Java Subtype Tests in Real-Time

Krzysztof Palacz Jan Vitek

S[3] Lab, Department of Computer Sciences, Purdue University

Abstract. Dynamic subtype tests are frequent operations in Java programs. Naive implementations can be costly in space and running time. The techniques that have been proposed to reduce these costs are either restricted in their ability to cope with dynamic class loading or may suffer from pathological performance degradation penalizing certain programming styles. We present R&B, a subtype test algorithm designed for time and space constrained environments such as Real-Time Java which require predictable running times, low space overheads and dynamic class loading. Our algorithm is constant-time, requires an average of 10.8 bytes per class of memory and has been shown to yield an average 2.5% speedup on a production virtual machine. The Real-Time Specification for Java requires dynamic scoped memory access checks on every reference assignment. We extend R&B to perform memory access checks in constant-time.

1 Introduction

Dynamic subtype tests are a staple of modern object-oriented programming languages. In Java subtype tests are executed in a variety of contexts, such as checked casts, array updates, exception handling and `instanceof` queries. The language runtime system is responsible for maintaining data structures to encode the subtype relation and efficiently answering queries. Subtype tests can be performed in linear time by traversing the type hierarchy. Unfortunately such implementations lead to unpredictable performance. This problem remains in many state-of-the-art implementations which have a constant-time fast path and a slow path which falls back on a form of hierarchy traversal.

The lack of predictability is particularly bothersome in real-time settings because giving the time bound of a simple instruction such as an array store requires making non-trivial assumptions about the concrete types of objects and knowledge of the implementation technique used by the VM. In this paper, our goal is to engineer a subtype test algorithm for memory-constrained real-time systems which satisfies the following requirements:

- Queries must run in constant time.
- Space overhead must not significantly increase system footprint.
- Preemption latency must be small and bounded.

In general-purpose virtual machines, such as Hotspot or Jikes, unpredictable performance is also a nuisance because it impacts programming style. For instance,

in many application subtype tests may be slower for interface than classes (usually if the number of implemented interfaces is larger than some VM-specific constant). This situation reinforces folklore about the cost of using interfaces and suggests that they should be avoided for performance critical activities.

The issue of implementation of subtype tests for object oriented languages has been addressed by many authors from both theoretical and applied communities [6, 15, 1, 7, 5, 11, 14, 18, 19, 2, 20, 17]. A number of non-incremental techniques for compact and constant time subtype tests have been proposed [7, 14, 17, 20]. Production virtual machines that implement fast incremental algorithms [2, 6] can, in special cases, exhibit suboptimal performance.

In this paper we investigate simple techniques based on well-known algorithms and strive to find a compromise between these three requirements. We reduce the amount of work needed upon class loading so that in most cases there is no recomputation; we guarantee fast and constant time subtype tests[1] and require very little space per class.

We also report on a proof-of-concept implementation of our algorithm, called R&B, in which we integrated R&B in a production virtual machine, the Sun Microsystems Research VM (or EVM). And we show that for that particular implementation we reduced space consumption and improved running times by an average of 2.5%. We had to change only about 100 lines in the original code of the VM and optimizing just-in-time compiler.

In this paper, we also discuss two extensions to R&B, one extension is a *combined encoding* which unifies the treatment of classes and interfaces, and the other extension is an algorithm for checking memory accesses. Memory access checks are mandated by the Real-Time Specification for Java (RSTJ) [4] on each reference assignment. We observe that these checks are a special case of subtype tests and that it is thus relatively straightforward to apply R&B to this problem. The technique described here has a slow path of two loads and two compares which is faster than previous work [3, 12, 8]. Moreover, we only need one word of storage per memory area.

2 Subtype tests in Java

We start by presenting the subtype test algorithm used in EVM. Subtype tests are mandated by the Java language specification for most checked cast expressions, type comparisons, array stores, and exception handler determination. In all of these cases, one of the following two primitive functions is evaluated:

`instanceof( o, T)`	returns true if `o.class <: T`
`checkcast( o, T)`	throws exception if not `o.class <: T`

We write `A <: B` to mean that type `A` is a subtype of `B`. The first function checks that an object is an instance of a given type and the second, that it is assignable

[1] Recomputation can create short pauses, but these are infrequent enough that we argue that they will not impact overall throughput. Furthermore R&B is thread safe allowing the runtime to be preempted at any time by a real-time thread.

to that type. These functions treat null values differently. On a null, an instance test returns false, while a cast succeeds. Java also defines a subtyping relation for arrays based on equality of their dimensions and subtyping of element types.

In the remainder of this paper we abstract those differences and focus on the core functionality of subtype testing as implemented by the `subtypeof` procedure of Fig. 1. EVM's type test algorithm treats class and interface queries differently. For classes, the hierarchy is traversed until the requested class is located or the root is reached. For interfaces, a per-type array of implemented interfaces is scanned. The basic scheme is optimized in two straightforward ways. The subtype test logic is guarded by an equality test so as to catch cases when both argument types are the same. To exploit type locality of tests a per-type cache is added. This cache always holds the last type that tested positively as a subtype of the given type. EVM's optimizing compiler inlines the code of `subtypeof` (but not `implements` or `extends`). The equality test in is inlined because it is cheap. The branch on the kind of the type `pr` (either a class or an interface) may be resolved if `pr` is loaded. The algorithm is simple and performs well, though in pathological cases the performance of tests may vary greatly.

```
type_info {
   type_info   parent;
   type_info[] interfaces;
   type_info   cache; }


subtypeof( type_info cl, type_info pr ) {
   if ( cl == pr || cl.cache == pr ) return true;
   if ( isInterface( pr ) )          return implements( cl, pr);
   else                              return extends( cl, pr); }


implements( type_info cl, type_info pr ) {
   for( int i = 0; i < pr.intefaces.length; i++)
      if ( cl == pr.interfaces[i] )
         { cl.cache = pr; return true; }
   return false; }


extends( type_info cl, type_info pr ) {
   for ( type_info pcl = cl.parent;  pcl != null; pcl = pcl.parent)
      if ( pcl == pr )
         { cl.cache = pr; return true; }
   return false; }
```

Fig. 1. EVM Subtype test: hierarchy traversal, one-entry cache and equality test.

3 Runtime behavior of subtype tests

A suite of twelve Java programs was used to characterize the runtime behavior of subtype tests in practice. The data was obtained by running these programs on an instrumented version of EVM. This benchmark suite is part of larger collection available from `www.ovmj.org`.

Runtime program size. While the average size of programs in the benchmark suite (inclusive of the JDK libraries) is close to ten thousand classes and interfaces, in practice a much smaller number is used – and, thus, loaded. This difference is significant because designing an algorithm based solely on static characteristics would be overly pessimistic. As can be seen, Table 1 includes counts of classes and interfaces that are loaded by the VM during the test runs, respectively in columns `#class` and `#itf`. The relatively small numbers of loaded types suggests that in a typical situation the size of data structures may not be a bottleneck. Table 1 also illustrates the average number of implemented interfaces (`avgit`) and the depth of the inheritance hierarchy defined as the number of classes between a type and the hierarchy root (`ichain`). While averages are low, it should be noted that in one benchmark there were as many as twelve implemented interfaces. The inheritance chain length suggest that the average iteration count of hierarchy traversal is less than two. We believe that the architecture of the benchmark programs accounts for some of the variations among programs. For instance `CO` has few interfaces because its implementors chose to minimize their number to avoid the (supposed) higher cost of interface dispatch. By contrast, `EH` has a clean design with a rich type hierarchy, a large number of interfaces and deep inheritance chains. Finally, it should be noted that this data is not a predictor of subtype test performance as we will see later.

