Containers on the Parallelization of General-purpose Java Programs

Peng Wu* David Padua

Department of Computer Science
University of Illinois at Urbana-Champaign
{*pengwu, padua*}*@cs.uiuc.edu*

Abstract

Automatic parallelization of general-purpose programs is still not possible in general in the presence of irregular data structures and complex control-flows. One promising strategy is tread-level data speculation (TLDS). Although TLDS alleviates the need of proving independent computations statically, studies showed that applying TLDS blindly to programs with limited speculative parallelism may lead to performance degradation. Therefore, a positive approach is to combine TLDS with strong compiler analyses. The compiler can provide a guideline of where to speculate by "lazily" detecting some dependences and leave dependences that are more dynamic to be detected at runtime. Furthermore, transformations can be applied to eliminate some of the dependences detected by the compiler to enhance speculative parallelism in the program. This paper proposes compiler techniques to implement this approach. In particular, we focus on general-purpose Java programs with extensive use of containers *that refer to any general-purpose aggregate data structures.*

1 Introduction

For the ever-growing computing power to benefit most users, it is crucial that parallelism is exploited not only in numerical codes but also in most general-purpose applications. However, despite success in dense numerical computations, automatic parallelization of general-purpose programs is still not possible in general in the presence of irregular data structures and complex control-flows. One promising strategy is *tread-level data speculation* (TLDS) [10, 13, 4, 9, 12]. Under the TLDS model, a program can be parallelized speculatively in the presence of potential dependences: dependences are detected at run-time upon which program states will be recovered and the program will be re-executed. Although TLDS alleviates the need to prove independent computations statically, studies [4] show that applying TLDS blindly to programs with limited speculative parallelism may lead to performance degradation. Therefore, a positive approach is to combine TLDS with strong compiler analyses. The compiler can provide a guideline of where to speculate by "lazily" detecting some dependences and leave dependences that are more dynamic to be detected at runtime. Furthermore, transformations can be applied to eliminate some of the dependences detected by the compiler to enhance speculative parallelism in the program. This paper proposes compiler techniques to implement this approach. In particular, we focus on general-purpose Java programs with extensive use of *containers* that refer to any general-purpose aggregate data structures.

Containers in general-purpose programs play a role similar to that of arrays in numerical programs. In this work, we focus on three standard container classes commonly seen in general-purpose Java programs: Vector, LinkedList and Hashtable. Since it is very difficult for the compiler to analyze containers from the implementation, we encode semantics of basic container operations into the compiler as if they are primitive operations. The intuition is that most container classes, despite their complicated implementations, exhibit clean semantics at the level of program interface. By raising the level at which compiler analysis is conducted, we can greatly enhance the analyzability of dynamic data structures in Java programs. In this paper, we propose several analyses and transformation techniques for the containers we studied.

The rest of the paper is structured as follows. Section 2 characterizes several important container patterns. Sections 3 and 4 present several container-based analyses and transformation techniques. Experimental results are given in Section 5. Section 6 discusses future work and gives a conclusion.

*This work is supported in part by Army contract DABT63-95-C-0097; Department of Energy contract B341494; the National Science Foundation and the Defense Advanced Research Projects Agency under the OPAAL iniative and grant , NSF DMS 98-7394; and, a Partnership Award from IBM. This work is not necessarily representative of the positions or policies of the Army or Government.

2 Containers in Java Programs

The results reported in this paper are based on a study of four Java applications: javac, javap, javadoc and jar from JDK1.1.5. Javac is a Java compiler; javap is a Java byte-code disassembler; javadoc is a html-document generator for Java codes; and jar is a compression tool.

Although all four applications exhibit a moderate degree of inherent parallelism at the algorithm level, at the implementation level, we discovered various dependences in all of the major loops. One major source of these dependences is container operations. Three types of Java container classes are used prevalently in the benchmark suite: Vector, Hashtable, and LinkedList[8]. Vector implements an extensible array; LinkedList implements a linked list; Hashtable implements a hash-table. All these containers store object references. In this section, we will present a few container patterns that frequently introduce dependences in the benchmark suite. The discussion of container-induced dependences in general is deferred to Section 3.

