# Porting the JMTk memory management toolkit

Robin Garner
Department of Computer Science
Australian National University
Canberra, ACT, 0200, Australia
u3401953@student.anu.edu.au

## ABSTRACT

JMTk is a memory management toolkit written in Java, originally for the JikesRVM Java Virtual Machine. JMTk provides reusable components for the development of allocation and garbage collection algorithms, an efficient implementation of a rapidly growing number of memory management schemes, and is an important enabler for memory management research. A portable version of JMTk will allow the growing body of memory management research done using it to be repeated in different language environments, and to study the different performance characteristics of algorithms across programming languages without the variation in implementation technique that clouds such issues today.

This paper describes the process of porting JMTk for use in C/C++ based language runtimes. The portable system uses the gcj ahead-of-time Java compiler and a source transformation technique to produce a version of JMTk that can perform memory management for traditionally compiled systems. It is currently running in a testbed environment, where it provides a comparison between accurate and conservative garbage collection techniques. Incorporation into a Haskell and C# language runtime are in progress. The source transformation tool is currently in use by a team of researchers integrating JMTk into the OVM Java Virtual Machine.

This paper describes the architecture and construction of the ported system, the source code transformation process used, and provides the performance results obtained so far.

## 1. INTRODUCTION

Automatic memory management (garbage collection) [12, 17] is an increasingly important field of computer science, as languages such as Java and C# become more popular in commercial applications. While a great deal of research has taken place since McCarthy's LISP system introduced garbage collection in 1960 [13], few of the known algorithms have been directly compared without significant variation in the implementation details of the collector. JMTk is a garbage collection toolkit originally developed for Jikes RVM and described in [4], and has facilitated the first 'apples to apples' comparison of a wide range of collectors in the same environment [3]. This paper describes the process of porting JMTk so that it can be used from C/C++-based language runtime environments. It summarises work done for an Honours project—more details can be found in [8].

### 1.1 Implementing memory management

After more than 40 years, garbage collection is still an active area of research. One of the reasons for this is that the relative performance of the techniques used are dependent on fine details of language and program behaviour, and subtle variation in implementation technique. Most garbage collectors are implemented in monolithic C, and are tightly coupled with the language runtime environment for which they are designed, making portability difficult. The construction of a garbage collector requires significant software engineering effort, and so language implementors are generally restricted in the number of GC schemes they can implement. Performance comparisons of a number of algorithms in the same environment are rare, and comparisons of the same algorithm across a number of environments are nonexistent.

A high performance portable toolkit for building memory managers, that language implementors could use 'off the shelf' would help to address this gap. JMTk provides a flexible toolkit for Java virtual machines, and is currently in use in Jikes RVM and OVM, and making this more widely available should facilitate memory managemnt research beyond the Java virtual machine environment.

Previous automatic memory management toolkits include the UMass Language Independent GC Toolkit [11] and GCTk [5]. For a comparison of the various memory management toolkits, see Blackburn et al. [3].

### 1.2 Construction of JMTk

The first issue that a systems-level program written in Java needs to tackle is how to represent the low-level machine-dependent features from which Java is designed to be free. Two broad approaches are available in doing this: the first is to use Java's primitive types to represent equivalently sized low-level objects, and to use native methods to call 'down' to unsafe languages such as C; and the second is to extend the Java language. Jikes RVM and OVM take the second approach, but use idiomatic Java [7] rather than syntactic extensions to implement the low-level features.

JMTk follows the Jikes RVM idiom [2, 1]. Five special types are recognized by the compiler, and are used to represent pointers, raw words of memory etc. The canonical example is the `VM_Address` class, which represents an address in the underlying machine architecture. From the perspective of a programmer in Jikes RVM, a `VM_Address` is a slightly restricted kind of object, with methods for address arithmetic and conversion to and from the types `Object` and `int`. When the Jikes RVM compiler processes objects of this class, they are represented as address words of the appropriate length, and the methods are compiled to optimal machine code

sequences for address arithmetic etc.

