Containers on the Parallelization of General-purpose Java Programs

Peng Wu *   David Padua

1304 W. Springfield Ave.
Urbana, IL 61801
Department of Computer Science
University of Illinois at Urbana-Champaign
217-244-5979   217-333-3501 (fax)
{*pengwu, padua*}@*cs.uiuc.edu*

Abstract

Static parallelization of general-purpose programs is still impossible, in general, due to their common use of pointers, irregular data structures, and complex control-flows. One promising strategy is to exploit parallelism at run-time. Run-time parallelization schemes, particularly data speculations, alleviate the need to statically prove independent computations at compile-time. However, studies show that many real-world applications exhibit limited speculative parallelism to offset the overhead and penalty of speculation schemes. This paper addresses this issue by using compiler analyses to compensate for speculative parallelizations. We focus on general-purpose Java programs with extensive use of Java container classes. In our scheme, compilers serve as a guideline of where to speculate by "lazily" detecting dependences that are mostly static, while leaving those that are more dynamic to run-time. We also propose techniques to enhance speculative parallelism in the programs. The experimental results show that, after eliminating static dependences, the four applications we study exhibit significant parallelism that can be gainfully exploited by a speculative parallelization system.

KEYWORDS: compiler analysis, parallelism, data speculation, dependence analysis

*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 initiative 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.

1 Introduction

Static parallelization of general-purpose programs is still not possible, in general. The difficulty lies in the need to prove independent computations statically in the presence of pointers, irregular data structures, and complex control-flows. One promising alternative is to defer dependence detection to run-time when program states are easily available. Various data speculation schemes have been proposed to implement such an approach [10, 13, 4, 9, 12]. Using speculative parallelization, a program can be parallelized with potential dependences. A dependence detection scheme will safeguard correct program semantics during the execution by monitoring every memory access at run-time. When dependence violations occur, a recovery mechanism will bring the execution back to a safe program point. Due to the overhead of run-time dependence checking and the penalty of miss speculation, speculative parallelization is feasible only for programs that have rare occurrences of dependences at run-time. However, as shown in the study[4], many real-world general-purpose programs exhibit limited speculative parallelism. Applying data speculation blindly to such programs will always result in performance degradation.

This paper addresses this issue by using compiler techniques to complement speculative parallelization. In our scheme, the compiler serves as a guideline of where to speculate by "lazily" detecting some dependences. Dependences can be characterized into two types: static dependences and dynamic dependences. Static dependences refer to those that most likely occur regardless run-time execution paths or the input data set. Dynamic dependences refer to those that usually occur along rarely-executed paths or under abnormal conditions. Our "lazy" dependence test aims at capturing and eliminating as much as possible static dependences. At the same time, our "lazy" scheme tolerates dynamic dependences so that speculative parallelism will not be compromised by overshooting "rarely occurring" dependences.

The results reported in this paper are based on a study of four Java applications: javac, javap, javadoc, and jar. We observe that, of the four applications, the majority of static dependences are introduced by operations on Java container classes, particularly by Vector and Hashtable operations. We, therefore, focus our analyses on these two classes. The semantics of containers are crucial for the analyses to succeed. In our scheme, the semantics of container classes are built into the compiler as if they are primitive data types since such information is very difficult to be extracted automatically by the compiler.

This work consists of two core parts:

- We present a qualitative analysis of the dependences that occur during the program execution and the effects that these dependences have on speculative parallelizations.
- We focus on dependences that are introduced by container operations (container-induced dependences) and propose compiler techniques to detect and eliminate such dependences.

The experimental results show that, after eliminating container-induced dependences, all of the four applications exhibit significant parallelism that can be gainfully exploited by a speculative parallelization system.

This paper is structured as follows. Section 2 classifies container-induced dependences that occur in the programs we studied. Section 3 presents the analyses to detect container-induced dependences. Section 4 introduces transformation techniques to eliminate container-induced dependences. Section 5 presents our experimental results. And, Section 6 concludes the work.