2.1 Iterator-based loop

Iterator-based loops are formed by enumerating the components of a container through an iterator. Depending on the iterator used, iterator-based loops may take different forms. Figure 1 shows two typical iterator-based loops: the loop in Figure 1.a uses an iterator of type Enumeration, while the loop in Figure 1.b uses an iterator of type ListIterator.

```
Vector v;
Enumeration e;
for(e = v.elements(); e.hasMoreElements();) {
    o = e.nextElement();
    ...loop body...
}
```

(a) through Enumeration

```
ListIterator it;
LinkedList l;
for(it = l.listIterator(0); it.hasNext();) {
    o = l.next();
    ...loop body...
}
```

(b) through ListIterator

Figure 1: Iterator-base loops

In an iterator-based loop, the internal states of the iterator used in the pattern are changed in each iteration (e.g., by `nextElement`), and are read in the next iteration (e.g., by `hasMoreElements`). These accesses cause loop-carried dependences in iterator-based loops. Most major loops in the benchmarks are iterator-based loops.

2.2 Unique update

Unique update is a pattern that adds a "new" element (i.e., one that has not been added before), into a vector or a list. Figure 2 shows two unique update patterns. The pattern shown in Figure 2.a uses vectors only. The pattern in Figure 2.b uses a hash-table to determine whether an element is new in the vector or not. Since this hash-table is always updated by an object that serves as both the key and the value object, we refer to such a hash-table as a hash-set[1]. In the unique update pattern, the read and write accesses to the same container are the source of dependences.

```
if(!vector.contains(o)) {
     vector.addElement(o);
}
```

(a) vector-based

```
if(!hashtable.contains(o)) {
     hashtable.put(o,o);
     vector.addElement(o);
}
```

(b) hashset-based

Figure 2: Unique updates

2.3 Put-Get

`Put-Get` is a hash-table pattern as shown in Figure 3. A `Put-Get` first performs an ordinary `get` on the hash-table. If the entry is available, it returns immediately, otherwise, a new entry is added into the hash-table and is returned. The symbol table used in javac is implemented as a hash-table accessed by `Put-Get` operations. Classes sun.tools.java.Identifier and sun.tools.java.Type also implement `Put-Get` operations to access their internal hash-tables. Both classes are commonly used in the programs we analyzed. In this pattern, dependences are introduced by the `get` and `put` operations applied to the same hash-table.

3 Detecting Container-induced Dependences

Due to potential aliases between Java references, the dependence test necessary to determine whether a loop is parallelizable requires general pointer disambiguation. Although pointer analysis has been the subject

[1] In Java API1.2, HashSet becomes a standard Java class.

```
Entry getEntry(String name){
  Entry c = hashtable.get(name);
  if (c == null) {
      c = new Entry(name);
      hashtable.put(name,c);
  }
  return c;
}
```

Figure 3: Put-Get

of much research [1, 5, 3, 6, 2, 14, 7] it still remains a difficult problem, especially when analyzing composite and recursively-defined data structures. In this work, we target on only dependences introduced by container operations. We rely on a TLDS system to detect other dependences at runtime.

We consider both index-based loops and iterator-based loops in the dependence test. To give an example of the container-related dependences that may occur in a loop, consider the loop shown below where v is a vector; it is an iterator; and $foo()$ is a method that has side-effects on o.

```
for(it = v.elements();
     it.hasMoreElements();){
     o = it.nextElement();
     o.foo();
}
```

In this example, two potential dependences that are of particular interests: (i) the dependence introduced by operations on iterator it as mentioned in section 2.1, or in general dependences introduced by writing to and reading from the same container (for good reasons, we consider the iterator of a container as the internal state of the container); and (ii) dependences that result from accessing the same element of vector v in different iterations and at least one of the iteration writes to the element. We proposed a structural dependence test and an access analysis to detect these dependences respectively. In many aspects, we feel that access analysis for containers is analogue to the array-based dependence test for arrays.

3.1 Structural dependence test

Conceptually a container can be viewed as a storage media whose elements are stored in addressable "cells". A container is usually implemented as a storage area that holds references to Java objects, and other member variables that stores information such as the size of the container, etc. We define *container structural dependences* as dependences that result from read and write accesses to the same container[15]. There are three types of structural of dependences:

- If two operations write to the storage area of container (i.e., either *add into* or *remove from* a container), there is an *update dependence* (write-write). For instance, an update dependence is introduced by applying two `addElement` to one vector.
- An *access dependence* (read-read)[2] occurs between a read access and a write access to the iterator of the container. For instance, there is an access dependence between the `nextElement` and `hasMoreElements`) applied to the same iterator.
- Given two container operations, if one reads from the storage area while the other writes to the storage area of a container, there is a *value dependence* (read-write or write-read). For instance, there is a value dependence between operations `put` and `get` applied to one hash-table.

