Trace Based Optimizations of the Jupiter JVM Using DynamoRIO

Kamran Farhadi
Department of Electrical & Computer Engineering, Computer Engineering Research Group
University of Toronto
kfarhadi@eecg.toronto.edu

Kai Yi Kenneth Po
Department of Electrical & Computer Engineering, Computer Engineering Research Group
University of Toronto
kpo@eecg.toronto.edu

Levon Stepanian
Computer Systems Research Group
Department of Computer Science
University of Toronto
levon@cs.toronto.edu

ABSTRACT

For our ECE1724 project, we use DynamoRIO to observe and collect statistics on the effectiveness of trace based optimizations on the Jupiter Java Virtual Machine. Some of the metrics we measure include general program performance and run time. Our final results indicate that Jupiter performs extremely poorly when run above DynamoRIO. We scour the logs generated by DynamoRIO for reasons and explanations as to why this is the case; we attribute its poor performance to a large number of indirect branch lookups, the direct threaded nature of the Jupiter JVM, small trace sizes and early trace exits.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors — *compilers, interpreters, optimization, run-time environments.*

General Terms

Measurement, Performance, Experimentation.

1. INTRODUCTION

Java programs are notorious for being slow because they require an interpreter to translate java byte codes to native machine instructions. In contrast, Java program interpreters or virtual machines (JVMs) are written in low-level languages, and are executed. In our project, we use DynamoRIO in an attempt to dynamically optimize executing instances of the University of Toronto (U of T) Jupiter JVM while it interprets benchmark programs from the Java Grande Forum (*JGF*). The results we obtain from the experiments are not very promising; the benchmarks programs perform poorly when run on DynamoRIO. We have chosen Jupiter because it was the one VM we were able to run atop the DynamoRIO framework.

The rest of this paper is organized as follows: Section 3 describes the background behind DynamoRIO. Section 4 describes Jupiter's architecture and design goals. Section 5 outlines our project and methodology. Section 6 presents our findings and results from the benchmark program runs. Section 7 provides some discussion behind our results. We conclude in Section 10.

ECE1724 — Software Systems for Runtime Program Optimizations
December 18, 2003 Toronto, Ontario, Canada. Copyright 2003.

2. RELATED WORK

Although the idea behind our project was derived independent of any pre-existing work, it closely mirrors research performed by Amarasinghe et. al. [2]. The main difference between their work and ours is that we treat the interpreter (JVM) as a black box; we do not feed logical or native program counters to DynamoRIO in order to produce longer executing traces. Also, experiments in [2] are brief and based on TinyVM which is a simple JVM while we focus on detailed statistics on a more complex JVM (Jupiter). Therefore, our work can best be characterized as a black-box analysis of the performance of Jupiter running on top of DynamoRIO.

DELI [5] is yet another framework that introspects code at runtime. Its goal is to provide DELI clients the ability to perform a number of different operations on running programs at the intermediate layer between hardware and software; these operations include ISA emulation, software patching, sandboxing and optimization among others. Though DELI does not have any direct relationship with our project, it is provided as a reference, to provide a sense of the operations that can be performed on a program at runtime. It is also a more recent project from the same company that created Dynamo.

3. DYNAMORIO FRAMEWORK

DynamoRIO is an extension of the HP Dynamo project [4] which was originally conceived to enhance program performance on the HP architecture by optimizing frequently executed traces of code at run time. Both MIT and HP collaborated and ported Dynamo to support the IA-32 instruction set. Accompanying the port were added enhancements, including basic block caching (to minimize the number of context switches from Dynamo interpreter mode to native execution), an adaptive level-of-detail instruction representation for IA-32, as well as an API which allows programmers to build custom DynamoRIO clients; most notably, the API allows for the creation of custom traces by specifying hook functions callable by DynamoRIO. We exclude an exhaustive explanation of DynamoRIO in this report; interested readers are can refer to [3] for more details.

The details of DynamoRIO that are significant to our work include the following:

a. There are large overheads associated with context switches (from the basic block or trace cache to DynamoRIO), indirect branch lookups and direct branches; these overheads are listed in decreasing cost order. Figure 1 in [3] provides a graphical display of the various transitions in state that occur in DynamoRIO, where the dotted lines represent the most costly transitions.

b. DynamoRIO traces are sequences of frequently executed basic blocks stitched together. Branches from traces can be in the form of direct branches or indirect branch lookups (which may go to the basic block cache or remain in the trace cache).