2 Dependences and Container-induced Dependences

The results reported in this paper are based on a study of four Java applications from SUN's JDK1.1.5 package: javac, javap, javadoc and jar. Javac is a Java compiler; javap is a bytecode disassembler; javadoc is a document generator for Java sources; and jar is a compression tool. The rest of the section is organized as follows: section 2.1 gives a qualitative analysis of the dependences we found in the programs; section 2.2 covers the basics of Java container classes; section 2.3 introduces basic container-induced dependences; and section 2.4 presents dependences resulting from composite container operations.

2.1 Understanding Dependences in the Programs

All four applications suggest potential coarse-grain loop-level parallelism at the algorithm level. For instance, javadoc takes multiple Java source files as input and generates html-documents for each source file independently. At the coding level, however, most major loops of the programs contain cross-iteration dependences. To better understand the impact of such dependences on a speculation system, we classify them into dynamic dependences and static dependences. Note that the dependences we discuss here are "real" dependences in the sense that there exists a feasible execution path of the program under which the dependence will occur.

2.1.1 Static Dependences

Static dependences are dependences that most likely occur regardless of run-time execution paths or input data-sets. In the programs we studied, static dependences are mainly introduced by multiple reads and writes to one scalar variable or to fields of one object, or by I/O operations that write to the standard output. Among them, dependences resulting from multiple accesses to one container object, primarily Vector and Hashtable[8] objects, occur the most frequently.

Usually occurring along frequently executed paths of a program, static dependences become a limiting factor of the amount of speculative parallelism in a program. For this reason, the four applications we studied, without any transformation, would benefit little from a data speculation scheme. It may be possible, however, to identify and eliminate some static dependences at compile-time. The rest of the paper will focus on how to identify and eliminate some static dependences, particularly container-induced static dependences. Static dependences that are not eliminated can guide the run-time system on where not to speculate.

2.1.2 Dynamic Dependences

Dynamic dependences refer to dependences that usually occur along rarely-executed program paths (e.g., under abnormal conditions) or those that are input-dependent. For example, a loop that may throw exceptions (e.g., null-pointer-exceptions and array-out-of-bound-exceptions) contains potential loop-carried control dependences. Without proving the loop to be exception-free, a static analysis has to assume, conservatively, loop-carried dependences in the loop. This excludes most Java loops from being statically identified as parallelizable. Another type of dynamic dependence particular to javac, javap, and javadoc, is introduced by multiple reads and writes to one hash-table. Hash-tables are addressed by keys (see section 2.2). If two iterations access one hash-table using the same key, and at least one of them is a write, then there is a loop-carried dependence. In javac, javap, and javadoc, there is one hash-table that is frequently looked-up and updated by different iterations. Since the keys used in these accesses are from input data, a static dependence test has to assume a potential loop-carried dependence.

Speculative parallelization provides a great opportunity to exploit parallelism in loops that contain mainly dynamic dependences. Static parallelizations always fell short in exploiting this kind of parallelism. The main reason is that many dynamic dependences tend to be realized only under rarely executed program paths (e.g., exceptions that are due to abnormal program executions). For the benchmarks we studied, except for javac, most dynamic dependences are not realized under normal input sets.

2.2 Java Container Classes

Java container objects store object references. The elements of a container refer to object references that are stored in the container. An empty container contains no elements. A container object usually consists of a major storage for its elements and other auxiliary fields that hold information such as the size of the container, etc. An iterator, when associated with a container, provides a way to access the elements of a container in a certain order.

Standard Java class library provides a variety of container classes [8]. In this work, we focus on two of the container classes: Vector and Hashtable. Class Vector implements extensible arrays. Figure 1.a shows the fields of Vector and vector operations used in the programs we studied. Modifier *synchronized* declares the method to be executed atomically. Method `elementAt` provides an interface to access a vector through indices. A vector also can be accessed using iterators of type java.util.Enumeration. The class declaration of Enumeration is shown in Figure 2.