We represent the dependences defined above as structural dependence edges in a dependence graph. In the analysis framework, container operations are considered primitive and atomic by the compiler. We also build into the compiler a knowledge-base of the static dependence relations between each pair of container operations, i.e., which dependences are to be introduced if applying these operations to the same container. Figure 4 shows the static dependence relations defined on some basic operations of a vector and a hash-table. During the compilation, methods of classes

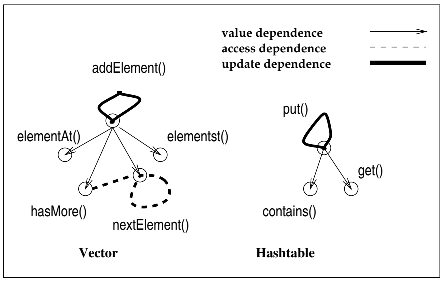

Figure 4: Static structure dependence relations

Vector, LinkedList, Hashtable, Enumeration, and ListIterator are represented as atomic operations in the intermediate representation (IR). Given a pair of container operations applied to one container, the dependence test can detect the dependence using its knowledge base of static dependence relations. For example, Figure 5 shows the structural dependence edges among statements `T1, T2` and `T3` of the following loop.

```
     for ( i = 0; i < n; i++) {
         ...
T1:      if(!hashtable.contains(o)) {
T2:           hashtable.put(o,o);
T3:           vector.addElement(o);
```

[2]It is considered as read-read because the dependence is usually introduced by consecutive read accesses to the storage area of a container

```
    }
}
```

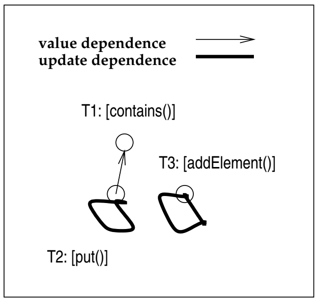

Figure 5: An example of the dependence graph

3.2 Access analysis

Access analysis summarizes accesses to the storage area of a container. Such access information can be used later in the dependence test to determine loop-carried dependences. For instance, if two iterations access the same "cell" of a container and the referred object is modified in one of the iterations, then there is a loop-carried dependence. Access information can also be used to detect finer-grained value dependence. Instead of treating the whole storage area of a container as one unit, we consider each "cell" as a separate one in the dependence test. Therefore, we detect a value dependence only if two computations read from and write to the same "cell" of a container.

Access analysis greatly depends on the addressing model of the container to be analyzed. Vector can be accessed implicitly by iterators, or explicitly through indices using the method `elementAt`; lists are addressed by iterators; hash-tables are accessed by keys.

For index-based container accesses, the analysis can summarize the access information using ranges of integers. For iterator-based container accesses, we translate an iterator-base access to an index-based access. Each iterator is associated with an index variable when it is bound to a container. The index variable is initialized to the position where the iterator initially points to. For instance, when obtainning an iterator through `vector.elements()`, an access index initialized to zero will be associated with the iterator. When an operation, such as `nextElement`, is applied to the iterator, the access index will be decremented. After such translation, iterator-based accesses can be summarized as ranges of integers as well. Since key-based accesses are difficult to be summarized symbolically and statically, we consider a runtime test most feasible for hash-tables. At runtime, access analysis can be conducted by recording keys used in the hash-table accesses in each iteration. And later a runtime dependence test can determine whether two iterations have accessed a hash-table using the same key[3]

4 Structural Dependence Elimination

Frequent occurrences of structural dependences in container-based programs become one of the major causes of limited inherent parallelism in the programs we analyzed in the study. Value dependences can be detected much more precisely by resolving dependences at the level of each container "cell". However, access and update dependences are almost statically built into basic container usage patterns.

4.1 Access dependence

We handle access dependences introduced by iterator-based loops. To parallelize such loops, each concurrent thread owns a local container and a local iterator. Elements of the container with which the iterator-based loop is associated are distributed into local containers prior to parallel execution[4]. Elements can be distributed either cyclically or by blocks. Then, each thread executes concurrently the loop based on its local iterator and container.