4. THE JUPITER JVM

The Jupiter JVM is an ongoing research project at the University of Toronto [1]. Its goal is to provide a JVM that scales up to a large (128) processor cluster of workstations addressing memory locality, parallel garbage collection, memory consistency, efficient threads and synchronization. To facilitate a quick and efficient development process, the JVM architecture is extremely modularized, providing for near plug-and-play-ability.

The details of the JVM that are significant to our work include the following:

a. Jupiter JVM can be run in one of two modes — a loop-dispatch mode or a direct threaded mode (we run the default direct threaded mode in our experiments)

The reason we ended up choosing to run Jupiter over other open source JVMs (such as Kaffe or TinyVM) was because Jupiter is single-threaded and does not have a dependence on signals (DynamoRIO does not work well with multi-threaded and signal dependant programs) [8]. Having said this, it should be noted that it took us close to four days to get the Jupiter JVM framework up and running.

5. PROJECT DETAILS

In this section, we describe the benchmark programs used for our experiments, as well as descriptions for each of the statistics we attempted to measure.

5.1 Benchmarks

We chose the Java Grand Forum (*JGF*) [9] as our source for benchmark programs. Ingoring *Section 3* of *JGF* (due to project time constraints), we focused our analysis on *Section 1* and *Section 2* programs; *Section 1* programs benchmark low-level operations (i.e. arithmetic, casting, loops, method calls etc.), while *Section 2* programs benchmark kernels: short code pieces that carry out specific operations frequently (i.e. heapsort, LU factorisation, encryption, matrix multiply etc.).

From the two *JGF* sections, we picked the following six programs for experimentation: *Cast*, *Create* and *Arith* (from *Section 1*) and *SeriesA*, *HeapsortA* and *CryptA* (from *Section 2*).

Our *Section 1* choices include the longest running (*Create*) and shortest running (*Cast*) benchmarks, in contrast to our *Section 2* choices which were picked at random. (Note that *Section 2* benchmarks come in three different verions: *A*, *B* and *C* in order of increasing input sizes. For the *Section 2* benchmarks we chose the smallest input sizes, namely 10000, 1000000 and 3000000 for *SeriesA*, *HeapsortA* and *CryptA* respectively.)

5.2 Configurations and Measurements

In this section we provide a brief description of the various experimental configurations used and the measurements recorded from running the various configurations.

5.2.1 Configurations

Before delving into the various measurements we recorded, we should first define the set of JVM configurations we used in our experiments.

5.2.1.1 Configuration A

This configuration represents the case when programs were run using Jupiter running above DynamoRIO. Essentially, this configuration was designed to record any performance increases/ decreases attributable to DynamoRIO. It also allowed us to generate DynamoRIO log files from which derived statistics.

5.2.1.2 Configuration B

This configuration represents the case when programs were run using Jupiter alone. It is the baseline configuration for Jupiter JVM, allowing for comparisons to be made between Configuration *A* and *B*.

5.2.1.3 Configuration C

This configuration is the case when programs were run using the Java JDK JVM in pure interpretive mode, that is, without any just-in-time compilation.

5.2.1.4 Configuration D

This configuration is similar to *Configuration C* but differs in that the Java JDK VM was run using the Sun Hot Spot just in time compiler. Configuration *C* and *D* were used to measure how good or bad Jupiter performed with respect to a widely-distributed and popular JVM implementation.

5.2.2 Measurements

We now decompose our analysis of Jupiter and the experiments we conducted into the following five statistical categories. Note that the statisics in 5.2.2.1 and 5.2.2.2 will be used in Section 6,

whereas 5.2.2.3 through 5.2.2.5 will be used for discussion in Section 7.

5.2.2.1 Benchmark Results

This statistical measure is simply the output from the six benchmark programs run using Configurations *A* through *D*. Each benchmark program performs a certain set of operations and then outputs the statistics for a particular operation. For example, the *Arith* benchmark program displays statistics for the number of adds per second (adds/s) for integers, longs, floats and doubles (among other statistics for multiplication and division).

5.2.2.2 Benchmark Runtimes

This statistical measure is simply the total running times for each of the six benchmark programs run using Configurations *A* through *D*. In some cases, runtimes were generated as output as part of the benchmark program, in other cases, we had to use the Linux *gettimeofday* function to time the run.

