A Java™ [1] Implementation of a BDD Package

Christopher D. Krieger[2]
University of Utah

Abstract. Efficient manipulation of Boolean functions is the foundation of many formal verification tasks. Several code libraries and development environments exist in the public domain which perform these tasks using *ordered binary decision diagrams* as the central data structure. This paper describes a re-implementation of one of these libraries, David Long's *bddlib*, in Sun Microsystems' Java computer language. Java offers many advantages over traditional C and C++ code in the areas of stability, memory management, portability, clarity, and code reuse. However, with the present generation of Java virtual machines, execution is significantly slower than comparable efficiently compiled native code.

1 Introduction

From a high-level perspective, there are many different methodologies for specifying formal verification problems. Included in these are various temporal logics, trace theories, Petri net models, and automatas, to name a few. Despite their theoretical differences, many of these disparate semantics can be implemented using *ordered binary decision diagrams* (OBDDs), a type of directed acyclic graph which has been motivated largely by the work of Bryant [1]. The fundamentals of these structures are covered elsewhere [2]. Because of this widespread applicability of OBDDs to logic and verification, considerable research has been conducted to optimize a computer's ability to represent and manipulate OBDDs [3, 4, 6]. David Long has released the fruit of his efforts in the form of a library package called *bddlib,* which has become a standard [5]. This paper discusses the merits of Java as a platform for verification tools and provides details about the conversion of the *bddlib* package to Java. Timing and memory usage is compared with that of the C language version of the package. Modifications for taking advantage of Java's strengths are also discussed.

[1] Java and related terms are trademarks of, or are copyrighted by, Sun Microsystems, Inc. or its subsidiary JavaSoft Inc.

[2] This research is made possible through the generous support of Sheldon Instruments, Inc.

2 Applicability of Java to verification programs

Java is an object-oriented programming language based on C with similarities to C++ [7]. However, there are many significant differences which influence its use in verification applications.

Java source code compiles to an intermediate object code format called *bytecode.* Bytecode is then interpreted at run time by an emulator program called a Java *virtual machine.* This additional level of abstraction provides several advantages. First, a Java program presented as bytecode will run without modification on any computer with a Java virtual machine. This eliminates the need to distribute source code with complex, compiler-specific makefiles or separate binary files for each unique architecture. It also allows one to provide tools in a machine-independent format, but still protect the proprietary nature of internal algorithms and techniques.

An additional advantage that the virtual machine provides is that of memory management. Under Java, the virtual machine is responsible for all memory allocation and garbage collection, somewhat like the Lisp interpreter [8]. This can eliminate large portions of programs which were previously required to perform this function. Approximately 10% of *bddlib*'s code deals directly with garbage collection, and was eliminated completely from the Java version.

Structures such as vectors and hash tables are provided as part of the standard Java library, which further reduced the source and object code length. Using standard structures also increases the understandability and robustness of code. Overall, the Java source code was approximately 40% shorter than the original C code.

The only disadvantage to using Java is that virtual machines must interpret between Java bytecodes and machine-specific op-codes. This translation process makes Java execution speeds slow. Newer virtual machines use a technique called *just in time* (JIT) compilation to help alleviate the speed penalty generated by code translation. This technique involves caching recently translated bytecode with its corresponding translation. If the bytecode reoccurs, the previously translated native code is executed without re-translation. The impact of JIT compilation depends heavily on the locality of the code. For short loops, the effect is dramatic, while in the worst case, there is actually a reduction in speed incurred due to increased overhead.

3 Specifics of the Java implementation of *bddlib*

To date, enough of the *bbdlib* code has been converted to Java to evaluate performance. Noticeably absent from *javaBDD*, the Java version, is support for multi-terminal OBDDs. Dynamic variable reordering has also not been implemented in Java. As noted by Bryant [3], dynamic variable reordering is of little benefit when the initial ordering has been chosen using the correct heuristic. Long observed that his algorithm for reordering can get caught in local minima, preventing the optimal ordering from being reached [5]. Many of the other OBDD packages currently available do not support this feature. Some higher level functions, such as implication, also have yet to be added to *javaBDD*, although all low-level support is present.

Other than the removal of garbage collection code and the use of standard hash table and dynamic array structures, *javaBDD* is very similar to *bddlib*. The most significant difference is in the use of pointers. Java does not allow pointers, replacing them with a memory reference data type. These references are strictly typed and can not be converted to unsigned integers or any other numeric type. Long makes extensive use of pointer to integer conversions and admits that without this ability, a major rewrite of the code is required. The most prevalent use is to use the least significant bit of a pointer with 8 byte alignment (3 least significant bits all zero) to implement complement edges. Thus, a pointer to a node with the pointer's least significant bit set means to complement the function implemented by the node. To maintain this functionality, *javaBDD* replaces a single 32 bit word / pointer with the following structure:

```
public class BDDref
{
        BDD             bdd;            // this is a Java reference to the BDD node
        boolean         isComplemented;         // this is the replacement for the lsb
}
```

Although this is a minor change, it significantly increases the time required to determine if the edge is a complement edge, as well as increases the amount of storage space required for each edge. It also requires additional memory dereferencing to determine whether two BDDref structures actually represent the same BDD node.