Class Hashtable implements hash-tables that store pairs of object references. One is referred to as the *key*; the other is called the *value object*. Hash-tables are accessed by keys. Keys of a hash-table are kept *distinct*[1]. Figure 3 shows the fields of Hashtable and the hash-table operations used in the programs we studied. Both the keys and the value objects of a hash-table can be accessed through iterators of Enumeration type.

[1] Two Java object references are *distinct* if they contain different values; otherwise they are *duplicate*.

```
class java.util.Vector {
Object elementData[];
int elementCount;

int size();
synchronized Object elementAt(int i);
synchronized Enumeration elements();
boolean contains(Object o);
synchronized void addElement(Object o);
...
}
```

Figure 1: Vector

```
interface java.util.Enumeration  {
    boolean hasMoreElements();
    java.lang.Object nextElement();
}
```

Figure 2: Enumeration

2.3 Container-induced Dependences

Roughly speaking, a container-induced dependence can be any memory conflict resulting from read- and write- accesses to the same field of a container object. For example, consider the following code:

```
...
s: v.addElement (o1);
t: v.addElement (o2);
```

`v` is a vector. Statements `s` and `t` both read and write to field `v.elementCount` and `v.elementData` (refer to Figure 1). In this simple example, there are anti-, output-, and flow- dependences between `s` and `t`. In Vector, method `addElement` modifies the internal fields of a vector. In general, there are dependences when two operations are applied to the same vector and at least one of them is an `addElement`. In class Hashtable, the equivalent of `addElement` of Vector is method `put`. Similar dependences are produced when two operations are applied to a hashtable and at least one of them is a `put`.

A pair of container operations are said to be *conflicting* if they introduce dependences when being applied to the same container object. For example, for Hashtable, operations `put` and `put`, and operations `put` and `get` are conflicting operations. Tables 1 and 2 summarize the conflicting operations of Vector and Hashtable, respectively. These tables are referred to as the dependence tables of the corresponding classes.

Iterator operations introduce dependences as well. For example, the following code enumerates elements of hash-table `h` through iterator `enum`.

```
for (e = h.elements(); e.hasMoreElements();)
    o = e.nextElement();
```

```
class java.util.Hashtable {
private  HashtableEntry table[];
private int count;

int size();
synchronized java.util.Enumeration keys();
synchronized java.util.Enumeration elements();
synchronized Object get(Object key);
synchronized boolean contains(Object key);
synchronized Object put(Object key, Object val);
...
}
```

Figure 3: Class Hashtable

	addElement
size	X
elements	X
elementAt	X
contains	X
addElement	X

Table 1: Dependence table of Vector

	put
size	X
keys	X
elements	X
get	X
contains	X
put	X

Table 2: Dependence table of Hashtable

`e.nextElement` accesses elements of `h` through an internal pointer, and then advances the internal pointer to point to the next element. `e.hasMoreElement` compares the internal pointer with the end position of the container. There are dependences between a `e.hasMoreElements` and a `e.nextElement`, and between two occurrences of `e.nextElement`. Note that if a vector or a hash-table is structurally modified at any time after the iterator is created, accessing the container through the iterator will throw a ConcurrentModificationException. For this reason, we need not consider dependences between operations that modify a container and operations that access the container through iterators.

2.4 Composite Container-induced Dependences

In the programs we studied, most container-induced dependences are manifested by a composition of several container operations. This section presents three patterns that are composed from multiple container operations: *iterator-based loops*, *unique updates*, and *Put-get*.

2.4.1 Iterator-based loop

Iterator-based loops are loops whose iterations are controlled by enumerating elements of a container through an iterator. Iterator-based loops take on different forms depending on the iterators being used. Figure 4 shows an iterator-based loop using an iterator of type Enumeration. As discussed earlier, operations on iterators introduce cross-iteration dependences on iterator-based loops.