Vector-induced access dependences can be handled in a more efficient way. A vector can be accessed efficiently based on indices using the method `elementAt`. Therefore, we can transform the iterator-based loop, such as the one in Figure 1.a, into an index-based loop as shown below:

```
Vector v;
Enumeration e;
for(int i = 0; i < v.size; i++) {
    o = v.elementAt(i);
    ...loop body...
}
```

4.2 Hashtable-induced update dependence

We handle hashtable-induced update dependences introduced by two instances of `put` applied to one hashtable. Such dependences can be tolerated in parallelization if the two `put` are commutable. According to traditional commutativity analysis[11], two operations are commutable if different execution orders of the operations always produce the same memory states. This condition, however, is too restrictive in our case because internal states of a hashtable may be sensitive to the execution order of `put`. We define a much more relaxed commuting relation between two `put` operations.

[3]Hashtable compares two keys using the `equals` method of the key object. The runtime test need to compare key objects using their `equals` method if necessary.

[4]This transformation is valid only if no element in the loop is added or removed from the container.

Definition 4.1 *Let $\cdots, h_i, \cdots, h_j, \cdots, h_n$ be a sequence of operations on a hash-table. Two* `put` *operations, h_i and h_j, are* **commutable***, if any operation h_l, where $i < l < j$, always produces the same results under either execution order of h_i and h_j.*

The algorithm to determine whether two `put`, h_i and h_j, are commutable is shown in Figure 6. $h_i \rightarrow h_j$ stands for h_i value depends on h_j. The algorithm first tests whether operations between h_i and h_j in the sequence are sensitive to switching h_i and h_j (lines 1 to 4). If any tested operation value-depends on either h_i or h_j then we can not switch h_i and h_j. Next it tests operations occurring after h_j in the sequence (lines 7 to 13). Since a `put` will overwrite the entry set by a previous `put` using the same key, reordering two `put` using the same key will affect any subsequent operations that value-depend on them. This is tested by lines 10 to 12. In addition, because operation `elements` (`keys`) produces an iterator to enumerate the value (key) objects of the hash-table, re-ordering `put` may affect the order in which value (key) objects are visited. We can not re-order `put` when there are `elements` or `keys` operations following either `put`. This condition is tested by lines 8 to 9. Since `put` changes only internal states of a hash-table, we regard `put` as producing no (external) results; therefore, only operations other than `put` are tested.

Inputs: i, j, $h_i, \cdots, h_j, \cdots, h_n$
Output: whether h_i and hj are commutable

```
1  // test operations between h_i and h_j
2  foreach h_l ∈(h_i, h_j) that is not put() do
3    if (h_i →  h_l  |  h_l →  h_j)
4         return false;
5
6  // test operations after h_j
7  foreach h_l ∈(h_j, h_n] that is not put() do
8    if (h_l is elements() or keys())
9        return false;
10   if (h_i →  h_l  |  h_j →  h_l)
11       if ([h_i,  h_j] use duplicate key)
12          return false;
13
14 return true;
```

Figure 6: Determine two `put` commutable

Commutable operations can be gainfully exploited in loop parallelization. In the example shown below, assuming that the update dependence is the only loop-carried dependence in the loop. Using the algorithm in Figure 6, we can prove that any two instances of `put` in the loop are commutable. Therefore, the `put` in the loop can be re-ordered and the loop is parallelizable (given that `put` is synchronized, which guarantees atomicity in the parallel execution).

```
for (i = 0; i < n; i++) {
   ...
   hashtable.put(new Integer(i), obj);
}
hashtable.get(key1);
```

4.3 Vector-induced update dependence

Update dependences that occur when two `addElement` are applied to one vector may be eliminated by exploiting associativity. For instance, a sequence of `addElement` applied to a vector is *associable.* Such a sequence is referred to as *associable updates.* Parallelizing associable updates is analogous to reduction parallelization in that during the parallel execution, each thread updates a local vector which will be merged after the loop. However, unlike reduction parallelization, scheduling schemes play a factor in parallelizing associable updates. For self-scheduling and cyclic-scheduling, the local vector will record both the element updated and the iteration that issues the update. While merging local vectors, an element must be inserted after all the elements issued by previous iterations are inserted.

Associativity can be exploited on unique updates like those shown in Figure 2 as well. A sequence of unique updates is *associable* if the hash-table, if any, is commutable. Such a sequence is called *associable unique updates.* For the purpose of analysis, in the IR, we represent the code sequence that implements the unique update pattern as a primitive and atomic operation, `addUnique`. For instance, the codes in Figures 2.a and 2.b are represented as `vector.addUnique(o)` and `vector.addUnique(o,hashtable)`, respectively. We parallelize associable unique updates only under block-scheduling. During parallel execution, each thread conducts `addUnique` on a local vector and a local hash-table, if any. Then after the parallel execution, local vectors and local hash-tables are merged uniquely.

