JNuke: Efficient Dynamic Analysis for Java

Cyrille Artho[1], Viktor Schuppan[1], Armin Biere[1],
Pascal Eugster[2], Marcel Baur[3], and Boris Zweimüller[1]

[1] Computer Systems Institute, ETH Zürich, Switzerland
[2] Avaloq Evolution, Zürich, Switzerland
[3] Al Dente Brainworks, Zürich, Switzerland

Abstract. JNuke is a framework for verification and model checking of Java programs. It is a novel combination of run-time verification, explicit-state model checking, and counter-example exploration. Efficiency is crucial in dynamic verification. Therefore JNuke has been written from scratch in C, improving performance and memory usage by an order of magnitude compared to competing approaches and tools.

1 Introduction

Java is a popular object-oriented, multi-threaded programming language. Verification of Java programs has become increasingly important. *Dynamic analysis,* including run-time verification and model checking, has the key advantage of having precise information available, compared to classical approaches like *theorem proving* and *static analysis*. There are fully automated dynamic analysis algorithms that can deduce possible errors by analyzing a single execution trace [1,7,14].

Dynamic analysis requires an execution environment, such as a Java Virtual Machine (VM). However, typical Java VMs only target execution and do not offer all required features, in particular, backtracking and full state access. Code instrumentation, used by JPaX, only solves the latter problem [9]. JNuke, our run-time verification and model-checking framework, contains a specialized VM allowing both backtracking and full access to its state. Custom checking algorithms can be implemented.

Related work includes software model checkers that apply directly to programs, for example, the Java PathFinder system (JPF) developed by NASA [17], and similar systems for checking Java programs [4,7,12,13] and other software [6,8,10,11]. JPF, as a comparable system, is written in Java. Hence it effectively uses two layers of VMs, the system layer and its own. Our benchmarks show that JNuke is more efficient.

2 System Design

The static part of JNuke includes a class loader, transformer and type checker, including a byte code verifier. When loading a Java class file, JNuke transforms the byte code into a reduced instruction set derived from the abstract byte code in [16], after inlining intra-method subroutines. Additionally, registers are introduced to replace the operand stack. A peep-hole optimizer takes advantage of the register-based byte code. Another component can capture a thread schedule and use code instrumentation to produce class files that execute a given schedule on an arbitrary VM or Java debugger [15].

At the core of JNuke is its VM, providing check-points [5] for explicit-state model checking and reachability analysis through backtracking. A check-point allows exploration of different successor states in the search, storing only the difference between states for efficiency. Both the ExitBlock and ExitBlockRW [2] heuristics are available for schedule generation. These algorithms reduce the number of explored schedules in two ways. First, thread switches are only performed when a lock is released, thus reducing interleavings. Second, the RW version adds a partial-order reduction if no data dependencies are present between two blocks in two threads. If no data races are present, behavioral equivalence is preserved. The supertrace algorithm [10] can be used to reduce memory consumption. For generic run-time verification, the engine can also run in *simulation mode* [17], where only one schedule defined by a given scheduling algorithm is executed. Event listeners can implement any run-time verification algorithm, including Eraser [14] and detection of high-level data races [1].

For portability and best possible efficiency, JNuke was implemented in C. A lightweight object-oriented layer has been added, allowing for a modern design without sacrificing speed. We believe that the choice of C as the programming language was not the only reason for JNuke's competitive performance. The ease of adding and testing optimizations through our rigorous unit testing framework ensured quality and efficiency. The roughly 1500 unit tests make up half of the 120,000 lines of code (LOC). Full statement coverage results in improved robustness and portability. JNuke runs on Mac OS X and the 32-bit and 64-bit variants of Linux (x86 and Alpha) and Solaris.

3 Experiments

The following experiments were used to compare JNuke to JPaX and JPF: two task-parallel applications, SOR (Successive Over-Relaxation over a 2D grid), and a Travelling Salesman Problem (TSP) application [18], a large cryptography benchmark [3] and two implementations of well-known distributed algorithms [5]: For Dining Philosophers, the first number is the number of philosophers, the second one the number of rounds. Producer/Consumer involves two processes communicating the given number of times through a one-element buffer. The experiments emphasize the aim of applying a tool to test suites of real-world programs without manual abstraction or annotations.

All experiments were run on a Pentium III with a clock frequency of 733 MHz and 256 KB of level II cache. Table 1 shows the results of run-time verification in simulation mode. Memory limit was 1 GB. Benchmarks exceeding it are marked with "m.o.".

The columns "Eraser" refer to an implementation of the Eraser [14] algorithm. Mode (1) distinguishes between accesses to individual array elements and is more precise. Mode (2) treats arrays as single objects and therefore requires much less memory for large arrays. Columns "VC" refer to the view consistency algorithm [1] used to detect high-level data races. Again, modes (1) and (2) concern array elements.

For comparison, the run times using the JPaX [9] and JPF [17] platforms are given. In the JPaX figures, only the time required to run the instrumented program is given, constituting the major part of the total run time. This refers to both the view consistency

Application	Sun's VM		JNuke	JNuke Eraser		JNuke VC		JPaX	JPF
	JIT	no JIT	VM	1	2	1	2	2	2
SOR	0.7	0.7	3.6	934.1	21.5	34.0	19.2	45.9	error
TSP, size 4	0.6	0.4	0.7	1.7	1.5	1.2	1.1	2.7	m.o.
TSP, size 10	0.6	0.4	1.8	10.0	9.3	9.1	8.4	56.3	m.o.
TSP, size 15	0.8	1.2	24.5	228.7	203.0	207.0	192.7	1109.5	m.o.
JGFCrypt A	6.6	19.1	415.0	m.o.	1667.7	m.o.	1297.7	m.o.	m.o.
Dining Phil. (DP 3 5000)	1.2	1.9	11.0	15.7	15.6	987.0	987.0	83.2	m.o.
Prod./Cons. (PC 12000)	1.6	1.5	5.6	8.1	8.1	71.8	71.8	error	m.o.