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

Figure 4: Iterator-base loops

2.4.2 Unique update

Vectors usually do not contain duplicate elements. Unique updates are used to update vectors whose elements are always kept distinct. This is done, as shown in Figure 5.a, by adding an object to a vector only if it has not been added into the vector before. Figure 5.b shows another common implementation of unique updates. It uses an auxiliary hash-table to keep track of all elements that have being put into a vector. For each pair of objects stored in the hash-table, the key and the value object are the same. We refer to such a hash-table as a hash-set[2]. Implementing a unique update by using a hash-table is more common.

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

(a) using vector

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

(b) using hash-set

Figure 5: Unique updates

2.4.3 Put-get

`Put-get` provides a special way to "get" an element from a hash-table. `Put-get` acts as an ordinary `get` when the hash-table contains the element to be retrieved; otherwise, it puts a new element into the hash-table (i.e., the "put" part), and return the element (i.e., the "get" part). Figure 6 shows an example of `Put-get`.

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

Figure 6: Put-get

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

`Put-get` combines the `put` and `get` operations together. In javac, javap, and javadoc, a hash-table operated by `put-get` implements the symbol table. `Put-get` is also used to access internal hash-tables in class sun.tools.java.Identifier and sun.tools.java.Type. Identifier maintains a hash-table that registers all the Identifier objects being created. `Put-get` ensures that, for any given identifier name, there exists only one Identifier object. Type uses `Put-get` for the same purpose.

3 "Lazy" Dependence Tests

A "lazy" dependence test assumes that all computations in the programs are independent, by default, unless they can be proven otherwise. For this "over-optimistic" dependence test to work correctly, a "lazy" dependence test needs to be combined with a run-time scheme. The latter will detect dependences at run-time and safeguard the correct execution of the program.

In this section, we propose two "lazy" dependence tests: the first detects container-induced dependences; the other, referred to as the *access analysis*, determines whether two computations access the same element of a container. In many respects, access analysis for containers is analogous to the array-based dependence test for arrays.

3.1 Detecting Container-induced Dependences

The analyses presented in this work are based on the semantics of container classes and their operations. This information is directly built into the compiler so that the compiler treats container classes as primitive data types, and container operations as primitive and atomic. We also built into the compiler the dependence table (see section 2.3) of each container class. Since internal states of a container object are accessible only through its operations, detection of container-induced dependences is fairly simple when provided with the dependence table of the class. There are dependences if any two *conflicting operations* are applied to the same container object.

We still need alias analyses to determine the set of container operations that are applied to a container object and its aliases. For example, in the following code,

```
       v = new Vector();
       u = v;
   s:  v.addElement();
   t:  u.addElement();
```

both statements `s` and `t` have updated the same container object. However, this alias problem is much easier to handle than a general alias problem. A nice property is that the fields of a container object are exposed only to the container object, its aliases (i.e., those that point to the same object), and any iterator associated with it. Many existing pointer analyses [1, 5, 3, 6, 2, 7] can handle this case. With the help of this type information, the analyses can be fast and simple.

3.2 Access analysis

For Vector and Hashtable, the fields (i.e., field `elementData` and `table`) to store container elements are arrays of object references. Because it would be too imprecise to treat such fields as one unit in the dependence test, we name *positions* for containers and treat different positions of a container as different memory accesses. For Vector, positions are named by integers. For example, the first element of a vector is considered at position 0, and an access `v.elementAt(i)` will be interpreted as accessing the position `i` of the vector `v`. For Hashtable, positions are named by keys.