5 Putting it All Together

5.1 Understanding Parallel Behavior

General-purpose programs exhibit much more dynamic parallel behaviors than numerical codes. We classify dependences in the program into dynamic dependences and static dependences.

5.1.1 Dynamic Dependences

Dynamic dependences are dependences that depend either on input sets, or on iterations (i.e., dependences may occur on some input set or in some iterations, but not others). For instance, a loop that may throw catchable exceptions contains control dependences (i.e., exception handling may cause the loop an early-exit). Such dependences are dynamic because they occur only in some iterations of a loop and on some input sets. Dynamic dependences resulting from potential exceptions are very common in Java loops. Of the benchmarks we studied, another major source of dynamic dependences is hashtable-induced value dependences, e.g., `get` and `put` an entry from/to a hash-table using the same key. Such dependences are dynamic because most of the keys used in the programs are strings taken from input data. In addition, when reading from and writing to the same file, file I/O operations introduce dependences as well. Such dependences are dynamic since filenames used to open a file are taken mostly from input data.

TLDS provides the opportunity to exploit parallelism on loops with dynamic dependences. In fact, for the benchmarks we studied, except `javac`, most dynamic dependences in the programs are not realized under normal input sets. On the other hand, static dependence test models always fell short in exploiting this type of parallelism.

5.1.2 Static Dependences

Static dependences usually do not depend on input sets nor iterations. Programs with static dependences usually contain very limited speculative parallelism. In our benchmark suite, sources of static dependences are update dependences, access dependences, reduction variables, privatizable objects, and I/Os that write to standard output. Table 1 summarizes container-induced static dependences for the benchmarks we studied. We classify the source by the pattern used where *I-loop* stands for iterator-based loops and *unique* stands for unique update.

		Access	Update		
Program-loop	$\%T_{seq}$	I-loop	Put-Get	Unique	Other
javac-parse	24%	X	X	X	
javac-compile	40%	X	X	X	
javap-main	98%	X	X		
javadoc-parse	31%	X	X	X	
javadoc-gen	54%	X	X		X
jar-addFiles	75%	X		X	

Table 1: Static structural dependences in major loops

Static analysis is crucial to effectively exploit parallelism on the programs we analyzed. On one hand, some of the static dependences can be eliminated by transformations to levitate inherent parallelism in the program. Of the loops in Table 1, all the update and access dependences can be eliminated. On the other hand, the existence of static dependences can serve as a guideline of where not to speculate.

5.2 Hand-parallelization

To measure the speculative parallelism that would be exposed after eliminating static dependences, we hand-parallelized the benchmark suite[5]. We applied the techniques proposed in the paper by hand to transform static update and access dependences. Iterator-based loops were converted into index-based loops. Synchronizations were added to commutable `putGet` operations and associable `addUnique` operations. For associable operations, we allocated local containers for each thread, and implemented procedures to merge the local containers properly. We allow I/Os that write to standard output to be out of order. We also applied reduction parallelization and privatization by hand. After these transformations, we consider a "parallelizable" loop as one that is free of static dependences (assuming ideal alias analysis) and free of dynamic dependences under normal input sets.

5.3 Performance

Since we did not have a TLDS system for experiments, we avoid dynamic dependences by carefully choosing the input sets. Performances were measured on a SUN Enterprise 450 server with four UltraSPARC 167 nodes. All the benchmarks ran under JDK1.2Beta. The measured performance did not take into account the run-time overhead and speculation penalty of a TLDS system. In fact, the execution of a Java program could be decomposed into: class loading, class verification, just-in-time (JIT) compilation, interpretation, and garbage collection. Source-level parallelization can only directly improve interpretation time. During the experiments, we disabled JIT and class-file verification; runtime heap and stack were specified large enough to guarantee that no garbage collection would occur; and the JVM was "warmed-up" by pre-loading all the necessary classes. Eliminating these disturbing factors makes the measured speedup a reasonable indicator of the maximum speculative parallelism that can be exposed by source-level parallelization.