	Name	Description	`#class`	`#itf`	`avgit`	`ichain`
EH	enhydra	Html2Java	313	113	2.6	2.7
CA	cap	Javac stress test	264	21	0.9	1.6
GJ	gj	Java compiler	328	27	0.7	1.6
KA	kawa	Scheme interpreter	511	19	0.6	2.2
BL	bloat	Bytecode optimizer	362	28	0.5	1.8
JE	jess	Expert shell	410	33	1.0	1.5
JA	javasrc	Html generator	232	22	0.6	1.7
XM	XML	XML tool	327	78	0.7	1.6
TO	toba	Java-to-c compiler	220	16	0.4	1.6
RH	rhino	Javascript interp.	305	21	0.6	1.7
SO	soot	Optimization fmk	641	102	1.2	2.2
CO	confined	Confinement check	467	29	0.3	1.6

Table 1. Benchmark suite. The number of classes dynamically loaded during the benchmark run is `#class`, the number of interfaces is `#itf`. `avgit` is the average number of implemented interfaces, and `i-chain` is the average inheritance chain height.

Hierarchy shape. The shape of a class hierarchy can be further characterized by the number of *runtime leaf* classes it contains. A runtime leaf class is one for which no subclass was dynamically loaded during the benchmark run. Fig. 2 shows that, on average, over 80% of loaded classes are leaves. Runtime leaf classes may, of course, have subclasses that simply were not loaded in a given run.

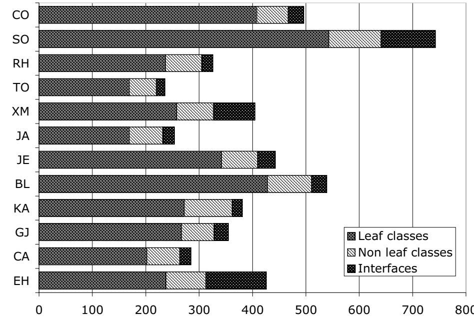

Fig. 2. Breakdown of dynamically loaded classes and interfaces. Leaf classes do not have subclasses.

Test frequencies. The benchmark programs performed an average of 320K tests per second as measured by instrumenting the optimizing JIT compiler and the interpreter and running on a SunBlade 100. Fig 3 breaks down subtype tests between `instanceof` and `checkcast`. As can be expected, casts occur more frequently in programs that manipulate generic data structures, such as the Generic Java compiler (`GJ`). At the other extreme, the bytecode analysis framework used in `CO` uses its own template macro expansion mechanism to generate container types thus avoiding casts. We have observed that the number of casts will be much higher if the optimizing compiler is turned off. This is accounted for by common programming idioms that are easily optimized by the JIT, *e.g.* the use of `instanceof` to guard a cast. Overall, the variability in test frequencies is most

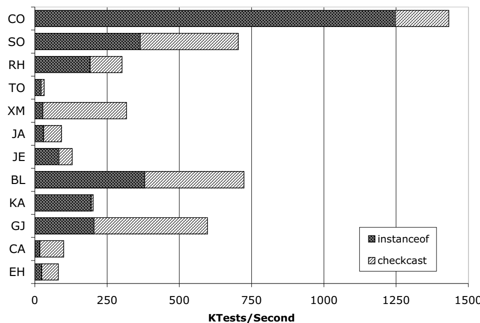

Fig. 3. Subtype tests per second.

likely due to programming style; for instance, in CO the inner loop is a visitor pattern over a complex instruction hierarchy and each iteration requires several tests. The data also shows that almost 90% of the tests are extends tests.

Test sites. The EVM JIT compiler emits an average of four hundred subtype test sequences per program. Fig. 4 shows that all but three of the benchmarks have less than 500 test sites. This is surprisingly low and suggests that code size increase due to inlining is likely to be negligible. No correlation between the number of sites and dynamic occurrences of tests could be established. Because the number of sites is so small, compiler implementors may even choose to keep track of test sites during program execution. As more information becomes available, the original code can be patched to remove the extra interface check.

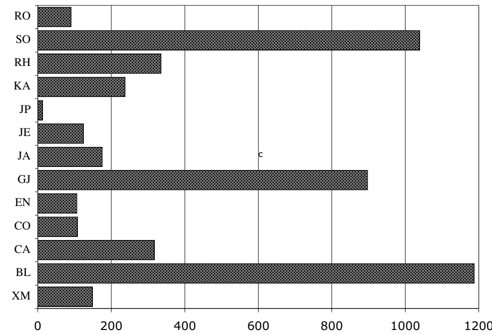

Fig. 4. Subtype tests sequences emitted by the JIT.

Success rate. A test is considered successful if it returns true (instanceof) or does not throw an exception (checkcast). Fig. 5 shows that 90% of tests are suc-

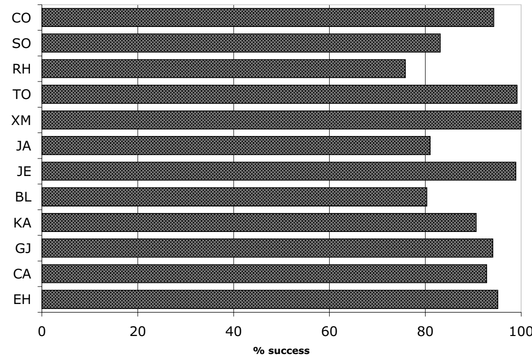

Fig. 5. Ratio of successful subtype tests. An instanceof test is successful if it returns true and checkcast if it does not throw an exception.

cessful. Clearly any implementation should optimize for success, but considering the frequency of tests the costs of the slow path can not be ignored.

Cache effectiveness. EVM uses a two element cache composed of an array cache and an object cache. The array cache is used to record the last successful cast performed on an array store, the object cache is used for all other tests. The average hit rate is 84.5%, but as can be seen in Fig. 6 these results can be highly variable. Hit rates can be as low as 50.3% (`EH`) or as high as 99.9% (`JE`). While these numbers confirm the usefulness of caches they also demonstrate that it is not a panacea.

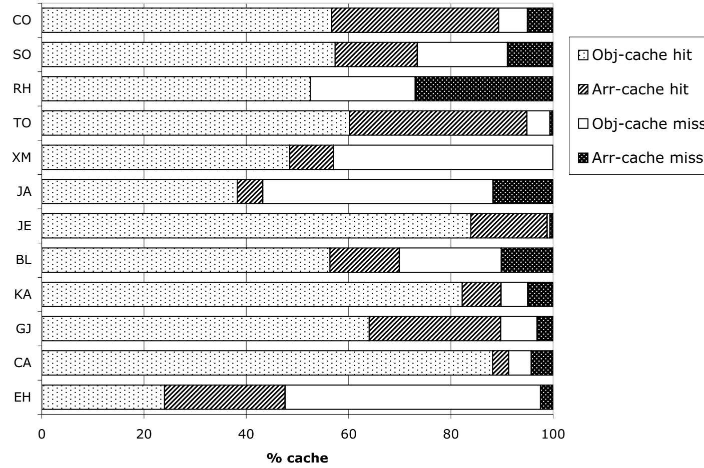

Fig. 6. Effectiveness of a two element cache. The first two bars denote, respectively, hits in the object cache and the array cache. The last two bars indicate misses in the object and array cache. Values are normalized wrt. successful subtype tests.