The main components of the JMTk idiom are:

- Magic classes. The classes `VM_Address`, `VM_Offset`, `VM_Extent` and `VM_Word` represent raw addresses, memory offsets, memory ranges and raw memory contents respectively. These classes are required to be represented as the corresponding hardware-specific type (i.e. *unboxed*), but have methods for operations such as address arithmetic, bit pattern manipulation etc.
- The `VM_Magic` class. This class contains static methods that directly perform low-level operations, such as memory barriers, unsafe type casts and raw memory access.
- Compiler pragmas. Jikes RVM implements pragmas with suitably named `implements` and `throws` clauses. These pragmas control compiler inlining, code atomicity and compiler optimization.

JMTk is constructed as a set of core classes that are virtual machine independent, and an interface layer (the `vmInterface` package) that adapts the virtual machine to JMTk and vice versa.

## 1.3 GCC and GCJ

The target compiler for the porting process is gcj, the GNU Compiler for Java. This is an ahead-of-time compiler for the Java language that uses the gcc back-end for code generation. Gcj was used without modification, which dictated that we use the first of the options described above for handling low-level features that Java is not designed for. This naturally leaves the problem of how to translate from the JMTk Java idiom into the native Java format, and the solution was the source code transformation described later in this paper.

Gcj produces object code that can be linked with systems compiled with languages in the gcc family, giving a large selection of target language runtimes. Gcj provides two native interfaces: the first is the standard JNI C interface; and the second is CNI, which provides close integration with C++.

## 1.4 Accurate and conservative collection

Conservative garbage collection [6] is designed for use in languages that provide no support for garbage collection, such as C and C++. The Boehm-Demers-Wieser collector (known increasingly as simply the Boehm collector) has an appealingly simple interface (simply allocate memory through `GC_MALLOC`), provides performance that is generally as good or better than explicit memory management, and is available for free download. What is interesting is that the Boehm collector is now being used by compilers for languages such as Scheme, Java, Dylan and Mercury that are suitable (from a language design point of view) for accurate collection.

One of the reasons that a conservative collector is being used is that many of these languages compile using gcc, which has no support for the stack scanning required by accurate garbage collectors. Henderson [10] has recently developed a source code transformation technique that enables accurate garbage collection in gcc-compiled code, but the only other (gcc-based) alternative has been to implement a virtual machine, and use its stack, registers etc.

Despite the widespread use of the Boehm collectors for these languages, little is known in practice about its performance relative to accurate garbage collection techniques. Henderson [10] reports that an accurate semi-space collector with his 'shadow stack' transformation has similar performance (for Mercury programs) to the Boehm collector. While the JMTk port is not yet running at its expected performance peak, some further comparisons can be made here.

## 1.5 Testbed environment

As an interim step in the process of incorporating JMTk into a new language environment, a C testbed system was developed that allows the portable code to be tested and evaluated independent of any target runtime system. The testbed environment will be of long term interest in JMTk development, as it will significantly simplify the testing and debugging process for JMTk collectors. The testbed provides a simplified version of a language runtime environment (designed from experience with the Haskell runtimes), and implements several 'point' tests of JMTk functionality, and some more complex tests that also serve as benchmarks. For benchmarking the testbed provides a 'test harness' that iterates over a test and reports timing results. Five of the JMTk collectors are currently running in the testbed, and the performance results in this paper were produced in this environment.

# 2. THE PORTING PROCESS

Porting JMTk required 3 steps: refactoring of the JMTk code base to isolate virtual machine dependent code in the interface layer; transformation of the source code to a form that gcj could compile; and building a new interface layer for the target machine, in the first case the testbed environment.

The Jikes RVM approach is possible because the project 'owns' the Java Virtual Machine. In this project we determined to use the standard gcj compiler, and so the JMTk source has been modified in a systematic way to transform it into something that a vanilla Java compiler would accept. Note this is not a one-time transformation producing a separate code base to the Jikes RVM version of JMTk, but one that is simply applied as a stage in the compilation process.