5.2.2.3 Percentage Time Spent in Various States

This statistic measures the percentage of runtime DynamoRIO spends in various states of execution for each of the six benchmark programs. Possible states include executing from the cache, performing context switches, interpreting and performing indirect branch lookups. This measurement is done using Configuration *A*, and was generated by DynamoRIO with profiling enabled.

5.2.2.4 Trace Sizes

We recorded the number of trace generated, and a distribution of their sizes. This was done by examining the trace logs generated by DynamoRIO and parsing out the required information using a C program.

5.2.2.5 Early Trace Exits

We recorded the proportion of time traces were exited early. These values were again derived from the generated logs using a small C program.

5.3 Methodology

We ran the experiments mentioned in 5.2.2 using a single Pentium 4 2.4 GHz 800 MHz FSB desktop machine (with Hyper-threading enabled). The system came equipped with 512 MB DDR400 RAM, and was running Redhat 9.0 with the 2.4.22 kernel (SMP enabled). We used version 0.9.3 of DynamoRIO, beta version 1.0.0 of Jupiter and version 1.3.1_09 of the Sun JDK JVM.

DynamoRIO was run six times for each benchmark program (varying its generated logs from level one to three, profiling in two mechanisms, and generating traces). It took us approximately four days to run all of JGF *Section* 1 (after which we selected the three programs to analyse); it took us two days to run the three programs from JGF *Section 2*. The next section describes the results of our measurements.

6. RESULTS

6.1 Benchmark Results

As described in Section 5.2.1.1, we recorded the output from the six benchmark programs run with all four configurations. Figure 1 presents a subset of the output from the programs. The benchmarks fare the best under Configuration *D*, and worst under Configuration *A*. In general, the performance difference between Configuration *A* and *D* is almost 1200 fold. Thus, Jupiter JVM under DynamoRIO performs horribly with respect to the Java JVM with JIT Compilation enabled. As a side-note, the difference between Configuration *B* and *D* is on average a factor of 300, while the difference between Configuration *C* and *D* is only five fold. These latter statistics coincide with what one would assume to occur.

	Config A	Config B	Config C	Config D
Arith (adds/s)	66861.68	208062.42	2419945.8	2528551.2
Cast (casts/s)	62323.496	306605.2	3.33E+07	7.17E+07
Create (arrays/s)	38740.92	95181.44	1582811.6	1644451.6
CryptA (kb/s)	1.8716381	6.5801525	964.3201	2676.182
HeapsortA (item/s)	234.90181	849.7938	107550.01	1273885.4
SeriesA (coeff./s)	7.6367183	26.253296	403.94675	995.02466

Figure 1 — Benchmark Results

6.2 Benchmark Runtimes

To verify the poor performance of all the Configurations relative to Configuration *D*, we also measured the runtimes of the benchmark programs. Figure 2 provides a bar graph representation of the four different runtimes for each benchmark; for *Section* 2 benchmarks, the bar graphs decrease in height from Configuration *A* through to *D* for each program. This is an expected result from the results displayed in Section 6.1.

For the three *Section 1* benchmarks, however, an interesting observation is made; it seems that all four configurations display similar running times. In fact, Configuration *A* seems to perform as well as Configuration *D* in some cases. It turns out that each of the *Section 1* benchmark programs (but not the *Section 2* programs) are implemented as a sequence of iterative loops with the following loopgaurd for each:

```
while (time < TARGETTIME && size < MAXSIZE)
```

Figure 2 — JGF Section 1 Loopgaurd

Section 1 programs do not really have well-defined tasks to perform; they are given an allotted time to find the maximum performance of the JVM (i.e. in a certain amount of time, how

many calculations or operations are performed). In all three cases, it seems that the first condition on the loopgaurd fails before the second one does, causing the benchmark program to halt the current statistic it is measuring, and continue with the next statistic. For example, the *Arith* benchmark will stop recording statistics for Integer additions, and switch to statistics for Long additions (and continue on for Floats, Doubles etc.). Thus, *Section 1* programs will run for shorter times, whereas *Section 2* programs (because of the absence of such loopgaurds), will run longer.