Miss costs. The cost of a cache miss depends on the number of comparisons required by the `implements()` and `extends()` functions. Average iteration counts, with caches turned off, appear in Table 2. The first counts the levels of inheritance traversed, the second the number of interfaces tested. The numbers are surprisingly high for some benchmarks, *e.g.*, `EH` in which an average of 7.6 interfaces are tested per cache miss. The implication is that pathological cases with important performance perturbations are quite likely to occur.

	EH	CP	GJ	BL	JE	XM	TB	RH	CO	JA
extends	1.03	1.67	1.40	3.21	1.05	1.99	0.95	1.00	1.39	2.54
implements	7.64	0.66	0.00	0.64	1.98	1.00	1.98	1.75	0.64	1.00

Table 2. Iterations of the hierarchy traversal algorithm.

Selftests. A selftest is a test of the form `subtypeof( A, A)`. On average 59% of the subtype tests evaluated in the benchmark suite are selftests. Fig. 7 also shows that the large majority of these selftests are performed on leaf classes.

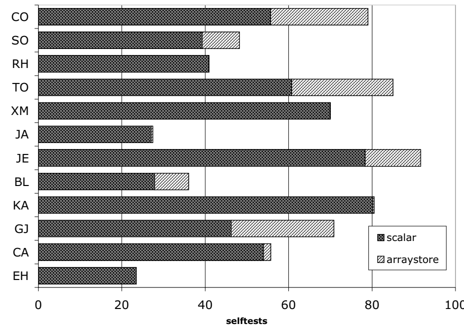

Fig. 7. Frequency of selftests. The first bar denotes the ratio of tests `subtype(A,A)` where `A` is a leaf. The second bar is the ratio of selftests for non-leaf classes.

Conclusions. Several conclusions should be drawn from this data. First, the runtime type hierarchy is more important then the compile time hierarchy. While many benchmark programs consist of many classes, the portion actually used is small. It would appear that at runtime it is common to encounter a shallow hierarchy with few interfaces. Selftests and caches are important for good performance, however they are not sufficient. Finally, there are surprisingly few test sites suggesting that code bloat is unlikely to be an issue.

4 The R&B algorithm

The algorithm presented in this paper, called R&B for *ranges and buckets*, has the following characteristics. Subtype tests are run in constant time. Caches can be added to the basic algorithm to yield different configurations, their tradeoffs are studied in Section 7. Space requirements are small. Each class and interface has a single word reserved for extends tests and, on average, 2.8 bytes for implements tests. The algorithm is incremental in nature. Type information may be recomputed eagerly (at each class load) or lazily (only when needed), depending on the requirements of the application.

To meet the responsiveness requirements of real-time systems it is essential that the algorithm be interruptible at any time, in particular, during updates to the type information because these can take several milliseconds. In R&B, updates are thread safe. The data structures are always in a consistent state and can be used at any time to answer subtype queries. The presentation of is structured as follows. Section 5 describes the range numbering scheme used for checking the inheritance relation (extends). Section 6 describes the bucketing technique used for multiple subtyping (implements).

5 Range-based extends tests

A well-known technique for representing a single inheritance relation is based on assigning a range to each type such that ranges of children are subranges of their parents' ranges and ranges of siblings are disjoint. The technique was first described by Schubert *et al.* [15] and independently rediscovered by the implementors of Modula-3.

Subtype tests are, essentially, range inclusion checks which can be computed in constant time and constant space. Assuming that each class is described by two variables called *low* and *high*, a test whether type A is a subtype of B becomes:

$$\texttt{B} <: \texttt{A} \quad \Leftrightarrow \quad \texttt{A.low} < \texttt{B.low} \ \wedge \ \texttt{B.high} < \texttt{A.high} \tag{1}$$

The range assignment is performed by a preorder walk on the inheritance tree. A consecutive number is given to each type's low bound the first time the type is encountered. The high bound is chosen so as to be larger than the maximum of the type's low bound and all of it's children's high bounds. A sample assignment is shown in Fig. 8.

Several questions need to be addressed for range-based queries to be practical.

1. How many bits are required to represent ranges?
2. How can the impact of class loading be minimized?
3. How can thread safety be ensured?

The first question is important for long running systems because these may load many more classes than the applications in the benchmark suite. It would be advantageous to pack the ranges in a single word. With a naive encoding this would

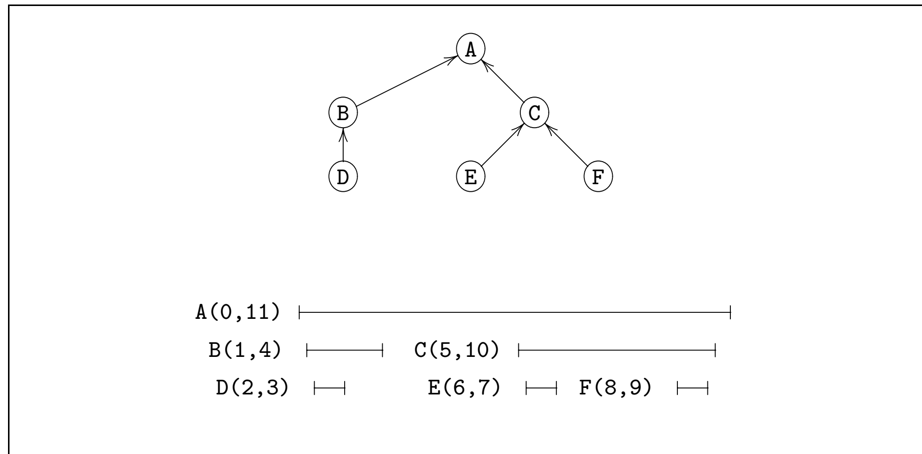

Fig. 8. A valid range assignment for an inheritance hierarchy. The class `A` is the root, all of its subclasses have subranges. Siblings such as `B` and `C` have disjoint ranges.

restrict VM to 32,768 classes. Note also, that the encoding of Section 9 requires packed ranges. Class loading is frequent and implies recomputing the range assignment each time a new class is added. The cost of computing the assignment should be minimized. Finally, in a real-time setting, a real-time thread may be released while the range assignment is in the middle of being recomputed. The algorithm should be designed so as to ensure that it can be preempted without invalidating the type information.

5.1 Refining the encoding

We now consider how to refine the encoding to address the three questions mentioned above. For a subtype query such as `o instanceof T`, we call type `T` the *provider* and the type of the object `o` the *client*. Thus we refer to the left hand-side of a test *client* <: *provider*, as the client position, and the right-hand side the provider position.

Observe that both high bounds are not needed. Thus equation (1) can be written:

$$\texttt{B} <: \texttt{A} \quad \Leftrightarrow \quad \texttt{A.low} < \texttt{B.low} \;\wedge\; \texttt{B.low} < \texttt{A.high} \tag{2}$$

The low bound is the only information required from the client. Furthermore, $\texttt{A.low} < \texttt{B.low}$ can be safely weakened to $\texttt{A.low} \leq \texttt{B.low}$ without invalidating the result. The invariants that must be maintained in cases where `B` is a subclass of `A` are, thus, $\texttt{A.low} \leq \texttt{B.low} < \texttt{A.high}$.