Access analysis summarizes accesses to container elements in terms of positions. There are three ways to access elements of a vector: through iterators of type Enumeration, through indices using method `elementAt`, and through method `addElement`. The first two read from the major storage of a vector, while the last one writes to it. For iterator-based accesses, we associate each iterator with a variable *curr*. *Curr* indicates the position of the container pointed to by the iterator. When an iterator is created through method `elements`, *curr* is set to zero. When an element is accessed through `nextElement`, it is interpreted as accessing the position *curr* of the container. It also increments *curr* by one. Method `addElement` is interpreted as accessing position *end* of the container. It increments *end* by one afterwards. After such mapping, reads and writes to the major storage of a vector can be summarized as ranges of integers. An index-based dependence test similar to the one for arrays can be applied. For instance, if two iterations access the same ”position” of a container and at least one iteration modifies the accessed element, then there is a loop-carried dependence. The dependence test is “lazy” because different “positions” of a vector may contain the same object reference.

Positions of hash-tables are named by keys. A hash-table can be accessed by `get`, `contains`, `put`, or iterators. Operations `get(key)`, `contains(key)`, and `put(key, obj)` access the position *key* of the hash-table. For accesses through iterators, we assume that they can access any position of the hash-table. Since the keys of a hash-table are usually string objects or general objects, it is difficult to summarize keys statically and symbolically. We consider a run-time access analysis most feasible for hash-tables. The run-time access analysis can record all the keys used to access a hash-table. A run-time dependence test can determine whether two iterations access the same position of a hash-table by comparing the keys[3].

4 Eliminating Container-induced Dependence

4.1 Iterator-induced dependences

Iterator-based loops, as discussed in section 2.4.1, contain loop-carried dependences. We handle such dependences by transforming iterator-based loops into index-based loops. The following shows the index-based loop that is transformed from the iterator-based loop shown in Figure 4.

[3]Hash-table compares two keys using the `equals` method of the key object. The run-time test needs to compare key objects using their `equals` method if necessary.

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

4.2 Hashtable-induced dependence

When two hash-table operations are applied to the same object and at least one of them is a `put`, there are hashtable-induced dependences. Such dependences can be tolerated in a parallel execution if they can be proven commutable.

Traditional commutativity analysis[11] proves two operations commutable if executing the operations in different orders will produce the same states of the memory. This condition, however, is too restrictive for handling hash-table operations. For example, applying two `put` to a hash-table in different orders may produce different internal states of the hashtable. Therefore, we introduce a more relaxed commutable relation (i.e., two operations are commutable if the computation that follows is insensitive to the order in which the operations are executed). For example, consider the following code sequence, which assumes that `key1`, `key2`, and `key3` are different.

```
...
s: ht.put(key1, obj1);
t: ht.put(key2, obj2);
u: ht.get(key3, obj3);
```

Since each statement accesses a different position of `ht`, `u` is not affected by `s` and `t`. Statements `s` and `t` are commutable. We define two hash-table operations as commutable as follows:

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

Algorithm 1 shows how to determine whether two hash-table operations (i.e., two `put`; a `put` and a `get`; or, a `get` and a `put`) are commutable.

Algorithm 1 h_i *and* h_j *are two hash-table operations and at least one of them is a* `put`*.* $h_i, \cdots, h_j, \cdots, h_n$ *is a sequence of operations applied to one hash-table.* key_l *is the key used by operation* h_l*. The algorithm determines whether* h_i *and* h_j *are commutable.*

```
if (key_i  =  key_j )
   return false;

//check for operations between h_i and h_j
foreach h_l that l ∈ (i, j) do
 if (h_l is get(), contains(), or put()) {
       if ( h_i is put() & key_i  =  key_l)
```

```
6            return false;
7         if ( h_j is put() & key_l  =  key_j)
8            return false;
9     } else if (h_l is elements() or keys())
10        return false;

    //check for operations after h_j
11  foreach h_l that l∈(j,n] do
12     if (h_l is elements() or keys())
13        return false;

15 return true;
```

The algorithm first checks whether operations between h_i and h_j in the sequence are affected by the reordering of h_i and h_j (line $1 \sim 10$). For operations such as `get`, `put`, and `contains`, the test is based on the keys used. Operation `elements` or `keys`, however, return an enumeration of the value objects or keys of a hash-table. They are sensitive to the re-ordering of h_i and h_j. Then, the algorithm checks whether operations after h_j in the sequence will be affected if h_i and h_j are re-ordered(line $11 \sim 13$).