## 2.1 Factoring

JMTk was designed from the beginning to be portable, although unavoidably the initial implementation was found to be considerably Jikes RVM specific when porting work began.

The first, and most significant step in factoring JMTk was to identify JMTk's requirements of the underlying virtual machine from the services it provides to the virtual machine, ie the 'incoming' and 'outgoing' interfaces. With this distinction made, the `vmInterface` package can be seen as consisting of 4 separate interfaces: 1) the services provided to the underlying virtual machine, implemented in terms of 2) the public classes and methods of JMTk proper, and 3) JMTk's requirements from the virtual machine, implemented in terms of 4) the services offered by the virtual machine.

Despite being designed to be portable, the core JMTk code had several Jikes RVM dependencies when porting began. Three strategies were used to remove these dependencies: vectoring static method calls through the main 'outgoing' interface class, `VM_Interface`; moving classes into the `vmInterface` package; and re-implementing functionality provided by Jikes RVM in an internal class.

## 2.2 Source code transformation

In order to translate between the idiomatic Java in which JMTk is implemented, and the more traditional native approach to low level features, a source code transformation tool was built. The required transformation is:

- Objects to `int`s. The various machine word types, `VM_Address`, `VM_Word` etc. are transformed to `int` or `long` depending on the address width of the target architecture.
- Instance methods to static methods. An `int` in Java cannot have a method, and so the instance methods of the types

transformed above must be transformed to some other construct that will perform the required operation. As an initial step, they were transformed to static methods of a class corresponding to the original type. For example, the code sequence

```
VM_Address x = method_x();
method_y(x.add(4));
```

becomes

```
int x = method_x();
method_y(VM_Address.add(x,4));
```

- Pragma exceptions are deleted. The Jikes RVM idiom uses phantom exceptions of the form

  ```
  some_method() throws VM_Pragma<xyz> {
  ```

  These exceptions are treated as directions to the Jikes RVM compiler as to how to generate code for the method. The transformed code drops these exceptions (although it may do something with at least some of them in the future).

- Renaming methods. Method names in Java classes can be overloaded, provided the overloaded methods take different parameter types. JMTk contains at least one class that has more than one method with a magic machine-word type. After the transformation of types described above, all of these methods take `int` parameters, and violate Java's overloading restrictions. The solution is to change the name of the method to reflect it's pre-transformation type.

Because transformations are applied based on the type of a method call and are required to swap instance variables for parameters etc, the transform utility needs to analyse the source program beyond the simple syntactic level. A simple filter based on `awk` or `sed` would be inadequate for the task. The transform tool was built using the Antlr parser generator [14, 15].

## 2.3 The testbed

The testbed is written in C, with an interface to JMTk in C++. The two features that the testbed must provide to JMTk in particular are an object model (in particular how to scan for pointers in heap objects), and a root set. Objects are laid out as a header of at least 4 bytes, followed by zero or more pointer words, then zero or more non-pointer words. The root set is a simple array, that is generally treated as a stack by testbed code.

Programming in the testbed environment is somewhat ideosyncratic because the language provides no support for garbage collection. It is therefore the programmer's responsibility to ensure that all heap pointers are held stably in roots across possible GC points.

# 3. TUNING AND OPTIMIZATION

Initial performance results observed during functional testing were poor, and it was clear soon after the port was running in the testbed environment that work was needed to improve it. In order to establish a baseline for performance tuning, two variants of the testbed were implemented, firstly using a pure 'bump pointer' allocator, and secondly using the system-supplied 'malloc' function[1]. Section 4 evaluates performance of the tuned collectors against each other and against other systems and describes the benchmarks and benchmarking methodology, while this section describes the steps taken to tune the collectors.

The following changes improved the performance of the testbed:

[1]This was on Linux with glibc 2.2, so this is a recent version of the Lea allocator.