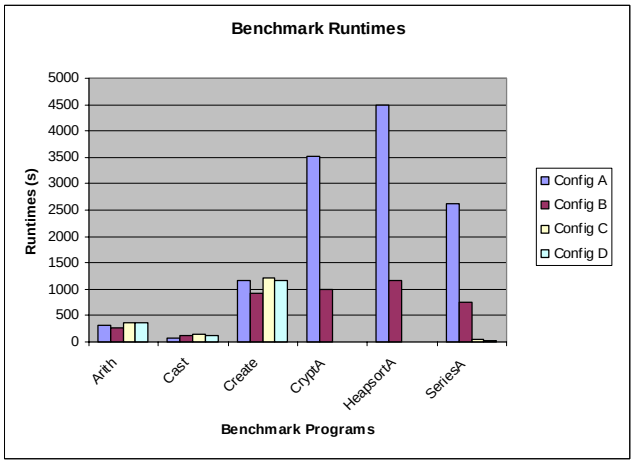

Figure 3 — Benchmark Runtimes

What is interesting is that the performance of Configuration *B* relative to Configuration *D* differs slightly from the results in [6]. In [6] Doyle argues a 2.20 factor in the slowdown observed in Jupiter, whereas in our case, we can see (from the Section 2 benchmarks in Figure 3), a near 350 factor in slowdown. This disparity can be attributed to the fact that we use different benchmarks, different machine setups, and a version of Jupiter compiled without specifying the optimization flag. We wish to stress that this difference in performance between Doyle's results and our results is by no means a scientific fact — it is just something we observed.

7. DISCUSSION

As is evident from the previous section, Jupiter does not fare too well when run using DynamoRIO; the performance increases promised by DynamoRIO are absent. We believe there are a number of reasons for this, some of which include expensive indirect branch lookups, short traces, the direct threaded nature of the Jupiter JVM and early trace exits.

7.1 Indirect Branch Lookups

Most of the runtime of the benchmark programs is spent executing native code from within either the basic block cache or the trace cache (both collectively referred to from now on as Fragment cache.) This is expected, since DynamoRIO caches native instructions into the basic block cache while interpreting them, thus avoiding subsequent context switches from program execution to interpretive mode.

Figure 4 depicts a breakdown of the percentage of time spent in four distinctive states when running Configuration *A*; on average, the benchmark programs spend 75% of the time executing code from within the Fragment cache, 23% of the time performing indirect branch lookups, 1.83% of the time performing context switches and spends the least amount of time, 0.5%, executing in interpretive mode.

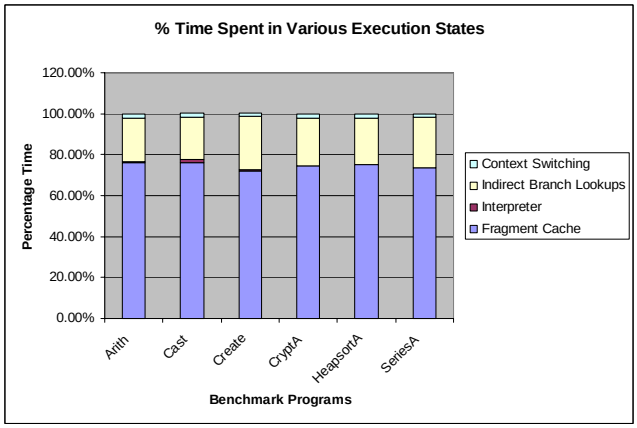

Figure 4 — Percentage of Time Spent in Various Execution States

From [3] and [4], we know that indirect branch lookups are very costly operations, second only to the overhead of performing context switches.

Using the generated trace logs, we were also able to calculate the percentage of exit stubs from the generated traces that contain indirect branch lookups (Figure 5 — Column 1), as well as the actual number of times these indirect branch lookup exit stubs were taken at runtime (Figure 5 — Column 2)

	Static Exit Stub Indirect Branch (%)	Indirect Branch Lookups Taken (%)
Arith	35.3	52.8
Cast	34.9	52.6
Create	36.8	44.1
CryptA	34.4	53.8
HeapsortA	34.2	54.3
SeriesA	35.1	52.1

Figure 5 — Indirect Branch Lookups (Static and Dynamic)

It is interesting to note also that this indirect branch lookup factor is something that is to be expected from Jupiter JVM since it is

direct threaded. From [7] we know that direct threaded interpreters have a heavy dependence on pointers to jump to the native instructions that will evaluate the bytecodes. They often have a huge number of indirect jumps (at every bytecode) which become expensive hash table lookups at runtime. From DynamoRIO's perspective, these indirect branch lookups "foil trace head identification heuristics" [2]. In other words, DynamoRIO does a poor job of predicting indirect jump targets since they depend on the positions of the interpreter in the high level program and not the location of jump instructions in the interpreter's binary. Furthermore, since each bytecode represents an address to branch to, the addresses "don't identify commonly occurring long sequences of native instructions" [2].