The key insight to limit the growth of ranges and reduce the cost of recomputing the assignment is that the high bounds can be computed on demand. As

```
type_info {
   ushort high;
   ushort low;
   type_info parent; }


extends( type_info cl, type_info pr) {
   if ( cl == pr ||
        pr.low <= cl.low && cl.low < pr.high)   return true;
   if ( invalid( pr) )
      { promote( pr); return extends( cl, pr);}
   return false; }


invalid( type_info t) { return t.high == 0; }
```

Fig. 9. Extend test.

long as the low bound has been initialized, the type can be used in the client position of subtype queries. The refined extends test function will thus perform a subrange check and, only if the test fails, will the validity of the provider be verified. If it is not valid, *i.e.* the high bound is zero, the high bound is computed — we say the type is *promoted* — and the test is attempted one more time. Fig. 9 gives pseudocode for extends tests. The `type_info` data structure contains two 16 bit values, `high` and `low`. An equality check is added because it will obviate the need to promote leaf types[2]. Tests of the form `extends( A, A)` will be shortcircuited by the equality check.

5.2 Range assignment

Every time a new class[3] is added to the system, the class is inserted in the hierarchy using the `insert()` procedure shown in Fig. 10. Thus every type starts out with an invalid range, but in a state that allows it to be used in the client position. The algorithm only recomputes the range assignment on calls to the `promote()` routine. In our benchmarks, `promote()` is called on average eleven times per program (one promotion for every 40 loaded classes, or one promotion per 13 million tests). The cost of promotion is linear in the number of types as will be shown next. Promotions can also be triggered eagerly upon class loading, in which case the extends procedure of Fig. 9 need not check for validity.

```
insert( type_info t) {
   t.high = 0;
   t.low = ( t.parent == null ) ? 1 : t.parent.low; }
```

Fig. 10. Adding a class to the hierarchy.

Consider the hierarchy of Fig. 11.a which is the result of several calls to `insert()`. All classes have the same invalid range $[1, 0]$. The first subtype test, `extends( F, C)` forces a relabeling. Fig. 11.b shows that `C` is promoted which triggers the promotion of its parent `A`. The query `extends( F, C)` can now evaluate to true as $2 \leq 2 \wedge 2 < 3$. The second test, `extends( H, C)`, will succeed without promotion. Note that children of `C` still have an invalid range since it has not been necessary to distinguish them so far. Fig. 11.c, shows the result of evaluating `extends( F, E)`. Since `E` is invalid it has to be promoted. `extends( F, E)` fails as $3 \leq 4 \wedge 4 < 4$ is false. Next, we evaluate `extends( H, H)` which is trivially

[2] The only case when leaf types have to be promoted is when testing `extends( A, B)` and `B` is a leaf. Of course, tests of this kind always fail, so there is, in fact, no need to promote a leaf.

[3] Interfaces are subclasses of `Object` but they do not need to be promoted because they can never be in the provider position.

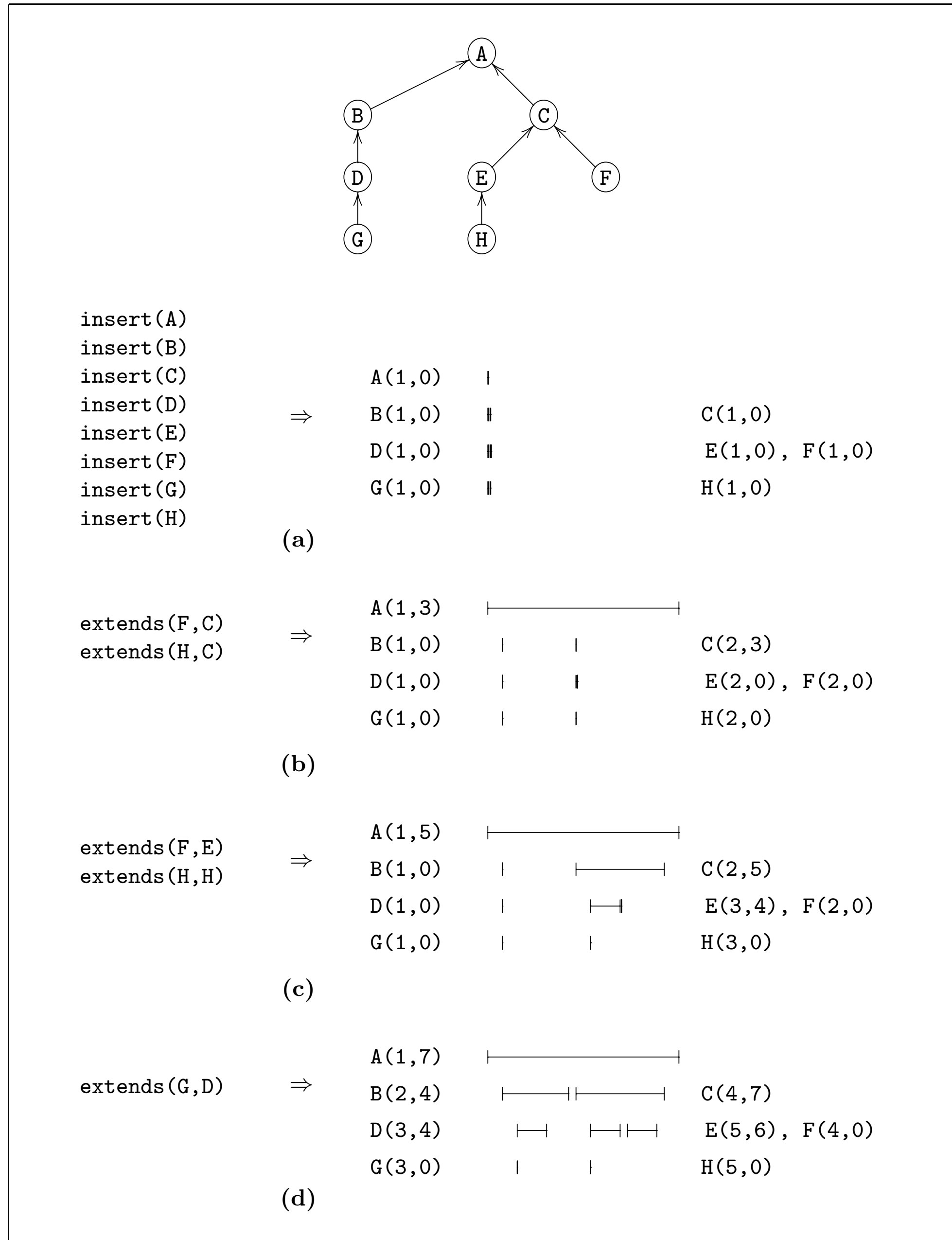

Fig. 11. Allocating bounds for a small hierarchy.

true. Finally `extends( G, D)` is evaluated in Fig. 11.d. After this step no more promotions will be required.

The pseudocode of our implementation is given in figures 12 and 13. The basic data structure representing types, `type_info`, was inherited from the EVM. It includes a reference to the parent of the class or interface, siblings and first child. Whenever `promote()` is invoked, the algorithm starts by flattening the hierarchy in an array of `entry` records (`flatten()`). Leaf classes are stored once, non-leaves are stored twice, once before all of their subclasses and once after. Ranges are allocated by increasing a counter for all non-leaf entries. This algorithm is doing slightly more than is strictly necessary because we recompute the assignment for the entire tree each time.

```
int MINRANGE = 1, MAXRANGE = 0xffff;
char FIRST='f', LAST='l', IGNORE='i';

type_info root;
entry[] array;
int array_pos;

entry {
   char type;
   type_info class;
   ushort position; }

type_info {
   ushort high;
   ushort low;
   boolean isInterface;
   type_info super, nextSibling, firstChild; }
```

Fig. 12. Data structures, constants and global variables.