As shown in Figure 7, for each benchmark, we measured the performance of major parallelizable loops and the overall program under two, three, and four processors, respectively. The numbers above each bar show the speedups. With realistic inputs, most parallel loops

[5] For `jar` a recursive algorithm that expands directories is changed to be non-recursive. We changed the output of `javap` so that disassemble class-files are put into separate files in stead of standard output. In `javac`, the class loading/resolving algorithm was simplified and the inlining pass and supports for inner classes were disabled

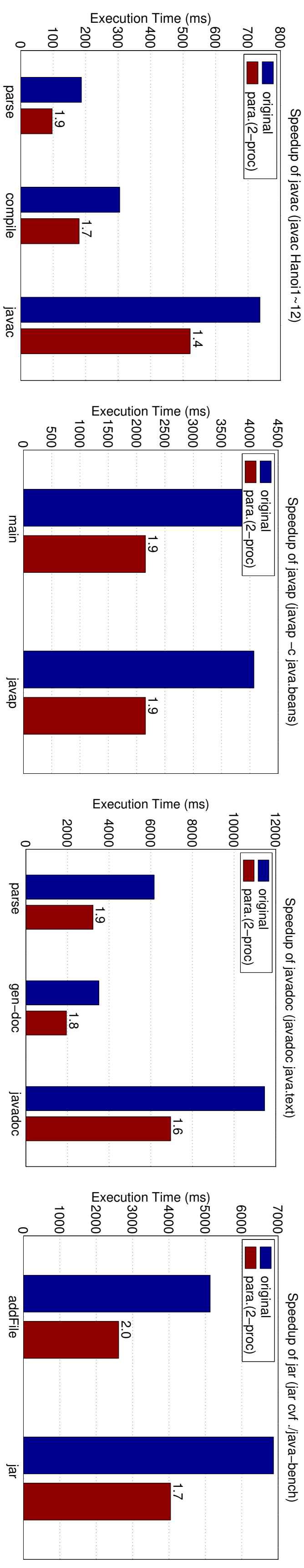

Figure 7: Summary of the speedups

have achieved more than 1.8 speedups out of 2 processors and 2.5 speedups out of 3 processors. However, speedups for 4 processors only ranged from 2.2 to 3.0. The major reason that the speedups did not improve from 3 processors to 4 processors is because normal input sets under 4 processors suffered severe load imbalancing. Also, as shown in Figure 7, the speedups of the overall programs ranged only from 1.6 to 2.2 for 4 processors. Except for javap, speedups of overall programs were brought down because of the sequential loops[6] in the programs.

6 Conclusion

Container-induced dependences contribute to a large portion of static dependences occurring in general-purposed programs. Such dependences greatly limit the inherent parallelism available in general-purpose programs, thereafter the speculative parallelism exploitable by a TLDS system. In this work, we studied three standard Java container classes: Vector, LinkedList, and Hashtable. We proposed analysis techniques to detect container-induced dependences and transformation techniques to eliminate some static structural dependences.

References

[1] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proc. of the ACM SIGPLAN'90 conf. on Programming Language Design and Implementation*, June 1990.

[2] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proc. of the 23rd ACM/SIGPLAN SIGACT Symp. on Principles of Programming Languages*, January 1996.

[3] R. Ghiya and L. J. Hendren. Putting pointer analysis to work. In *25th ACM Symp. on Principles of Programming Languages*, January 1998.

[4] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[5] L. J. Hendren and G. R. Gao. Designing programming languages for analyzability: A fresh look at pointer data structures. In *Proc. of International Conf. on Computer Languages*, April 1992.

[6] J. Hummel, L. J. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1994.

[6] In jar, the loop that outputs compressed files into one jar-file is inherent sequential. In javac and javadoc, the loop doing type checking is also inherently sequential

[7] J. Hummel, L. J. Hendren, and A. Nicolau. A language for conveying the alising properties of dynamic, poiinter-based data structures. In *Proc. of the 8th Inthernational Parallel Processing Symposium*, April 1994.

[8] S. Microsystem. Java platform 1.2 api specification. http://www.javasoft.com/products/jdk/1.2/docs/.

[9] J. Oplinger, D. Heine, S.-W. Liao, B. A. Nayfeh, M. Lam, and K. Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Stanford University Computer Systems Lab, February 1997.

[10] L. Rauchwerger and D. Padua. The lrpd test: Runtime parallelization of loops with privatization and reduction parallelization. In *ACM SIGPLAN'95 Conference on Programming Languages Design and Implementation*, June 1995.

[11] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Lanugages and Systems*, 19, November 1997.

[12] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, July 1995.

[13] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[14] P. Wu, P. Feautrie, and D. Padua. Combness analysis: a fresh approach for shape analysis. Technical report, University of Illinois, Urbana-Champaign, Computer Science, July 1999.

[15] P. Wu and D. Padua. Beyond arrays – a container-centric approach for paralllelization of real-world symbolic applications. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing, LCPC'98*, 1998.