7.2 Trace Sizes

DynamoRIO generates traces of varying size, but most of the sizes aggregate around the 150 to 210 byte size. This contributes to the poor performance we observed in Section 6. The reason behind the performance degradation is that DynamoRIO is failing to amortize the cost of trace creation over the duration of execution. This overhead involves not only trace identification and generation, but also any optimization that DynamoRIO performs on the "straightened" code.

A plot of the trace sizes (Figures 6 and 7) seems to illustrate that the traces of Jupiter with each of the six benchmark programs is extremely similar with very small variance.

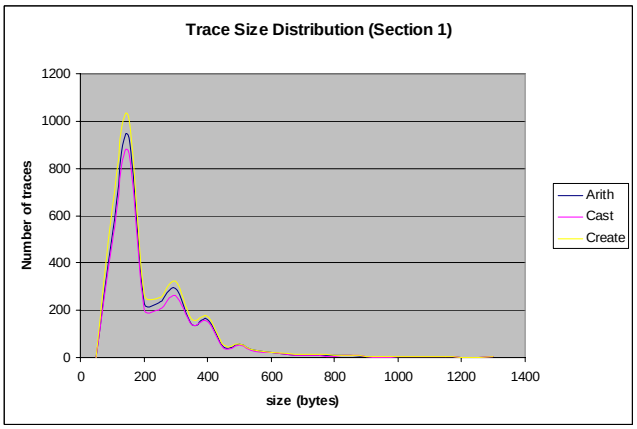

Figure 6 — Trace Size Distribution (Section 1)

For aesthetical purposes, we graph only trace sizes up to 1400 bytes, whereas there are traces generated that are larger in size. Figure 8 verifies this fact, since the maximum trace sizes are much larger than the upper bound on the x-axis of our graphs. Plotting these would distort the graphs and for that reason we do not include them. It is important, however, to note that the frequency of large sizes traces is very small. (There is one trace for a few sizes greater than 1400 bytes.)

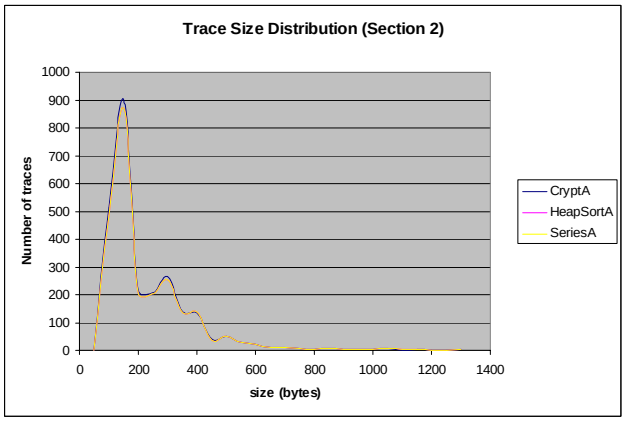

Figure 7 — Trace Size Distribution (Section 2)

The minimum trace size for each of the runs is the same (54 bytes), whereas the largest varies from 2136 to 3320 bytes as seen in Figure 8. The reason why there are a lot of traces for the *Create* program is because Create is the longest running benchmark, possibly providing more opportunities for trace identification and generation.

	Num Generated	Smallest	Largest	Average
Arith	2748	54	2136	209
Cast	2526	54	2136	210
Create	3095	54	3320	208
CryptA	2589	54	2136	205
HeapsortA	2496	54	2496	207
SeriesA	2555	54	2136	207

Figure 8 — Trace Statistics (in bytes)

Our intuition behind the short traces is that most of the bytecodes used in the benchmark programs will be translated to a small machine instruction implementations. Since bytecodes are quite often repeated during execution of the benchmark programs (i.e. *Arith* performs many IADD, FADD, LADD etc), they are detected as being hot; but their implementations are rather short in length.