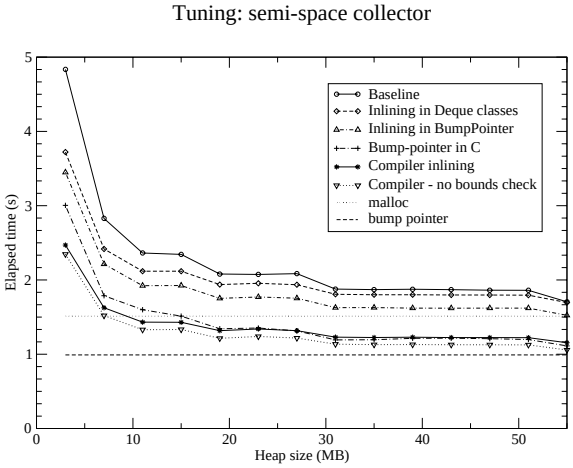

Figure 1: Tuning results

1. Replacement of `VM_xxx` calls in Deque, LocalDeque and VMResource classes.

   The first two classes are the supertypes of a family of double-ended queues used extensively during garbage collection. The body of the methods consists mainly of arithmetic on various magic types, and their poor performance appeared to be due to the method invocations involved. The other class is fundamental to managing regions of virtual memory. Replacement of the method invocations with the appropriate arithmetic operations was performed, effectively inlining the method calls at a source code level.

2. Inlining of `VM_xxx` methods in the BumpPointer class. This optimization pass was targeted at the allocation path, performing inlining as above.

3. Duplicating the Java code that represents the 'fast path' of the bump pointer allocator into the C++ interface code, replacing the call `JMTK::allocate(bytes)` with the function[2]

   ```
   char *buf, *bufEnd;
   inline char *bp_alloc(int bytes) {
     if( buf + bytes > bufEnd ) {
       buf = JMTK::allocate(CHUNK);
       bufEnd = buf + CHUNK;
     }
     buf += bytes;
     return buf - bytes;
   }
   ```

   The principal benefit of this optimization is that the language boundary between C++ and Java needs to be crossed less frequently.

4. Using the cross-class inlining provided by gcj 3.3 and above, by compiling the testbed application and JMTk on a single command line with the `-O3` flag.

5. Using the compiler option to eliminate array bounds checks.

The net result of these optimizations was a 51% speedup in the smallest heap size, and a 38% speedup in the largest. These results are shown in Figure 1. Note that in the largest heaps, performance approaches that of a pure bump-pointer allocator.

[2]The actual function is more complex than this, but this gives the essential idea.

One note of interest is the small amount of code optimization required to achieve this speedup. This justifies the 'optimize later' approach adopted[3].

# 4. RESULTS

Currently, 4 of JMTk's 7 collectors are running in the C testbed environment. Performance results were obtained using synthetic benchmarks in the testbed environment.

The restrictive programming model of the testbed environment dictates that any benchmark be a simple artificial benchmark. Two benchmarks have been used: the first (known as FTree) manipulates a persistent binary tree in the way that a functional language would, by allocating new nodes rather than mutating existing ones. Three variants of FTree were measured, using varying amounts of non-pointer 'payload'—these are called FTree($n$), where $n$ is the number of bytes of additional data per node. The second benchmark is a list-based quicksort (known as QSort), which allocates new objects when partitioning and appending lists.

In addition to measuring the JMTk collectors, the Boehm collector in the testbed environment was measured, providing a direct comparison between accurate and conservative collectors in the same environment. Graphs of the Boehm collector are included with those of the JMTk collectors.

Finally, the testbed was compared with other Java virtual machines.

## 4.1 Methodology

Results of the performance tests are graphed in terms of elapsed time (best time of 5 iterations) at a range of different heap sizes. For the cross-platform tests, the minimum heap size that each benchmark will run in varies from collector to collector, the leftmost point of each curve represents performance at the minimum heap size for that collector. The x-axis of the graph is given in terms of MB in excess of the minimum for each collector. For example, if the minimum heap for the semispace collector is 3MB, for gcj is 6MB and for Jikes RVM is 15MB, the point '8' on the x-axis represents an absolute heap size of 11MB, 14MB and 23MB respectively[4]. The minimum heap size for all benchmarks run in the testbed is kept constant across collectors, and actually represents the minimum heap in which the semi-space collector would run. The x-range of the FTree(0) and QSort benchmarks were chosen so that no garbage collection was performed at the rightmost edge of the graph. The remaining FTree benchmarks were too large to do this in the main memory of the test platform, so they were graphed on the same scale as the base FTree benchmark.