5.3 Thread safety and Real-time

We now argue that the algorithm of Fig. 13 is thread safe. The `promote()` procedure maintains the following invariants. If `A` is the type information before a promotion and $\mathtt{A}'$ is the same after a promotion we have:

$$\mathtt{A.low} \leq \mathtt{A}'\mathtt{.low} \quad \wedge \quad \big(\mathtt{A.high} = 0 \ \vee \ \mathtt{A.high} \leq \mathtt{A}'\mathtt{.high}\big) \tag{3}$$

Furthermore, for any pair of type `A` and `B` we have:

$$\mathtt{A.low} \leq \mathtt{B.low} \quad \Rightarrow \quad \mathtt{A}'\mathtt{.low} \leq \mathtt{B}'\mathtt{.low} \tag{4}$$

```
synchronized promote() {
   array = new entry[SizeOfHiearchy*2];
   array_pos = 0;
   if (root.isValid()) root.setRange( MINRANGE, MAXRANGE);
   root.flatten();
   allocateRange();
   updateClasses();
   array = null; }

flatten() {
   if (firstChild == null) add( IGNORE);
   else
      { add(FIRST); firstChild.flatten(); add( LAST); }
   if (nextSibling != null) nextSibling.flatten(); }

add( char c) {
   entry e = new entry();
   e.type = c;
   e.class = this;
   e.position = 0;
   array[ array_pos++ ] = e; }

allocateRange() {
   int position = 0;
   for (int pos = 0; pos < arr_pos; pos++) {
      if (array[pos].type != IGNORE) {
         char prev = array[pos - 1].type;
         char this = array[pos].type;
         if (!(prev == LAST && this == LAST)
               && !(prev == FIRST && this == LAST))   position++;
      }
      array[pos].position = position;
   } }

updateClasses() {
   for (int pos = array_pos - 1; pos >= 0; pos--) {
      entry e = array[pos];
      if (e.type == LAST || e.type == IGNORE)
         if (e.class.low != e.position)
            e.class.low = e.position;
      else if ( e.class.high != e.position)
         e.class.high = e.position;
   } }
```

Fig. 13. Computing the range assignment.

These invariants follow from the fact that ranges are assigned in the order classes occur in the siblings list. The order of siblings is not modified during insertion because types are always added at the end of the sibling list of their parent and no other operation modifies this order.

To ensure that the data structures are consistent at every step of the update we schedule write to the `type_info` according to a preorder right-to-left tree traversal. High bounds are always written before low bounds and before any bounds of children. Consider the updates between Fig. 11.a and Fig. 11.b. Types `A, C, F, E, H` have to be updated. The following order ensures that they are valid at every step:

1. `A.high` = 3,
2. `C.high` = 3,
3. `F.low` = 2,
4. `H.low` = 2,
5. `E.low` = 2,
6. `A.low` = 1.

The state immediately after (4) may cause one to wonder about validity of the approach since we have `E(1,0)` and `H(2,0)`. However the state is still valid, since both types are still recognized as subtypes of `A(1,3)`. And any attempt to evaluate `extends( H, E)` will block until the `promote()` procedure returns.

In a real-time Java VM, a real-time thread should never have to block. This can be achieved by configuring R&B to perform eager range assignment. Every time a new class is loaded the entire hierarchy will be recomputed. This imposes an added cost to class loading, but this cost is acceptable because real-time threads are not expected to trigger class loading (unless a way is found to bound the costs of class loading).

6 Bucket-based implements tests

Our algorithm for subtype tests in a multiple inheritance type hierarchy (required for `implements` tests) is a variant of the packed encoding algorithm [17], extended to handle dynamic hierarchy extensions. Implements tests have the form:

$$\texttt{o <: I}$$

where the client `o` is an instance of some class `C` and the provider `I` is an interface type. In this approach, every interface is represented by two numbers that we call a bucket (`bucket`) and an interface identifier (`iid`). The algorithm will maintain the following invariants for any two distinct interfaces `I` and `J`,

$$\texttt{I} <: \texttt{C} \wedge \texttt{J} <: \texttt{C} \;\Rightarrow\; \texttt{I.bucket} \neq \texttt{J.bucket} \tag{5}$$

$$\texttt{I} <: \texttt{J} \;\Rightarrow\; \texttt{I.bucket} \neq \texttt{J.bucket} \tag{6}$$

$$\texttt{I.bucket} \neq \texttt{J.bucket} \vee \texttt{I.iid} \neq \texttt{J.iid}. \tag{7}$$

```
type_info {
   byte[] display; }

interface_info {
   byte iid;
   byte bucket; }
implements( type_info cl, interface_info pr) {
   return cl.display[ pr.bucket ] == pr.iid; }
```

Fig. 14. Implements test.

Our goal is to find heuristics that minimize the number of buckets. For each class the runtime system maintains a *display*, i.e., an array of `iid`s indexed by `bucket`. If a class does not implement any interfaces from a given bucket the display element for that bucket contains an `iid` of 0. The subtype test is performed by comparing the provider interface's `iid` against the value stored in the class' display at `bucket` and hence require one array access and a compare, as illustrated in Fig. 14.

This test returns the correct answer provided that any two interfaces with a common subtype are never assigned to the same bucket. Our algorithm satisfies this requirement while striving to keep the number of buckets low since the total space taken up by the displays is proportional to the number of interfaces in the system times the number of buckets. The number of interfaces in a given bucket is typically small (obviously bounded by the total number of interfaces in the system, *cf.* Table 1), hence one byte can be used to encode the interface `iid`. In case of overflow a new bucket will be allocated.

Dynamic class loading can violate the invariant that interfaces with common subtypes belong to distinct buckets because a newly loaded class may implement two interfaces which did not have, up to this point, a common subtype. In this case it is necessary to recompute the assignment of interfaces to buckets. This may be done by the algorithm given in [17], however, we present a simpler approach that gives satisfactory results in practice.

When a class `C` is loaded, we first determine if the invariant has been violated. To achieve this, we check if any of the existing buckets contains more than one interface implemented by `C`. If this is not the case, then the invariant is not violated and no recomputation is needed. Otherwise the following procedure is performed for each bucket b that contains k interfaces implemented by `C`: $k-1$ new buckets are created, one superinterface of `C` is left in b and the remaining $k-1$ interfaces are assigned to one of the new buckets. New buckets receive the next available bucket number. Subsequently the remaining interfaces from b are assigned to the new buckets so that both b and the new buckets contain approximately the same number of interfaces.

When an interface is moved from bucket b_1 to bucket b_2 its `iid` remains unchanged and the `iid` is added to b_1's *exclustion list*. If an interface is later

assigned to b_1, it will never receive an `iid` that appears on b_1's exclusion list. Once all buckets are processed we iterate over all the loaded classes and reallocate their displays. Existing entries in the displays remain unchanged and new entries are added to account for the new buckets. This means that in a given class's display an `iid` identifying the same interface may appear more than once, first at the index corresponding to its original bucket and, then, at the indices corresponding to the buckets it has been subsequently moved to.

