unresolved method, it has to be considered as escaping even though the method might not make the object accessible to

Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'02)
1087-4089/02 $17.00 © 2002 IEEE

```
Complex compute() {
     ...
     Complex c = new Complex(1,2);
     Complex d = new Complex(3,4);
     Complex e = c.multiply(d);
     /* c and d are not used hereafter */
     ...
}

Complex multiply(Complex x) {
     double r = this.re * x.re - this.im * x.im;
     double i = this.re * x.im + this.im * x.re;
     return new Complex(r,i);
}
```

Figure 3. Thread local objects example

```
Complex compute() {
     ...
     Complex c = new Complex(1,2);
     Complex d = new Complex(3,4);
     Complex e = c.multiply(d);
     /* c and d are not used hereafter */
     ...
}

Complex multiply(Complex x) {
     double tre = this.re;
     double tim = this.im;
     double xre = x.re;
     double xim = x.im;
     double r = tre * xre - tim * xim;
     double i = tre * xim + tim * xre;
     return new Complex(r,i);
}
```

Figure 4. Optimized program for thread local objects example

another thread.

There are two ways to tackle the analysis imprecision from dynamic class loading:

1. ***pre-loading classes***: All the classes to be used in the application can be pre-loaded so that there are no unresolved methods when doing analysis during compilation. However, this option is available only when all classes that might be loaded during program execution can be statically determined. This precludes programs that use the Java `forName` method to dynamically choose a class to use.

2. ***re-analyzing methods***: Methods can be re-analyzed and more precise results can be obtained. This requires an adaptive recompilation system [3] that allows methods to be recompiled during execution of the program. To avoid expensive re-analysis overhead, only execution hotspots are recompiled. These hotspots typically include code sections with many fences because they are very expensive. The cost model for recompiling must take into account both the number of fences in the hotspot, and the performance improvement to be gained by removing some fraction of them.

4. Example programs

Some example programs to illustrate the algorithms used in our compiler infrastructure are presented in this section.

4.1. Thread local objects

Figure 3 shows a program fragment that multiplies two complex numbers. The `compute` method creates two complex objects referenced by `c` and `d`. These two objects are used only in the statement `Complex e = c.multiply(d)` and are never accessed outside the `compute` method. In the `multiply` method, the values of `this` and `x` are accessible by only one thread initially and never passed to another thread or assigned to static variables. Therefore, the variables `this.re`, `this.im`, `x.re`, and `x.im` are thread local. In fact, this information can be derived automatically by a context-sensitive, flow-sensitive escape analysis [4].

Classical optimizations can be applied to the body of the `multiply` method even if the `compute` method (and hence the `multiply` method) is called in a multithreaded program. This will not violate the correctness for any memory model because the variables are not accessed by multiple threads. Thus, the redundant loads inside `multiply` can be eliminated. The optimized program is shown in Figure 4. In the optimized program, 4 loads are performed instead of 8 loads when computing the value of `r` and `i`.

4.2. Interthread analysis

Figure 5(a) shows an example program that uses multiple threads to create a binary tree data structure. The main thread initializes the root of the tree, and then spawns two new threads, `ThreadLeft` and `ThreadRight`. The root of the tree is passed to these new threads in thread construction, and they use it to allocate and initialize nodes at level one of the tree. Thus, `parent1` and `parent2` point to the object referred to by `root` in the main thread. Also, note that the `temp` variables are thread local.

For this example, sequential consistency is the memory model used. It is assumed that the classes for `ThreadLeft` and `ThreadRight` have already been loaded when the code for the main thread is being analyzed. Since the object referenced by `root` is created in the main thread and passed to `ThreadLeft` and `ThreadRight`, escape analysis marks it as escaping from the main thread. Also, the objects referenced by `parent1` and `parent2` are passed to `ThreadLeft` and `ThreadRight` from the `Main` thread when the threads are created, and they are marked as shared variables. `parent1` and `parent2` do not further escape from `ThreadLeft` and `ThreadRight` respectively. Thus, the inter-thread analysis can determine that `parent1` and `parent2` are shared between instances of the main thread, `ThreadLeft`, and `ThreadRight`, and no other threads can access them. There is no conflict between any access in `ThreadLeft` and `ThreadRight`, which are the two threads that run concurrently. This is because there is only one shared variable (corresponding to `root` in the main thread), and nei-

Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'02)
1087-4089/02 $17.00 © 2002 IEEE

```
Main Thread

P1:  root = new TreeNode();
P2:  root.depth = 0;
P3:  root.value = 1;
P4:  t1 = new ThreadLeft(root);
P5:  t2 = new ThreadRight(root);
P6:  t1.start();
P7:  t2.start();
P8:  t1.join();
P9:  t2.join();

ThreadLeft(TreeNode parent1)          ThreadRight(TreeNode parent2)

T1:  n = new TreeNode();              S1:  n = new TreeNode();
T2:  temp = parent1.depth;            S2:  temp = parent2.depth;
T3:  n.depth = temp + 1;              S3:  n.depth = temp + 1;
T4:  if (parent1.depth == 0)          S4:  if (parent2.depth == 0)
T5:    n.firstlevel = true;           S5:    n.firstlevel = true;
T6:  parent1.left = n;                S6:  parent2.right = n;
```

Figure 5. Program that needs inter-thread analysis for optimization

ther `ThreadLeft` nor `ThreadRight` write to a commonly accessed field of this shared variable. Thus, the values of `parent1.depth` at T2 and `parent2.depth` at S2 can be kept in registers and reused at T4 and S4, instead of loading them again from memory.

However, if the example is modified so that the classes `ThreadLeft` and `ThreadRight` are as shown in Figure 6(a), then this optimization is not possible. Now, there is a write to `parent2.depth` at S2 in `ThreadRight`. This is the same shared variable accessed by `parent1.depth` at T2 and T4 in `ThreadLeft`. So, there are conflicts between T2 and S2, and T4 and S2. Also, now `ThreadLeft` and `ThreadRight` synchronize using `parent1.depthSet` (`parent2.depthSet`) at statements T3 and S4. This introduces the conflict edge between T3 and S4. Thus, we have a cycle T3, T4, S2, S4, T3 in the graph. This induces the delay edges from T3 to T4, and from S2 to S4.

Applying RLE to `parent1.depth` at T4 effectively reorders T3 and T4, and this violates the delay edge from T3 to T4. Semantically, RLE reuses the value loaded at T2, and this is not necessarily the value updated by S2. For correct execution of the program, T4 should necessarily read the value that was updated by S2. Figure 6(b) shows the control flow graph and the *conflict* and *delay* edges.

In the example, the delay edges must be enforced by inserting fence instructions between T3 and T4, and between S2 and S4 when generating machine code.

4.3. Dynamic class loading

Figure 7 shows another program fragment that performs computation on complex numbers. In this example, two complex numbers are added by calling `Adder.add(c,d)`. When `Main.main` is analyzed, the `Adder` class has not been loaded yet. Therefore, the analysis has to be conservative for `Add.add(c,d)` because it

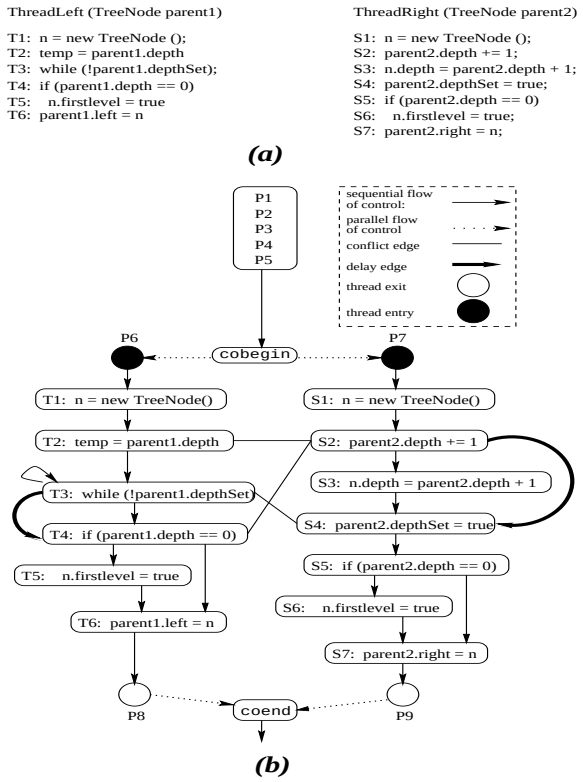

Figure 6. Program to illustrate inter-thread conflicts and delay set analysis

does not know the method body. In this case, it has to be assumed that the objects passed to the method escape to other threads even though this is not the case. This makes the redundant load elimination of `S3` invalid because it cannot be disproven that some other thread could change the value of `c.re` and `d.re` after the execution of `S1`. This problem can be alleviated by re-analyzing `Main.f()` when the `Adder` class has been loaded. At that point, it can be determined that `c` and `d` do not escape from `S1` and `S2` and redundant load elimination can be performed.