Times given are wallclock times for the kernel of the benchmark. The benchmark is run 5 times with a garbage collection requested between runs, and the best time of the 5 is used. Measurement was done on a 433MHz Intel Celeron system, with 256MB memory, running RedHat Linux 8.0.

## 4.2 Testbed measurements

The performance of the testbed collectors was measured with 4 benchmarks: FTree($n$) where $n = 0, 40$ and $80$ [5], and QSort. Graphs of the benchmarks are given in Figures 2 through 5.

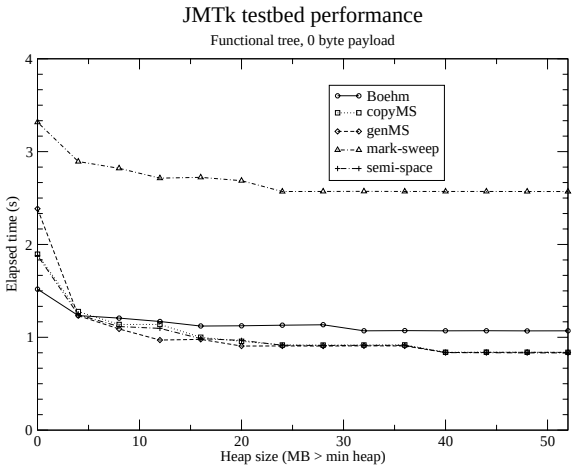


Figure 2: JMTk testbed, FTree(0) benchmark

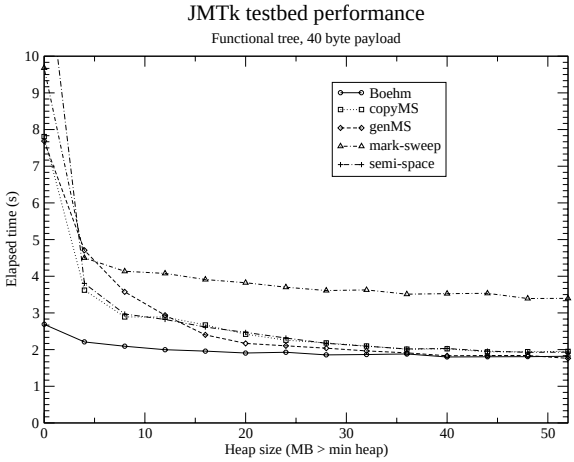


Figure 3: JMTk testbed, FTree(40) benchmark

Things of interest to note from these graphs:

Mark-sweep is consistently the worst performing collector. The mark-sweep collector is the only one to use JMTk's free-list allocator—the other JMTk collectors use a bump-pointer, part of which is 'inlined' into the C testbed code. The difference in cost for mark-sweep appears to be the cost of crossing the language boundary into the Java code for each allocation. In Figure 2 and Figure 5, the performance of the mark-sweep collector is significantly worse than the other collectors, but the difference is less noticeable in the other graphs, which have larger object sizes. This is consistent with there being a per-object overhead which diminishes in significance as per-byte costs increase with the larger object size. The performance of the genMS collector, which allocates its mature space with the same class (especially visible in Figure 4) indicates there is nothing wrong with the performance of the free-list allocator when called from Java code.

[3]We took Knuth's advice that "premature optimization is the root of all evil".

[4]Blackburn et. al. [3] use multiples of the minimum heap size. I found that when comparing systems with different fixed overheads this gave collectors with high heap requirements an unrealistic advantage. For example comparing the Sun Java VM against Jikes RVM with 10 different heap sizes, the rightmost point on the graph would represent 10MB and 150MB respectively.

