# Continuous Bytecode Instruction Counting for CPU Consumption Estimation

Andrea Camesi* Jarle Hulaas Walter Binder
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL), CH–1015 Lausanne, Switzerland
Email: firstname.lastname@epfl.ch

## Abstract

*As an execution platform, the Java Virtual Machine (JVM) provides many benefits in terms of portability and security. However, this advantage turns into an obstacle when it comes to determining the computing resources (CPU, memory) a program will require to run properly in a given environment. In this paper, we build on the Java Resource Accounting Framework, Second Edition (J-RAF2), to investigate the use of bytecode instruction counting (BIC) as an estimation of real CPU consumption. We show that for all of the tested platforms there is a stable, application-specific ratio of bytecodes per unit of CPU time – the experimental bytecode rate ($BR_{exp}$) – that can be used as a basis for translating a BIC value into the corresponding CPU consumption.*

## 1. Introduction

The Java programming language [8] and the Java Virtual Machine (JVM) [13] are a prominent programming language and deployment platform. However, the lack of mechanisms to predict, monitor, and limit the resource consumption of programs (resource management) is a severe limitation in many settings where Java is used nowadays, such as application servers, web services, grid computing, and embedded systems.

In previous work [3, 4, 12] we introduced a portable bytecode instruction counting (BIC) scheme for Java, the *Java Resource Accounting Framework, Second Edition* (J-RAF2, http://www.jraf2.org). J-RAF2 relies on a platform-independent dynamic metric, the number of executed JVM bytecode instructions [6]. BIC can also be used as profiling metric [2]. Profiles based on BIC are precise, platform-independent, reproducible, directly comparable across different environments, and valuable to gain insight into algorithm complexity.

*Mr. Camesi is supported by the Swiss National Science Foundation.

Computing CPU consumption is essential for software development, and for monitoring and management of software deployment. Nonetheless, BIC and CPU time are distinct metrics for different purposes. Currently, there are no tools supporting this activity in a portable way throughout heterogeneous computing environments. Hence, the study of the relationship between BIC and CPU consumption is crucial for its potential for estimating CPU consumption based on static and portable analysis methods. The idea followed here is to use profiles based on BIC (i.e. extending the J-RAF2 bytecode instrumentation) to estimate, as accurately as possible, the CPU time of an application on a particular target system.

The first contribution of this paper is the presentation of a novel methodology for studying the relationship between BIC and CPU consumption, based on a continuous profiling system that has minimal influence on the executed system. The fine-grained measurement methodology proposed here combines bytecode instrumentation with hardware performance counters, in order to improve the quality of results (i.e., it is efficient and gives precise samples with low measurement perturbation). On the opposite, we have found that classical profiling tools for Java, based on the JVM Profiler Interface (JVMPI) [17] result in enormous overhead, hence a higher perturbation, and requires profiling agents to be written in platform-dependent native code. Since our methodology entirely relies on bytecode instrumentation, it does not require any modification to whichever JVM is employed, and remains therefore applicable in very heterogeneous systems.

As a second contribution of this paper, we show *experimentally* that for each platform there is a stable, application-specific *bytecode rate*

$$BR_{exp} = \frac{number\ of\ executed\ bytecode\ instructions}{elapsed\ CPU\ time}$$

that can be used for translating a BIC value into the corresponding CPU consumption. Our experiments show that this stability will usually cover whole applications, but may also sometimes be more local, depending on the specific activities of each application. We demonstrate the relative

stability of $BR_{exp}$, on the basis of the SPEC JVM98 and SPEC JBB2005 benchmark suites, executed in several different environments. We also introduce a *theoretical BR* (written $BR_{th}$ in the following), which corrects the influences due to our specific measurement methodology, in order to evaluate the $BR$ that an uninstrumented application would yield, as well as to indirectly assess the validity of the results of this paper.

This paper is structured as follows. Section 2 presents some potential applications of $BR_{exp}$. Section 3 explains the approach of BIC to estimate CPU time consumption in the context of J-RAF2. Section 4 presents results of $BR_{exp}$ measurement in different environment using the SPEC JVM98 and SPEC JBB2005 benchmarks. Section 5 introduces our notion of *theoretical BR* for assessing our results. Section 6 presents the measurement optimizations leading to the continuous $BR_{exp}$. Section 7 discusses the results obtained so far. Section 8 outlines related work. And finally, Section 9 concludes this paper.