When an interface is loaded, we have to choose which bucket to assign it to. When no buckets have been created yet or all buckets are full (i.e., all numbers within the `iid` range are used or appear on the bucket's exclusion list) we have to add a new bucket. Otherwise we choose the bucket with the fewest interfaces among m most recently created buckets and add the interface to it. Here m is a small integer constant, five in our implementation. This heuristic is based on the observation that the most often implemented interfaces such as `Cloneable` and `Serializable` are loaded early during VM initialization and, hence, have low bucket numbers. A class implementing an interface defined in the application code is likely to also implement one of the system interfaces, hence putting these two interfaces in the same bucket would likely require the bucket to be divided. On the other hand, two interfaces loaded from application code are less likely to be implemented by the same class, and can, therefore, be put in the same bucket. The `iid` of the interface is chosen to be the next number from the `iid` range that is not assigned yet to any interface in the bucket and does not appear on the bucket's exclusion list.

We now argue that the implements tests algorithm is also thread-safe. This follows from the following facts:

- provider's `iid` never changes,
- provider's bucket number is changed before displays are reallocated,
- existing entries in displays never change.

The test given in Fig. 14 can use either old values of `client.display` and `provider.bucket` or their new values, denoted `client.display'` and `provider-.bucket'`. We have the following cases.

- `client.display` and `provider.bucket`: client and provider must have been loaded prior to the test and the answer is correct;
- `client.display'` and `provider.bucket'`: correct by construction;
- `client.display'` and `provider.bucket`: the client display contains the same entry both for the old and the new bucket number, thus the test returns the correct answer;
- `client.display` and `provider.bucket'`: this case will never occur if all displays are updated before the `provider.bucket` fields are written and memory barrier is issued between these two steps.

Therefore, in all cases the test returns the same answer. In many virtual machines threads are only stopped at GC safe points. In such systems thread safety is atomic if we assume that safe points are not inserted in the middle of subtype test sequence.

7 Experimental results

We implemented our algorithm in the Sun Labs Virtual Machine for Research (EVM). EVM uses a one-element general-purpose cache and a one-element array store cache. The subtype test algorithm used by EVM is the one described in Section 2. The just-in-time compiler emits inline code to check the cache and a call to the out-of-line test routine. We made several modifications to the baseline EVM build. To evaluate the benefits of caching we removed the inlined cache checks emitted by the SPARC JIT for `instanceof` and `checkcast` tests as well as array stores (the SunNoCache configuration). We experimented with two configurations of our algorithm. In the UsJIT configuration we preserved the original cache checks emitted by the JIT but replaced the rest of the test code with an inline type equality test followed by out-of-line calls to our `extends()` and `implements()` routines. In the UsINL configuration we removed the original caches and changed the JIT to emit the fast path of our extends test. We used lazy range assignment in both configurations. We performed our experiments on a Sun Blade 100 workstation with a 500 MHz SPARC-IIe processor and 384 MB RAM.

The speedup results are summarized in Fig. 15. These were obtained because the average over five runs and are displayed as percentage speedups over the baseline EVM configuration (higher is faster). The results show that UsJIT is better on average than the baseline, with speedups up to 4.3% and on average of 2.5%. Inlining of the subtype test does not appear to pay off and performs

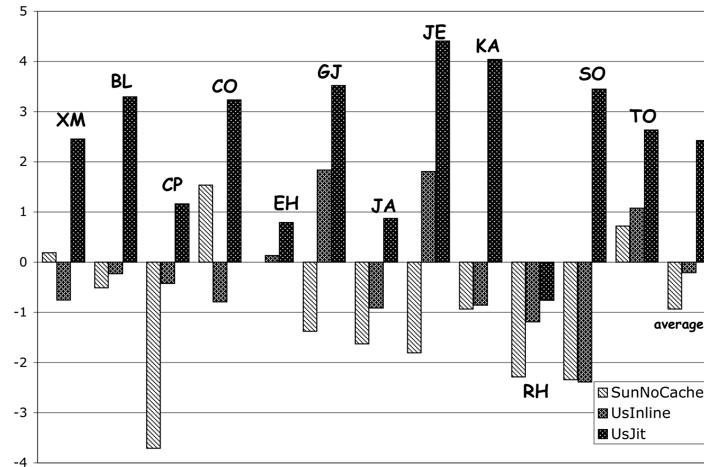

Fig. 15. Performance of the proposed subtype test techniques as percentage speedup over the baseline. The baseline is the unmodified EVM system. SunNoCache is the EVM with cache turned off. UsInline is a variant of our algorithm with inlining of the subtype test sequence and no caches. UsJit is the variant with caches. The last category shows average speed ups. UsJit exhibits a 2.5% average improvement (as compared to the base line).

slightly worse than the baseline. This clearly indicates that caches are important for performance.

The number of range promotions and bucket recomputations for each program are shown in Fig. 16. They show clearly that recomputation is infrequent. The individual costs of recomputation are given in Fig. 17. The recomputation times are small and mostly linear in the number of classes in the system.

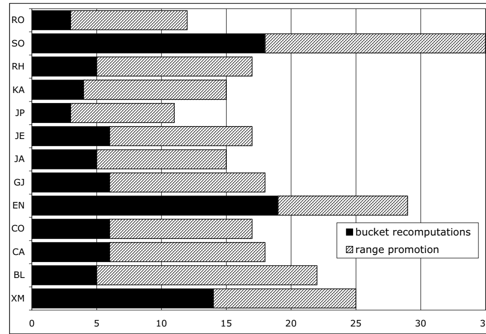

Fig. 16. Number of recomputations of implements and extends metadata. Class loading creates invalid ranges which may result in recomputations, for buckets recomputation is triggered if a new class introduces a conflict in bucket assignments.

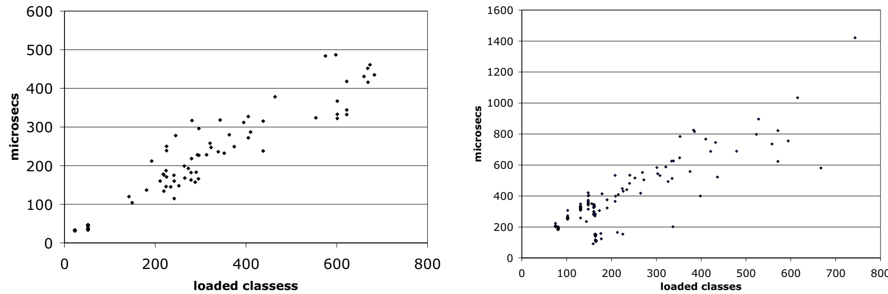

Fig. 17. Range and bucket recompute times as a function of the number of classes and interface loaded in the system at recompute time.

We have also computed the space required to store the metadata as well the code size emitted by the JIT. Fig. 18 gives the breakdown of costs. While we can not guarantee constant space for the type information data, we note that the average size required is 10.8 bytes per class of which eight bytes are fixed overhead. The increase in code size is modest. Our representation is smaller than EVM's which requires three words per class and does not share type displays.

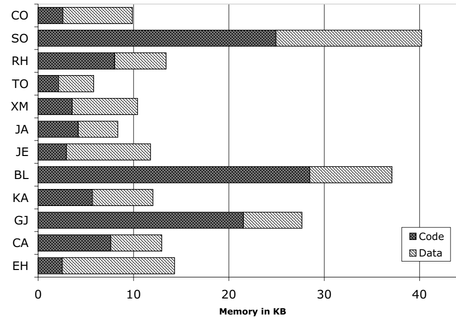

Fig. 18. Memory of requirements of our algorithm in KB. The first bar denotes the space required for the inlined test sequences, the second bar denotes subtype metadata. The average memory cost per class is 10.8 bytes for data structures.