[5]The mean object size in Jikes RVM for the Spec Jvm98 benchmark suite is 32 bytes, and 98% of objects are 96 bytes or less. The objects used in these benchmarks are 16, 56 and 96 bytes respectively.

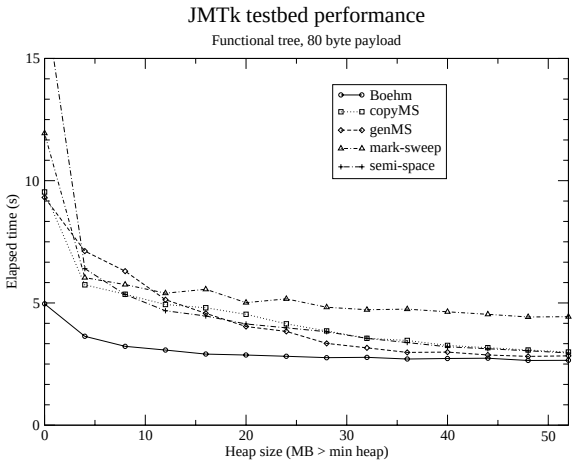


Figure 4: JMTk testbed, FTree(80) benchmark

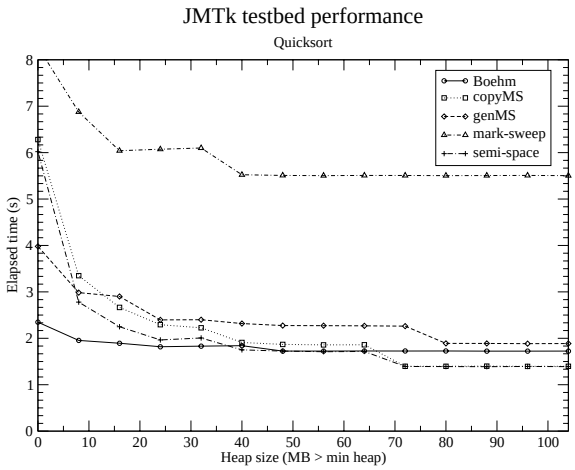


Figure 5: JMTk testbed, QSort benchmark.

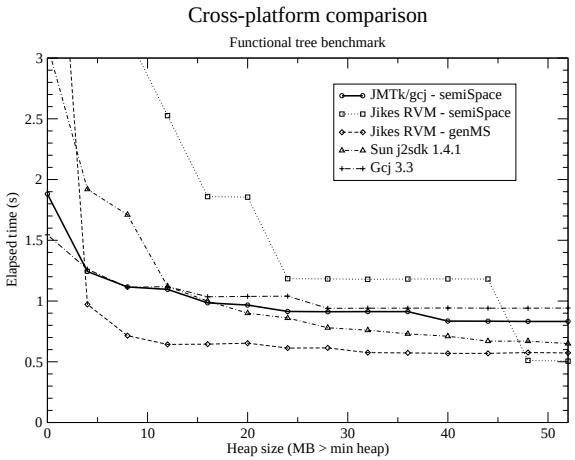


Figure 6: Cross platform comparison: FTree(0) benchmark.

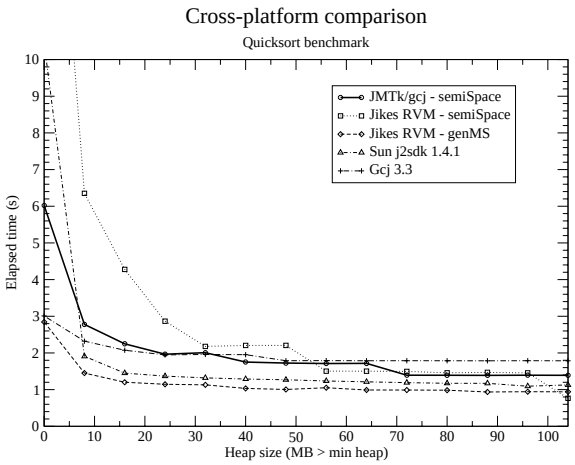


Figure 7: Cross platform comparison: QSort benchmark.