## 2. Applications of $BR_{exp}$

The stability of $BR_{exp}$ revealed in this paper is a valuable property in many ways. First, it serves to better understand the behaviour of JVMs and may provide foundations for new kinds of optimizations. Second, it helps assessing the value of BIC as a portable basis for estimating CPU consumption.

We see two immediate approaches for exploiting the stability of $BR_{exp}$ in practice. One possibility is to first compute the $BR_{exp}$ offline (by benchmarking or profiling) for a given program in a particular environment, and then to use this value at production time to predict CPU time; this requires one calibration for each platform and application. The second possibility is to dynamically determine the $BR_{exp}$ of a given program in a particular environment - *the continuous* $BR_{exp}$ ($CBR_{exp}$) - based on the hypothesis that $BR_{exp}$ will always tend to become (at least locally) stable, and then to use the knowledge of $BR_{exp}$ in various management tasks, like load-balancing or usage-based billing.

However, as a longer-term goal, new scenarios may be envisaged using BIC as enabler, such as:

- cross-profiling, especially when developing for limited devices: The developer could profile an application on his preferred platform $P_{develop}$ (typically a workstation), providing the profiler some configuration information concerning the intended target platform $P_{target}$ (e.g. an embedded system). The profile obtained on $P_{develop}$ would allow the developer to approximate a CPU time-based profile on $P_{target}$;
- performance prediction (sizing of resources, with limited over-provisioning);
- load balancing, with different applications in concurrency;
- resource awareness, when deploying applications throughout heterogeneous computer systems.

## 3. Estimating CPU Time via BIC

Monitoring of server systems is important to quickly detect performance problems and to tune the system depending on the workload. Moreover, resource management is a prerequisite to prevent resource overuse in extensible middleware that allows hosting of foreign, untrusted, potentially erroneous or malicious software components (prevention of denial-of-service attacks).

Existing work has studied static and dynamic discrete metrics related to the execution of Java bytecode [6, 7, 9]. Here we propose to extend this investigation with continuous bytecode instruction counting and to study the correlation between BIC and actual CPU consumption. In contrast to related work which takes a low-level approach [11, 15, 20], here we attempt a top-down approach by considering the entire execution platform, including the application and the JVM, as a black box, and we try to characterize this with a metric of bytecode execution rates. To this end, we measured the number of executed bytecode instructions at sufficiently frequent intervals, along with the elapsed CPU time for each thread with a high level of precision, as detailed below.

### 3.1. Bytecode Rewriting with J-RAF2

J-RAF2 uses a platform-independent dynamic metric, the number of executed JVM bytecode instructions [6]. The J-RAF2 BIC approach is based on the principle of bytecode instrumentation. The bytecode of Java classes is rewritten in order to make the execution of bytecode sequences explicit (in the present paper, BIC includes only the original bytecode instructions, not counting the additional instrumentation code). At runtime, each thread continuously maintains its own BIC, expressed as the number of executed JVM bytecode instructions. Periodically, each thread aggregates the collected information concerning its own execution within an account that may be shared with a number of other threads. During these information update routines, implemented in so-called *CPU managers*, the thread will also execute management code, e.g., to ensure that a given resource quota is not exceeded. J-RAF2 ensures that threads invoke their custom CPU manager regularly, after each execution of a given number of bytecode instructions, the *granularity*, which is a dynamically adjustable value. This scheme has the advantage of allowing the activation of such update routines as frequently as wanted, independently of the resolution of any predefined timers or schedulers. For an application that is deterministic, these activations will

Table 1: $BR_{exp}$ in JBB2005 (Sun JDK 1.5.0, **Intel Pentium 2.6 GHz, Linux 2.6**)

| | Mode | Average | Median | Std. deviation | % std. error |
|---|---|---|---|---|---|
| JBB2005 | Xint | 27 613 | 28 005 | 1 657 | 6.00% |
| JBB2005 | client | 395 073 | 395 316 | 25 483 | 6.45% |
| JBB2005 | server | 517 532 | 517 624 | 45 518 | 8.80% |