Hash-tables that are accessed through `put-get` operations need to be handled specially. Instead of treating each hash-table operation in a `put-get` individually, we treat `put-get` as one operation. `Put-get` operations have a nice property: a sequence of `put-get` operations are commutable no matter which keys are used.

Commutable operations can be gainfully exploited in loop parallelization. Consider the following loop:

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

We can prove that any two instances of `put` in statement `s` are commutable. If these hashtable-induced dependences are the only dependences in the loop, the loop can be parallelized provided that `put` executed atomically. This is done automatically since `put` is synchronized. When parallelizing loops with `put-gets`, however, we need to add synchronizations to ensure that each `put-get` is executed atomically.

4.3 Vector-induced dependence

Dependences occur when two `addElement` operations are applied to the same vector. Such dependences may be eliminated by proving the operations associable. For instance, given a sequence of operations that are applied to a vector, if all of them are `addElement` operations, the sequence is *associable*. We refer to such a sequence of operations as associative updates. Parallelizing associable updates is analogous to parallelizing reduction variables: for each parallel task, we allocate a local vector that will be updated during the execution of the task; after all the iterations are finished, local vectors are merged into one vector. Since associable updates are not commutable, the merging process has to be done in the order of the iterations.

More complicated scheduling schemes, such as self-scheduling, may require auxiliary data structures to associate an iteration number with the elements of the local vector.

A sequence of unique updates is also associable. Again, we treat a unique update as one operation. To parallelize associable unique updates, for each parallel task, we allocate a local vector and, if the unique update uses hash-tables, a local hash-table. Each task updates its local vector and hash-table during the execution. After all the iterations are finished, local vectors and hash-tables are merged. In the merging process, an element is added into the vector only if it is not in the local vectors of previous tasks. In another words, local vectors need to be merged into the global vector using unique updates.

5 Experimental Results

5.1 Hand-parallelization

To measure the parallelism exposed after eliminating static dependences, we hand-parallelized javac, javap, javadoc, and jar[4]. We consider a loop "parallelizable" if it is free of static dependences and free of dynamic dependences under normal input sets. Since we do not have a speculation system for experiments, we avoid dynamic dependences by carefully choosing the input sets. Fortunately, for the applications we studied, most dynamic dependences are not realizable under normal input sets.

We apply the techniques proposed in this paper by hand to transform container-induced dependences. Iterator-based loops are converted into index-based loops. Commutable `put-get` operations are declared as *synchronized*. For associable updates, we allocate local containers for each thread and implement the procedures to merge local containers. For other static dependences, we apply reduction parallelization and privatization by hand. I/Os that write to standard output can be out of order. Table 3 summarizes the loops that are "parallelizable"after these transformations, the percentage of execution time each loop takes up, and the container-induced dependences each loop contains. $I\text{-}loop$ stands for iterator-based loops. $unique$ stands for unique update. $Basic$ covers basic container-induced dependences that are not part of the previous three.

Program-loop	$\%T_{seq}$	I-loop	Put-Get	Unique	Basic
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 3: Container-induced dependences in major loops

[4]For jar, a recursive algorithm is changed to be non-recursive. The output of javap is changed so that disassembled class-files are put into separate files instead of standard output. In javac, the class loading and class resolution algorithm was simplified and the inlining pass and supports for inner classes were disabled.

5.2 Performance

We measure the performance on a SUN Enterprise 450 server with four UltraSPARC 167 nodes. All the programs run under JDK1.2Beta. The measured performance does not include run-time overhead and speculation penalty of a speculation system. The execution of a Java program consists of class loading, class verification, just-in-time (JIT) compilation, interpretation, and garbage collection. Source-level parallelization can directly improve only the interpretation time. To make the reported speedups a reasonable indicator of the speculative parallelism that can be exposed by source-level parallelization, we set up the experiment so that time spent on other parts of the JVM is minimized. We disable JIT and the class-file verification pass of the JVM. Run-time heap is specified large enough so that no garbage collection occurs during the experiments. We also pre-load all the necessary classes to eliminate class loading time.