The Boehm collector also uses a segregated free-list allocator, but optimized over the 16 years since the collector was first developed. It should be possible for the JMTk free-list allocator to achieve a similar level of performance.

In small heaps, and where the object size is large, the Boehm collector outperforms the JMTk collectors. With larger object sizes, the overhead of copying collectors is proportionally higher, and the mark-sweep algorithm used by the Boehm collector obtains more of an advantage. Also, the bump-pointer 'cache' of the JMTk collectors is less effective when objects are larger, and the slow allocation codepath is called more frequently. The size of the buffer used to cache the bump pointer is currently limited by the JMTk allocation interface which automatically allocates large objects in the large object space. By exposing a little more of the JMTk internals, performance of the bump pointer could be improved.

Performance of these collectors does not match the relative performance of the JMTk collectors in Jikes RVM as shown in [3]. One factor in this may be the choice of benchmark, but the results of the next section (where two collectors running in Jikes RVM are compared) don't seem to support that hypothesis.

The most likely explanation is that in the tuning effort, the semi-space collector received more attention than any other collector, and there is still significant tuning to be done.

Another, less welcome possibility is that semi-space performs well because it is a very simple collector. The generational collectors perform less well than in Jikes RVM because the gcj compiler is unable to perform the same amount of optimization as Jikes RVM, and that complex, well-structured code (from an OO-design point of view) will not perform well under gcj. If this is true, then significant effort will need to be expended to make JMTk competitive with hand-tuned C implementations, and may require considerable work.

## 4.3 Cross-platform

Figures 6 and 7 show the relative performance of the testbed with a semi-space collector and 4 other Java systems: Jikes RVM running the JMTk semi-space collector; Jikes RVM running the genMS collector; gcj 3.3 running the Boehm collector; and the hotspot Java virtual machine from the Sun Java SDK 1.4.1, which uses a generational copying collector.

The basis of comparison is problematic. While there are bases for comparison for all systems, there are also substantial differences. The comparison of the testbed with Jikes RVM seems on the face of it to be the most valid, as this is the exact same collector. On the other hand, while the heap in the testbed contains only the data structure of the benchmark, the Jikes RVM heap also contains considerable amounts of the code and support data structures for Jikes RVM and the benchmark itself. This impacts significantly on the cost of doing collection in the semi-space collector. This overhead could be ameliorated once Gilani's work [9] is incorporated into JMTk, and the Jikes RVM code can be separated into a

| Configuration | Minimum heap (MB) | |
|---|---|---|
| | FTree | QSort |
| JMTk testbed | 3 | 7 |
| Jikes RVM semiSpace | 15 | 15 |
| Jikes RVM genMS | 12 | 16 |
| gcj | 6 | 9 |
| Sun Hotspot | 1 | 3 |

Table 1: Minimum heap sizes for cross-platform comparisons

different heap region to the application.

The performance of the Jikes RVM semi-space configuration at the maximum heap sizes, however, shows how efficient JMTk can be in its native environment, providing a 'best case' for the port. This on the other hand is not as straightforward as it sounds: the mutator code in Jikes RVM has assistance from the compiler: heap pointers can be held and passed as parameters in registers (for example) rather than stored stably in memory across all points where GC could possibly be initiated.

The Jikes RVM genMS collector shows how well JMTk collectors can perform in their native environment: this collector does not suffer as much as the semi-space collector from the overheads of Jikes RVM's Java-in-Java implementation.

The gcj comparison is quite natural, as it compares two implementations of the same algorithm with essentially the same compiler technology. The result of this comparison shows that the testbed platform is a comparable programming environment to Java with gcj.