8 Combined encoding

We now outline an alternative approach to our algorithm which we call *combined encoding*. It unifies the treatment of `extends` and `implements` checks. While not as space efficient, its implementation is somewhat simpler. Each loaded class and interface is placed in a bucket and receives an `iid` within the bucket. First, `MAX_INLINE` buckets are reserved for classes only. `MAX_INLINE` is a small constant, such as eight. If class `C` has an inheritance chain of length $d <$ `MAX_INLINE` then `C` will be put in bucket d. If $d >=$ `MAX_INLINE` the class is treated as if it were an interface. Interfaces are assigned a bucket according to the algorithm described in Section 6. Classes in the first `MAX_INLINE` buckets never change their bucket assignment. Hence, for these classes, the packed encoding technique reduces to Cohen's algorithm.

```
type_info {
   short inl_parents[MAX_INLINE];
   byte[] other_parents;
   short iid;
   short bucket; }


subtypeof( type_info cl, type_info pr) {
   buckets = ( pr.bucket < MAX_INLINE ) ?
                  cl.inl_parents : cl.other_parents;
   return buckets[ pr.bucket ] === pr.iid; }
```

Fig. 19. Combined encoding test sequence.

Fig. 19 illustrates our combined encoding. The key idea is that `inl_parents` field is a display *inlined* in the `type_info` structure. It may be padded with zero entries if necessary. Note that we use `short` integers as bucket in the first `MAX_INLINE` buckets because these buckets are likely to grow large. When the provider is known at compile time to be a non-leaf class from one of the inlined buckets, its bucket number and `iid`s are guaranteed not to change and the subtype test can be executed in three instructions (load client's display element, compare and branch).

The code can be implemented as either a runtime routine or inlined. Dynamic subtype testing using truncated and padded Cohen displays has been explored in [2, 6]. However, these algorithms did not guarantee constant-time tests.

9 Real-time Java Scoped Memory Access Checks

The Real-Time Specification for Java (RTSJ) [4], introduces the concept of scoped memory to Java. Scoped memory is similar in principle to the familiar notion of stack-based allocation that is present in languages like C, and C++ and to the region construct of ML [16]. The semantics of scoped areas are defined in [4]. The salient features are described below. After a scoped memory area is entered by a thread all subsequent allocations come from that scoped memory area. When a thread exits a scope, and there are no more active threads within the area, the entire memory area can be reclaimed along with all objects allocated within it. Scoped areas can be nested. The scoped memory hierarchy forms a *tree* as each scope can have multiple subscopes. Because a scoped memory area could be reclaimed at any time, a memory area with a longer lifetime is not permitted to hold a reference to an object allocated in a memory area with a shorter lifetime. The RTSJ further defines two distinguished memory areas, called `HeapMemory` and `ImmortalMemory` which, conceptually, act as a root to

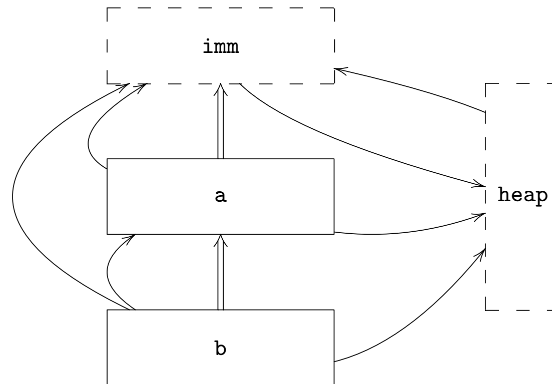

Fig. 20. Valid reference patterns. Double arrows indicate parent relations between scopes, *e.g.* `a` is a parent of scope `b`. Single arrows represent allowed reference patterns, *e.g*, a reference to `a` can be stored in a field of an object allocated in `b`.

the scope tree and are, thus, considered longer lived than all scoped memory areas. Fig. 20 gives all valid reference patterns for a scope tree composed of two scoped memory areas.

RTSJ implementations such as jRate and jTime enforce the RTSJ reference semantics by means of scope checks. These scope checks are performed each time a reference is stored in memory. Much like subtype tests, predictability in space and time is essential.

A range-based encoding can be used straightforwardly to implement dynamic scope checks. Consider the following assignment:

$$\texttt{x.f} = \texttt{y};$$

The assignment is only allowed if the region in which `y` is allocated is longer lived than the region of `x`, we write this `x.region` <: `y.region`. Assuming every object has an added `region` field holding a reference to some scoped memory area, the code of the write barrier is given in Fig. 21.

```
rx = x.region;
ry = y.region;
if ( extends( rx, ry) )
   x = y;
else
   fail();
```

Fig. 21. Scoped memory access check.

We now outline a variant of the algorithm of Section 5 adapted for implementing access checks that ensure constant time and space performance and with a test sequence short enough to inline.

Range computation. Ranges are computed eagerly when a scoped memory area is entered. When an area that contains no threads is first entered from a scoped area `S`, that area becomes the parent of the newly entered region. The same area can be entered from different parent scopes at different times, however, there can never be two parents at the same time. While eager computation may seem costly, we recall that the cost is linear in the number of memory areas and in practice we have not encountered application scenarios that would require more than a handful of them. Thus, computing the assignment is unlikely to be prohibitive or a major part of the bookkeeping associated with region management. To reduce the need for recomputing ranges, each memory area caches the range it was last assigned, a new range will be computed only if it is entered from a different parent. The following changes to the algorithm are needed:

1. Heap and immortal memory are always assigned the maximum range to allow them to reference each other and scoped memory areas to reference both of them.
2. Sparse ranges are used to limit the need for recomputing ranges.
3. Bounds of inactive scoped memory areas are cleared when recomputing the range assignment.
4. When an inactive memory area is entered, an extends check is performed against the cached range, if the extends check is successful, ranges need not be reassigned.

The last point will drastically reduce the need to recompute the range assignment because the most common RTSJ coding idiom is to use a scoped area to repeatedly perform the same computation in a periodic task. The representation of ranges is discussed next.

Scope checks. Eager range computation obviates the need to check for promotions as any newly allocated object will reside in a region with a valid range. The scope check sequence is thus made much simpler. We shorten the sequence further by packing both bounds in the same word and performing the range check in a single compare. Fig. 22 shows the code of the compact test. The check assumes that an equality test is always performed first. Furthermore, memory areas have two range fields one called `prange` constructed as `(low<<16)|high|MSK` and the other called `crange` constructed as `(low<<16)|high`. Furthermore, the `insert()` procedure of Section 5 is modified so that children are always allocated a low range value of `parent.low + 1`. This ensures that if `client <: provider` then the client's low bound is larger than it's provider.

Previous works on implementing access checks have relied on hierarchy traversal [3, 12] and Cohen's encoding [8]. Hierarchy traversal is clearly unacceptable because its performance is a function of the depth of a scoped memory area in the scope tree. Corsaro and Cytron's [8] approach has a slow path that requires three loads and three compares (this is assumes the addition of an equality test). The check outlined here is faster and more compact.

```
region_info {
   unsigned prange;
   unsigned crange; }

MSK = 0x80008000;
RES = 0x00008000;

extends( region_info r, region_info s) {
   return ( r == s ) ? true :
          ( provider.prange - client.crange) & MSK == RES; }
```

Fig. 22. Compact scope access check.