As shown in Figure 7, for each program, we measure the speedups of the "parallelizable" loops as well as speedups of the overall programs under two, three, and four processors, respectively. The numbers above each bar show the speedups. With realistic inputs, most parallel loops achieve very good speedups under two- and three- processor executions. However, execution time under four processors does not improve much from that under three processors. This is because most loops we parallelized contain a limited number of iterations (tens $\sim$ hundreds), while each iteration contains a large amount of computation. Because of the coarse-grained parallelism we are exploit, execution under four processors suffers severe load imbalance.

6 Conclusion

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

References

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

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

[4] Lance Hammond, Mark Willey, and Kunle 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] Laurie J. Hendren and Guang 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] Joseph Hummel, Laurie J. Hendren, and Alex 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.

[7] Joseph Hummel, Laurie J. Hendren, and Alex 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] Sun Microsystem. Java platform 1.2 api specification. http://www.javasoft.com/products/jdk/1.2/docs/.

[9] Jeffery Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica Lam, and Kunle 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] Lawrence Rauchwerger and David Padua. The lrpd test: Run-time parallelization of loops with privatization and reduction parallelization. In *ACM SIGPLAN'95 Conference on Programming Languages Design and Implementation*, June 1995.

[11] Martin C. Rinard and Pedro 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. Gregory Steffan and Todd 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.

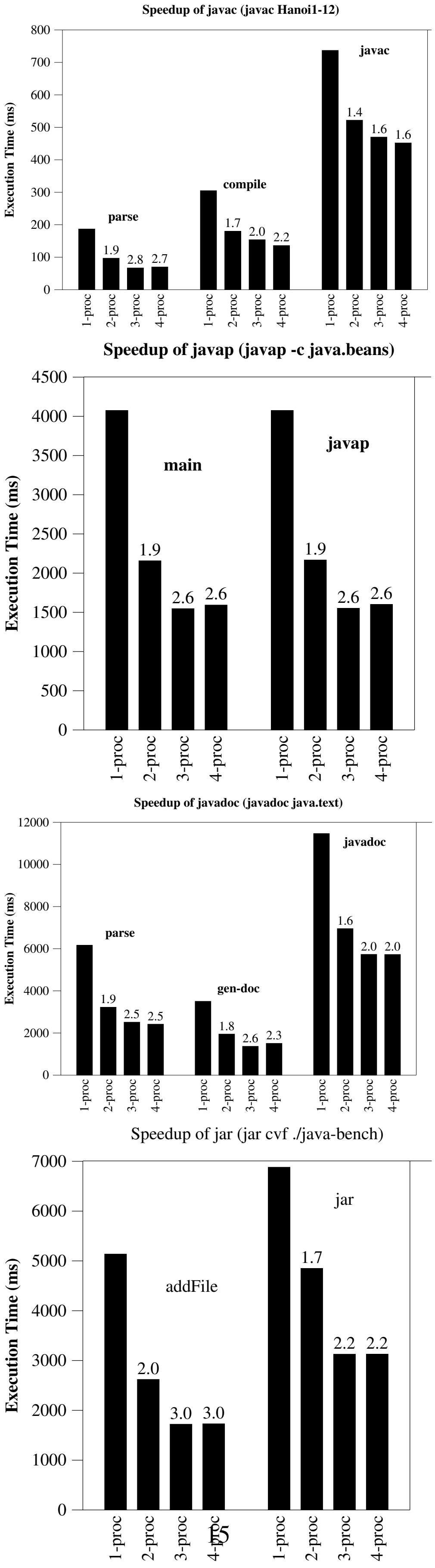

Figure 7: Summary of the speedups