The Sun Hotspot JVM is probably the least justifiable comparison, but it does provide a sanity check with commercial Java virtual machine technology. Although it is consistently marginally slower than Jikes RVM's genMS collector, it is the second fastest for most heap sizes, and runs in the smallest heap size of all collectors. The collector used is a generational copying collector.

Despite the difficulties discussed above, the graphs of performance across machines give an interesting picture of where the portable JMTk implementation fits with respect to other systems. A valid summary might be that it is respectable, but has some room for improvement, particularly in small heaps.

## 5. FUTURE WORK

The source code transform technique introduces several performance improvement possibilities, including: source-level inlining, both in the transformed methods and of straight JMTk code; specialization of methods (to promote inlining and eliminate dynamic dispatch); and 'out-lining' of 'hot' code so that optimizations such as the bump-pointer might be done automatically.

A broader set of benchmarks would be useful, but defining a set of synthetic benchmarks that can predict real-world application performance is notoriously difficult. One possiblity is to develop a trace-driven benchmark, that would also be useful in reproducing errors.

The future of JMTk as a portable memory manager has just begun: there are many language runtimes that could benefit from JMTk once we can demonstrate a convincing performance advantage over conservative techniques. Work is underway to integrate JMTk into the ghc Haskell compiler, and the Rotor C# environment.

## 6. CONCLUSION

JMTk can now claim to be a portable memory management toolkit, although it cannot yet claim to achieve high performance in all environments.

Source code transformation has proven to be an effective technique for porting of systems-level Java code. The transform tool is currently in use, translating JMTk's Java idiom for the port to the OVM Java Virtual Machine, which uses its own Java idiom.

The testbed environment provides a direct comparison between accurate and conservative garbage collection techniques. The JMTk port is not yet mature enough for this to be conclusive, and the question of the validity of synthetic benchmarks will continue to be an issue, but labguage runtime implementors should soon be in a position to know empirically what the costs are of conservative collection.

## 7. REFERENCES


[1] B. Alpern, C. R. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, T. Ngo, M. F. Mergen, J. C. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '99)*, pages 314–324, 1999.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, T. Ngo, M. F. Mergen, J. C. Shepherd, and S. Smith. The Jalapeño virtual machine. *IBM System Journal*, 39(1):211–238, Feb. 2000.

[3] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *Submitted for Publication to SIGMETRICS*, 2004. http://www.cs.utexas.edu/users/mckinley/papers/jmtk-sigmetrics-submit-2003.ps.gz.

[4] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with JMTk. In *Submitted for Publication to ICSE*, 2004. http://www.cs.utexas.edu/users/mckinley/papers/jmtk-icse-submit-2003.ps.gz.

[5] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Berlin, Germany, June 2002. ACM.

[6] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[7] C. Flack, T. Hosking, and J. Vitek. Idioms in ovm. Technical Report CSD-TR-03-017, Department of Computer Sciences, Purdue University, 2003.

[8] R. J. Garner. Honours thesis. A.N.U., http://eprints.anu.edu.au/, 2003.

[9] F. Gilani. Title unknown. Master's thesis, Australian National University, 2003.

[10] F. Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the third international symposium on Memory management*, pages 150–156. ACM Press, 2002.

[11] R. Hudson, E. Moss, A. Diwan, and C. F. Weight. A language-independent garbage collector toolkit. Technical Report TR 91-47, University of Massachusetts at Amherst, 1991.

[12] R. Jones and R. Lins. *Garbage Collection*. John Wiley and Sons, 1996.

[13] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):173–197, Apr. 1960.

[14] T. Parr. ANTLR: Another Tool for Language Recognition, 2002. `http://www.antlr.org/`.

[15] T. J. Parr and R. W. Quong. Antlr: A Predicated-LL(k) Parser. *Software—Practice And Experience*, 25(7):789–810, July 1995.

[16] P. R. Wilson. Uniprocessor garbage collection techniques. In H. Baker, editor, *Proceedings of the International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, Sept. 1992. Springer-Verlag.

[17] P. R. Wilson. Uniprocessor garbage collection techniques. Expanded version of [16], 1994.