10 Related Work

Constant time (CT) techniques. The simplest constant time algorithm treats the subtype relation as a large sparse N^2 binary matrix where N is the number of types as discussed in [17]. For large programs matrices can grow to the megabyte range. Furthermore there is no clear strategy for incremental update. Despite these disadvantages, the simplicity of the binary matrix approach has motivated its use in practice [13, 9]. Attempts to reduce the space requirements of binary matrices while retaining the constant time access can be viewed as techniques for compressing the matrix.

CT for single subtyping. One particularly effective idea due to Cohen [7] is a variation of Dijkstra's "displays" [10]. Each type is identified by a unique type identifier, `tid`, which is simply a number. The runtime type information data structure also records the complete path of each type to the root as a sequence of type identifiers. The key trick is to build, for each type x, an array of $card(ancestors(x))$ type identifiers so that for each ancestor y, the `tid` of y is stored at an offset equal to $level(y)$ in the array. With this encoding, type inclusion tests reduce to a bound-checked array access and a comparison operation. The bound check is necessary if array sizes are not uniform. This approach is being used for extends checks in the Jikes RVM as described in [2] and in Wirth's Oberon.

CT for multiple subtyping. The hierarchical encoding proposed by Krall, Horspool and Vitek [14] is another constant time technique which represents each type with a set of integers chosen so that

$$x <: y \quad \Leftrightarrow \quad \gamma(y) \subseteq \gamma(x)$$

where $\gamma(x)$ maps type x to its set representation. Thus, the set of a subtype has to be a superset of the set representing its parent. This slightly counterintuitive relation allows a natural representation as bit vectors. In the bit vector representation the test function becomes $\gamma(x) \wedge \gamma(y) = \gamma(y)$, thus a type is a subtype of another if the bit pattern of the parent occurs in the child. The problem of finding optimal bit vector encodings for partial ordered sets is NP-hard [11] and there are some classes of partial ordered sets where an optimal encoding is as large as the number of types with only one supertype. The graph coloring algorithm of [14] is both fast and generates compact sets for most hierarchies (less than 32 bits). Unfortunately, there is no obvious way to support incremental recomputation. Another technique for compacting subtype hierarchies is the *packed encoding* of Vitek, Horspool and Krall [17]. In the binary matrix encoding, there is a one-to-one mapping from types to matrix indices. Each type has a column and a row of the matrix. In the packed encoding, columns for unrelated types are merged. This reuse of columns is similar in spirit to the reuse of genes in hierarchical encoding and to the levels of Cohen's algorithm. Subtype tests with packed encoding run as fast as with Cohen's algorithm, space usage

is somewhat higher because there are some empty entries in the arrays holding type ids. This technique is the basis of our treatment of interfaces. Zibin and Gil have published several recent papers improving these techniques [20].

Incremental techniques. Currently the most efficient subtype test algorithms used in production virtual machines are the ones by Click and Rose [6] and the Jikes RVM team [2]. Both are variants of Cohen displays with a slow path that may require scanning a linear list of types. The techniques are slightly more space consuming than our approach since they inline some of the metadata in the class data structure. Zibin and Gil have presented several incremental algorithms that provide alternatives to the bucketing technique used here [21]. Since the space overhead of the data for implements test is very low in our benchmarks, we have not evaluated their techniques.

11 Conclusion

In this paper, we have presented R&B, a subtype test algorithm that can perform subtype tests in constant time and which has fairly modest space requirements. R&B supports incremental modifications to the type hierarchy and is thread safe. This algorithm has all the properties required for addition to a real-time virtual machine. We have evaluated R&B on a production VM and shown that it is possible to get predictable performance and at the same time improve both time and space (though this was not our goal). On average our benchmarks ran 2.5% faster and required less memory than the baseline virtual machine. Last but not least, this was achieved without adding unnecessary complexity to the virtual machine (about 100 lines of original code were modified).

Acknowledgments This work is supported by grants from DARPA, and NSF (CCR–9734265). The authors thank Dave Detlefs for his help with EVM and EVM team for producing an excellent system. Some of the programs from our benchmark come from the Ashes suite, we thank the McGill Sable research group for making these available. Finally, we thank David Holmes, Urs Hölzle, Alex Garthwaite, Doug Lea, Bill Pugh, Michael Hind and Scott Baxter for their comments.

References

1. Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, 1989.
2. B. Alpern, A.Cocchi, and D. Grove. Dynamic type checking in Jalapeno. In *Java Virutal Machine Research and Technology Symposium*, April 2001.
3. William S. Beebee, Jr. and Martin Rinard. An implementation of scoped memory for real-time Java. *Emsoft - LNCS*, 2211, 2001.

4. Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.
5. Yves Caseau. Efficient handling of multiple inheritance hierarchies. In *Proc. Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA'93*, Published as SIGPLAN Notices 28(10), pages 271–287. ACM Press, September 1993.
6. Cliff Click and John Rose. Fast subtype checking in the HotSpot VM. In *Java Grande 02*, November 2002.
7. Norman H. Cohen. Type-extension type tests can be performed in constant time. *ACM Transactions on Programming Languages and Systems*, 13(4):626–629, 1991.
8. Angelo Corsaro and Ron K. Cytron. Efficient memory-reference checks for real-time java. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, 2003.
9. J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proc. Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA'96*. ACM Press, October 1996.
10. E. W. Dijkstra. Recursive programming. *Numer. Programming*, (2):312–318, 1960.
11. Michel Habib and Lhouari Nourine. Tree structure for distributive lattices and its applications. *Theoretical Computer Science*, 165:391–405, 1996.
12. Teresa Higuera-Toledano and Valerie Issarny. Analyzing the performance of memory management in rtsj. In *Proceedings of the Fifth International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)*, 2002.
13. Andreas Krall and Reinhard Grafl. CACAO – a 64 bit JavaVM just-in-time compiler. In Geoffrey C. Fox and Wei Li, editors, *PPoPP'97 Workshop on Java for Science and Engineering Computation*, Las Vegas, June 1997. ACM.
14. Andreas Krall, Jan Vitek, and R. Nigel Horspool. Near optimal hierarchical encoding of types. In *Proc. European Conference on Object-Oriented Programming, ECOOP'97*, Lecture Notes in Computer Science. Springer-Verlag, June 1997.
15. M. A. Schubert, L.K. Papalaskaris, and J. Taugher. Determining type, part, colour, and time relationships. *Computer*, 16 (special issue on Knowledge Representation):53–60, October 1983.
16. Mad Tofte and Jean-Pierre Talpin. Region based memory management. *Information & Computation*, 132(2):109–176, February 1997.
17. Jan Vitek, Andreas Krall, and R. Nigel Horspool. Efficient type inclusion tests. In *Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'97*, October 1997.
18. Niklaus Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, 1988.
19. Niklaus Wirth. Reply to "type-extension type tests can be performed in constant time". *ACM Transactions on Programming Languages and Systems*, 13(4):630, 1991.
20. Yoav Zibin and Joseph Gil. Efficient subtyping tests with PQ-Encoding. In *Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'01*, October 2001.
21. Yoav Zibin and Joseph Yossi Gil. Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA-02)*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 142–160. ACM Press, November 4–8 2002.