Table 1. Execution times for run-time verification in simulation mode, given in seconds.

and Eraser algorithms, which can use the same event log. JPaX currently only offers mode (2). View consistency is currently not implemented in JPF [17]. In its simulation mode, it ran out of memory after 10 – 20 minutes for each benchmark.

Generally, JNuke uses far less memory, the difference often exceeding an order of magnitude. Maximal memory usage was 66.7 MB in the SOR benchmark in mode (2). Analyzing a large number of individual array entries is currently far beyond the capacity of JPaX and JPF. Certain benchmarks show that the view consistency algorithm would benefit from a high-performance set intersection operation [14]. Because file I/O is not available in JPF, most benchmarks required manual abstraction for JPF.

The focus of our experiments for explicit-state model checking capability is on performance of the underlying engines, rather than on state space exploration heuristics. Therefore correct, relatively small instances of Dining Philosophers and Prod./Cons. from [5] were chosen where the state space can be explored exhaustively. JPF supports two modes of atomicity in scheduling, either per instruction or per source line. JNuke offers both ExitBlock and ExitBlockRW (EB and EB/RW, [2]) relying mainly on lock releases as boundaries of atomic blocks. Hence, a direct performance comparison is difficult. For this reason, Tab. 2 provides instructions per second in addition to absolute numbers. Timeout ("t.o.") was set to 1800 s.

JNuke often handles 5 – 20 times more instructions per second than JPF. Atomicity in scheduling has a large impact, as shown by the number of instructions executed for different heuristics. In absolute terms, JNuke is usually an order of magnitude faster.

4 Conclusions and Future Work

JNuke implements run-time verification and model checking, both requiring capabilities that off-the-shelf virtual machines do not offer. Custom virtual machines, however, should achieve a comparable performance, as JNuke does. Scheduler heuristics and verification rules can be changed easily. JNuke is more efficient than comparable tools.

Future work includes a garbage collector and a just-in-time compiler. The segmentation algorithm [7] would reduce false positives. Static analysis to identify thread-safe fields will speed up run-time analysis. Finally, certain native methods do not yet allow a rollback operation, which is quite a challenge for network operations.

	JNuke (EB/RW)			JNuke (EB)			JPF (lines)			JPF (byte codes)		
	time	#ins	#ins/s	time	#ins	#ins/s	time	#ins	#ins/s	time	#ins	#ins/s
DP 2 10	6.4	189	29.6	8.6	323	37.6	84.7	503	5.9	816.9	2,770	3.4
DP 3 1	0.2	4	16.3	1.8	60	33.7	29.7	151	5.1	845.7	2,457	2.9
DP 3 2	5.1	127	25.0	20.7	693	33.5	186.7	1,112	6.0	t.o.	t.o.	t.o.
DP 3 3	32.5	818	25.2	92.8	2,994	32.3	597.6	3,670	6.1	t.o.	t.o.	t.o.
PC 100	1.1	67	61.9	1.1	80	70.4	48.4	279	5.8	390.8	1,363	3.5
PC 1000	15.9	661	41.5	17.7	794	44.8	409.4	2,795	6.8	t.o.	t.o.	t.o.

Table 2. Results for model checking. Time is in seconds, no. of instructions in thousands.

References

1. C. Artho, K. Havelund, and A. Biere. High-level data races. *Journal on Software Testing, Verification & Reliability (STVR)*, 13(4), 2003.
2. D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.
3. J. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. A methodology for benchmarking Java Grande applications. In *Proc. ACM Java Grande Conference*, 1999.
4. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. Intl. Conf. on Software Engineering (ICSE'00)*. ACM Press, 2000.
5. P. Eugster. Java Virtual Machine with rollback procedure allowing systematic and exhaustive testing of multithreaded Java programs. Master's thesis, ETH Zürich, 2003.
6. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. ACM Symposium on Principles of Programming Languages (POPL'97)*, 1997.
7. J. Harrow. Runtime checking of multithreaded applications with Visual Threads. In *Proc. SPIN Workshop (SPIN'00)*, volume 1885 of *LNCS*. Springer, 2000.
8. R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. Winter USENIX Conf. (USENIX'92)*, 1992.
9. K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Proc. Run-Time Verification Workshop (RV'01)*, volume 55 of *ENTCS*. Elsevier, 2001.
10. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
11. G. Holzmann and M. Smith. A practical method for verifying event-driven software. In *Proc. Intl. Conf. on Software Engineering (ICSE'99)*. IEEE/ACM, 1999.
12. M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a run-time assurance tool for Java programs. In *Proc. Run-Time Verification Workshop (RV'01)*, volume 55 of *ENTCS*. Elsevier, 2001.
13. Robby, M. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proc. European Software Engineering Conf. (ESEC'03)*, 2003.
14. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. on Computer Systems*, 15(4), 1997.
15. V. Schuppan, M. Baur, and A. Biere. JVM-independent replay in Java. In *Proc. Run-Time Verification Workshop (RV'04)*, ENTCS. Elsevier, 2004.
16. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer, 2001.
17. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. IEEE Intl. Conf. Automated Software Engineeering (ASE'00)*, 2000.
18. C. von Praun and T. Gross. Object-race detection. In *OOPSLA 2001*. ACM Press, 2001.