Table 2: $BR_{exp}$ in JBB2005 (Sun JDK 1.5.0, **Sun Sparc 1.5 GHz, Solaris 10**)

| | Mode | Average | Median | Std. deviation | % std. error |
|---|---|---|---|---|---|
| JBB2005 | Xint | 24 273 | 24 436 | 1 674 | 6.90% |
| JBB2005 | client | 421 648 | 423 074 | 19 287 | 4.57% |
| JBB2005 | server | 591 893 | 594 347 | 44 122 | 7.12% |

also happen in a deterministic and hence reproducible sequence.

Our approach allows to measure $BR_{exp}$ in a portable way, applicable to any JVM. The advantage of avoiding any modifications to the JVM is that we are able to easily collect the $BR_{exp}$ of a given instrumented application on many different combinations of hardware, OS and JVM. We use J-RAF2 bytecode instrumentation in order to measure the CPU time consumed by Java code. More precisely, we repeatedly compute the (per-thread) CPU time elapsed between each time the instrumented Java code invokes the information update routines and subtract the time spent inside the routines themselves. Hence, we can obtain the CPU time for a particular application on any given system.

### 3.2. Benchmarking Setup

For all our tests, the benchmarking was performed with the SPEC JVM98 benchmark suites [19] and the SPEC JBB2005 benchmark [18]. Both the benchmark code and the runtime libraries, i.e the JDK, were rewritten following our simple, unoptimized bytecode transformation schemes [12]. For SPEC JVM98, we used the standard input size of 100, but our experiments revealed that changing the input size has minimal effect on the results presented here.

We ran the benchmarks on a Linux Fedora Core 2 computer (Intel Pentium 4 CPU at 2.6 GHz, 512 MB RAM) and on a Sun Solaris 10 workstation (Sun Blade 1500 with an UltraSparc IIIi CPU at 1.5 GHz, 1024 MB RAM). These measurements were made with the Sun JDK 1.5.0 platform in its different execution modes, i.e., using the `Xint` command-line option for selecting a purely interpretative mode, and the `client` or `server` options for a delayed just-in-time compilation (JIT), respectively a more intensive load-time compilation. We also executed the tests with IBM JDK 1.4.2 on the above mentioned Linux machine, and the main results were quite comparable to the ones obtained with the Sun JDK, therefore we do not present them in this paper.

### 3.3. Using Hardware Performance Counters

The objective of determining the CPU consumption for Java bytecodes is difficult because of the level of precision that is required: the time taken to execute any single bytecode on recent hardware is usually far below the measurement resolution offered by the JVM or by the OS itself. Another difficulty is that the desired timings are specific to each {JVM, OS, hardware} platform combination.

We exploit the added precision provided by processor cycle counters, as found in *hardware performance counter* (HPC) enabled processors (such as the Intel Pentium 4 and Sun Ultrasparc CPUs), since standard Java APIs – even the Java 1.5 `System.nanoTime()` method – do not in practice provide the required level of resolution. In previous experiments, we used standard APIs (notably the JVMPI [17] profiling API) for measuring elapsed per-thread CPU time, but the inherent lack of resolution resulted in important measurement perturbations, hence noticeably worse sampling distributions than achieved here, especially in multithreaded applications. We estimate that the resolution increased approximately from 10 milliseconds up to 1.5 microseconds on our Intel Pentium machine with the adoption of a HPC-based library.

We used *PCL* (the *Performance Counter Library*) [1], which is a lightweight portable HPC library with C, C++, Fortran and Java APIs. This library enables our CPU time computation scheme to become portable across a wide variety of operating systems and architectures.

## 4. $BR_{exp}$ in Standard Benchmark Suites

On the basis of the sampled $\{BIC, cpu\ time\}$ couples, we computed the instant values of $BR_{exp}$. Then, for each test, we calculated the average (i.e. arithmetic mean), the median, the standard deviation and the percentage standard error (i.e. the standard deviation expressed as a percentage of the average) of the collected $BR_{exp}$.