7.3 Early Trace Exits

One of the goals in [2] was to delay the exits from traces till most of the trace had been executed. Specifically, the authors were able to increase the percentage of time a late exit stub was taken.

Having traces exit early counteracts the amortization property upon which trace generation and execution is based on. In our case, there is an argument to be made about early trace exits and their contributions to Jupiter's degraded performance.

We analyzed the traces that were logged by DynamoRIO and found that there were a large number of traces logged that had a

small number of exit stubs (one to four) that occurred at the end of the trace and were "taken" most of time. In the cases where a trace had more than four or five different exit stubs (we had some traces that had up to 13 different exit stubs), we observed that the trace would exit not only from the latest exit stub, but also from the earlier stubs.

In trying to quantify our observations, we picked twenty random traces from a run of the *Cast* program, and found that on average, early trace exits were occurring 58% of time during program execution.

7.4 Miscellaneous

Another possible source of degraded performance is the log generating functionality of DynamoRIO. This fact has been documented in both [2] and [8], but the extent of the performance hit is unclear to us.

8. FUTURE WORK

The scope of our project was intended to accommodate the limited amount of time we had to complete it. For that reason, there is a lot of open work that still needs to be addressed in figuring out why Jupiter performs poorly with DynamoRIO.

One may perform a more exhaustive study on the percentage of times early exits are taken from generated traces. We have only hand picked a few traces to argue our point.

It may also be interesting to perform a disassembly of some of the most common sized traces (150 bytes long), and the longest traces (see Figure 8) and figure out where they map to in the interpreter. This was something we intended to due for the project, but it just did not pan out.

It would also be interesting to run Jupiter on DynamoRIO with the SpecJVM 98 benchmarks as Boyle [6] has done and compare performance results.

Finally, one can do the similar work done in [2] for modifying Jupiter source code to guide the process of dynamic optimization for DynamoRIO so that it avoids pitfalls mentioned in this paper. Observation of a significant increase in performance of the modified Jupiter on DynamoRIO will essentially confirm this paper's results for the current slow performance.

9. ACKNOWLEDGEMENTS

We would like to thank Michael Voss for his help in defining the scope of our project and for his help answering questions throughout the course of the work. We would also like to thank Marc Berndl for taking the time to clarify some of the ideas behind trace based optimizations.

10. SUMMARY

We have used DynamoRIO to observe and collect statistics on the effectiveness of trace based optimizations on the Jupiter Java Virtual Machine. We observe poor runtimes and program performance when utilizing DynamoRIO. We believe the poor performance is attributable to a large number of indirect branch lookups, the direct threaded nature of the Jupiter JVM, small trace sizes and early trace exits. Overall, DynamoRIO's trace selection process is inefficient because of the misleading dynamic behavior of interpreters. There is still future work to confirm our preliminary explanations, including a more in depth analysis of early trace exits, and possibly the disassembly of generated traces.

11. REFERENCES

[1] Abdelrahman T.S., Doyle P. Jupiter: A Modular and Extensible JVM. 3rd Workshop on Java for High Performance Computing, ICS 2001, June 2001.

[2] Amarasinghe S., Baron I., Breuning D., Garnett T., Sullivan G. Dynamic Native Optimization of Interpreters. ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators, June 2003.

[3] Amarasinghe S., Bruening D., Garnett T. An Infrastructure for Adaptive Dynamic Optimization. IEEE CGO 2003.

[4] Bala V., Duesterwald E., Banerjia S. Dynamo: A Transparent Dynamic Optimization System. ACM SIGPLAN PLDI 2000.

[5] Desoli G., Duesterwald E., Faraboschi P., Fisher J., Mateev N. DELI: A New Run-Time Control Point. The 35th Annual International Symposium on Microarchitecture (MICRO 2002), Nov. 2002.

[6] Doyle P. Jupiter: A Modular and Extensible Java Virtual Machine Framework. Master's Thesis 2002. University of Toronto.

[7] Garnett T. Dynamic Optimization of IA-32 Applications Under DynamoRIO. Master's Thesis 2003. Massachusetts Institute of Technology.

[8] The DynamoRIO Collaboration. Using DynamoRIO. http://www.cag.lcs.mit.edu/dynamorio/doc/using.html

[9] The Java Grande Forum Benchmark Suite. Benchmark Suite Contents. http://www.epcc.ed.ac.uk/javagrande/seq/contents.html