4 Results

A simple test program was written for the purpose of comparing the C and Java versions of the BDD package. For the sake of equality, variable reordering was disabled and multi-terminal BDD functions were not used. The test consisted of the creation of 200 variables, followed by 2000 AND operations and 2000 if-then-else operations. All tests were performed on the same 120 Mhz PowerPC workstation with 32 megabytes of RAM. The original test sizes were kept small enough that memory page swapping did not occur. The *bbdlib* C code was compiled with the GNU gcc C compiler. Three different Java virtual machines were used, one of which uses JIT compiler techniques. As the table shows, the difference between virtual machines was considerable, especially between the two standard virtual machines and the JIT compiler implementation. This test shows that the C version of the code is about 12 times faster than the fastest Java version, which is nearly twice as fast as the other Java machines. The test size was increased to 200,000 if-then-else operations and 200,000 AND operations, keeping the number of variables at 200. This showed the JIT Java virtual machine to be 37.38 times slower than C. It should be noted that this test has a high level of code locality which should greately improve JIT virtual machine performance.

The Java implementation used about 6 times as much memory as the C package. Part of this is due to the lack of pointers, as previously discussed. Some of the difference can be explained by the different ways Java and C report the memory usage data. The C code forces a garbage collection before reporting its memory usage, while the Java machine's results can include unused but not yet collected nodes. The memory usage statistics in the larger test show the Java code using 20 times as much memory. While the exact ratio is undetermined, it is likely that *javaBDD* uses more memory than *bddlib*. Further investigation is required before memory usage can be confidently compared.

For interest, some results of *javaBDD* on a CADE Sun workstation using Sun's Java developer's kit version 1.0 are also reported. The different workstations both ran the same Java bytecode, and in spite of the difference in processor power, the Sun gave very comparable results to a standard Java virtual machine running on the PowerPC.

Performance Comparison Between BDD Libraries (2000 ite's + 2000 and's + 200 variables created)		
Machine	Time (ms)	Memory Used (bytes)
Metrowerks CW11	3768	566231
MRJ 1.1	4407	521520
MRJ JIT v. 1.5 Beta	2360	532455
bddlib (C version)	320	89640
Sun JDK 1.0 (on Sun)	4361	432655

Figure 1

Larger Comparison Between BDD Libraries (200,000 ite's + 200,000 and's + 200 variables created)		
Machine	Time (ms)	Memory Used (bytes)
MRJ JIT v. 1.5 Beta	2250990	3571400
bddlib (C version)	6021	165464

Figure 2

5 Conclusion

At present, it appears that *javaBDD* is useful only for applications where performance is not critical, of which there are very few. The memory overhead could be reduced by converting the complemented edge paradigm to something more compatible with Java’s reference types. There are numerous ways to improve its speed, such as by optimizing the size and load of the hash tables and the operation cache. Java supports multithreading, and a multithreaded *javaBDD* library could give some performance increase as well. Better virtual machines should also enhance its performance. However, it is unlikely that any combination of these improvements will eliminate the order of magnitude gap between the two versions. At best, a Java verification tool would be about twice as slow as C code, which is in keeping with findings for other types of programs. Until Java resolves its interpretation speed problem, it will not be an effective solution for CAD tools.

6 Acknowledgment

The author would like to thank David Long, now of AT&T, for writing and distributing the *bddlib* source code.

References

1. Randal Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comp.* , August 1986, pp. 677-691.

2. Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. Report CMU-CS-92-160, Carnegie Mellon University, July 1992.

3. Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient Implementation of a BDD Package. In *Proc. 27th Design Automation Conference*, June 1990, pp. 40-45.

4. Jagesh V. Sanghavi, Rajeev K. Ranjan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. High Performance BDD Package By Exploiting Memory Hierarchy. In *Proc. 33rd Design Automation Conference*, June 1996, pp. 635-640.

5. David E. Long. ROBDD Package. Carnegie Mellon University, November 1993.

6. Tony Stornetta and Forrest Brewer. Implementation of an Efficient Parallel BDD Package. In *Proc. 33rd Design Automation Conference*, June 1996, pp. 640-644.

7. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley Publishing, 1996.

8. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing, 1995.