Table 1 shows the results for SPEC JBB2005 in different JVM execution modes on our Linux machine. We recon-

Table 3: $BR_{exp}$ in JVM98 (Sun JDK 1.5.0 in **Xint mode**, Intel Pentium 2.6 GHz, Linux 2.6)

| | Average | Median | Std. deviation | % std. error |
|---|---|---|---|---|
| 201_compress | 54 225 | 53 918 | 2 012 | 3.71% |
| 202_jess | 25 970 | 25 887 | 1 579 | 6.08% |
| 209_db | 30 586 | 31 392 | 3 806 | 12.44% |
| 213_javac | 27 822 | 28 593 | 1 932 | 6.94% |
| 222_mpegaudio | 76 432 | 75 907 | 2 853 | 3.73% |
| 227_mtrt | 19 207 | 18 923 | 1 922 | 10.01% |
| 228_jack | 22 440 | 22 501 | 781 | 3.48% |

Table 4: $BR_{exp}$ in JVM98 (Sun JDK 1.5.0 in **client mode**, Intel Pentium 2.6 GHz, Linux 2.6)

| | Average | Median | Std. deviation | % std. error |
|---|---|---|---|---|
| 201_compress | 1 146 590 | 1 145 674 | 124 275 | 10.84% |
| 202_jess | 623 211 | 607 167 | 107 858 | 17.31% |
| 209_db | 194 540 | 122 619 | 171 637 | 88.23% |
| 213_javac | 395 946 | 413 049 | 110 746 | 27.97% |
| 222_mpegaudio | 1 608 147 | 1 620 712 | 88 212 | 5.49% |
| 227_mtrt | 673 549 | 684 004 | 88 131 | 13.08% |
| 228_jack | 245 388 | 175 241 | 137 460 | 56.02% |

ducted the JBB2005 benchmark test on our Sun Sparc machine, with the results shown in Table 1. We observe that $BR_{exp}$ remains linewise fairly stable, i.e., the standard error is low in all execution modes and environments. The samples are also well centered, which is testified by the fact that the average and median of each test remain quite close to each other. The goal of SPEC JBB2005 (Java Business Benchmark) is to evaluate the performance of server-side Java implementations by emulating a three-tier client/server system. It is designed to reflect the most common types of server-side Java applications, by exercising the implementations of the JVM, Just-In-Time (JIT) compiler, garbage collection (GC), threads and some aspects of the operating system [18]. Based on this description, we may infer that this benchmark implements a fairly varied set of activities, and that the statistical characteristics of the collected samples, especially the stability of $BR_{exp}$ are representative of many real-world applications. In the following we verify whether SPEC JVM98 confirms this tendency of applications to yield a constant $BR_{exp}$ throughout their execution.

SPEC JVM98 [19] is a well-known general-purpose benchmark suite, which consists of the following Java programs:

- *compress*: a popular utility used to compress/uncompress files;
- *jess*: a Java expert system shell;
- *db*: a small data management program;
- *javac*: an old Java compiler, compiling 225,000 lines of code;
- *mpegaudio*: an MP3 audio stream decoder;
- *mtrt*: a dual-threaded program that ray traces an image file;
- *jack*: a parser generator with lexical analysis;

Tables 3 and 4 list the results obtained with SPEC JVM98 run in `Xint`, respectively in `client` mode on our Linux machine. We observe that in this benchmark each application has specific $BR_{exp}$ properties. We attribute this to the fact that each of these applications consists of highly specialized internal activities. We appreciate that 201_compress and 222_mpegaudio exhibit the lowest percentage standard errors, as well as the highest average $BR_{exp}$; this is certainly because these applications execute bytecodes that are rather simple (i.e., inexpensive in terms of required CPU time) and repetitive (i.e., with strong code and data locality). However, it is important to realize that the $BR_{exp}$ values available here must be weighted by the overhead due to our bytecode instrumentation scheme. This issue will be addressed in the next section.