5. Current implementation status

Our goal is to implement the analysis, optimization and code generation techniques described previously. The Jikes RVM[2] [5] is used to implement our system. This is a Java virtual machine written mostly in Java. It does JIT compilation for the whole program, i.e. before any method gets executed, machine code is generated. This feature is useful for insertion of fences to enforce the memory model consistency. The RVM also supports an adaptive recompilation system that is used to handle dynamic class loading.

[2]Originally called Jalapeño

Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'02)
1087-4089/02 $17.00 © 2002 **IEEE**

```
class Main {
     public void main(String[] args) {
        Complex c = new Complex(1,2);
        Complex d = new Complex(3,4);
        Adder.add(c,d); /* c and d have "escaped" */
S1:     int r1 = c.re;
S2:     int r2 = d.re;
S3:     int r3 = c.re;
        ...
     }
}

class Adder {
     static void add(Complex x, Complex y) {
        x.re += y.re;
        x.im += y.im;
     }
}
```

Figure 7. Example to illustrate the effect of dynamic class loading

Our implementation performs conservative analysis, optimization and code generation, which gives a lower bound for program execution time when memory model constraints are enforced. The memory model to be used for program compilation is an input parameter in our design, but this information is hardwired into the system at present. Sequential consistency and the Java memory model are currently supported.

Escape analysis is performed before doing any optimizations or transforming the high level representation of the source to the intermediate form. The escape information is used to augment existing optimizations like redundant load/store elimination, scalar replacement of loads, and loop invariant code motion. This is done to avoid performing optimizations for memory accesses to shared variables.

In the Jikes RVM, the optimization phases are structured such that redundant load/store elimination is done earlier on. Thereafter, the number of accesses to memory variables are fixed in the code. Further optimizations work on the results of these load/store instructions and temporary variables. When tailoring these optimization phases to account for a particular memory model, the effects of code motion on the relative ordering of these load/store instructions have to be accounted for. However, elimination of loads and stores is not a cause for concern as only temporary variables can be eliminated at this stage. For example, CSE as implemented in Jikes at this time, eliminates common subexpressions involving only temporary variables that cannot be changed by another thread. So CSE does not need to be modified to account for different memory models.

6. Conclusions

We have outlined the design of a compiler that will compile programs in which the memory model is a characteristic of the program and not the programming language. Such a compiler is desirable because it allows some key questions in the compilation of explicitly parallel languages to be answered, i.e.: (i) how useful is inter-thread analysis for the optimization of different memory consistency models, (ii) how expensive, in compiler execution time and space, are easy-to-use memory models than more performance-oriented models, and (iii) how expensive, in application execution time and space, are the simpler memory models. Moreover, the compiler will serve as a useful tool to prototype and test new memory consistency models.

We are constructing a compiler to implement these ideas. This compiler builds on the Jikes RVM [5] compiler. The compiler will implement delay set analysis, the algorithms described in this paper (and [9]), as well as new data-flow optimization and analysis algorithms we are developing. Algorithms to map from the language memory model to a storage memory model have been developed [8] and are being implemented.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, December 1996.

[2] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, pages 18–28, Feb 1996.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Oct. 2000.

[4] J.-D. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings ACM 1999 Conference on Object-Oriented Programming Systems (OOPSLA '99)*, pages 1–19, Nov. 1999.

[5] B. A. et al. The Jalapeño virtual machine. *The IBM Systems Journal*, Feb. 2000. Available in PDF format at http://www.research.ibm.com/jalapeno/.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. The Java Series. Addison-Wesley Publishing Company, Redwood City, CA 94065, USA, 2000.

[7] M. D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, August 1998.

[8] J. Lee and D. A. Padua. Hiding relaxed memory consistency with compilers. In *Proceedings of The 2000 International Conference on Parallel Architectures and Compilation Techniques*, October 2000.

[9] J. Lee, D. A. Padua, and S. P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of The 1999 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1999.

[10] W. Pugh. Fixing the Java memory model. In *Proceedings of the ACM 1999 Java Grande Conference*, June 1999.

[11] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'02)
1087-4089/02 $17.00 © 2002 **IEEE**