Focusing more on the very bytecodes that constitute each application, it seems rather obvious that the variation of instant $BR_{exp}$ values, expressed as the percentage standard error, depends at least partly on the difference of costs of the bytecodes involved: if an application contains a relatively homogeneous set of bytecodes, the percentage standard error will tend to be lower. Other factors of variation are the execution of GC, JIT, and native code (as part of the JVM itself, or of the JDK, or finally as the result of dynamic compilation). As confirmed by comparing Tables 3 and 4, the JIT itself, as well as the mixture of interpreted and compiled bytecodes in the same execution will increase the percentage standard error.

The higher percentage standard error in 228_jack (Table 4) is caused by an intensive workload during initialization of the program and certainly also by the fact that this application is known to be particularly exception intensive [5][14] (i.e. exception management causes an additional workload in the VM, which in turn reduces $BR_{exp}$).

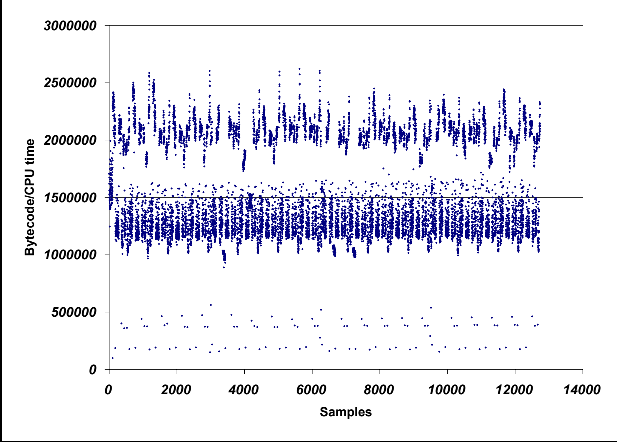



Figure 1: Samples taken during a run of 201_compress on our Linux machine, `server` mode.

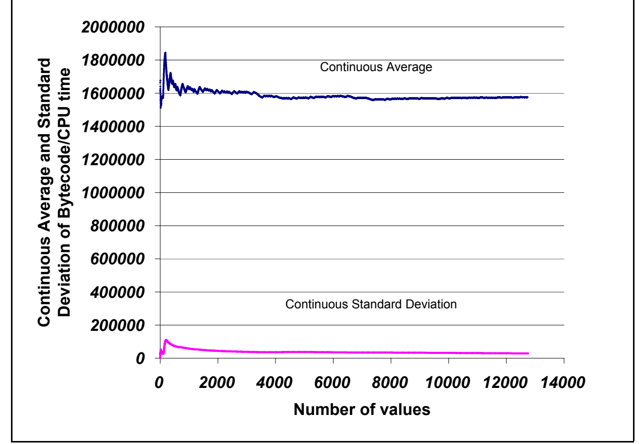



Figure 2: Evolution of average and standard deviation of sampled $BR_{exp}$ values (201_compress on Linux, `server` mode).

For 209_db, we attribute the high percentage standard error to its unoptimized use of memory, since it spends most of the time sorting its internal database using a simple algorithm that ignores data locality, which results in serious thrashing of the underlying hardware memory management system [16]. It is striking that SPEC JVM98 was initially constituted exclusively of real-world applications, except for 209_db, which is a synthetic benchmark.

In `server` mode, both $BR_{exp}$ average and standard error values are somewhat higher in comparison to the `client` mode. This seems natural, because in `server` mode, the JIT behaves more aggressively. Similar results were found with the IBM virtual machine on our Linux machine.

For an example illustrating the samples taken, see Figure 1, which exhibits the raw results in the case of the 201_compress benchmark on our Linux machine in `server` mode. Figure 2 shows - for the same run - the evolution of the average and standard deviation of $BR_{exp}$, obtained simply by recalculating the statistics for each new sample since the beginning of the execution. For obvious performance and non-perturbation purposes, this computation was actually done offline on the samples collected during execution.

## 5. Determination of a Theoretical $BR$

An interesting point to be addressed in our approach is the study of the influence of our instrumentation on the $BR_{exp}$ measurement and the possibility to estimate as precisely as possible the theoretical $BR$ (i.e. the $BR$ that each application would consume in its non-instrumented version), called $BR_{th}$ in the following. This fictive value should in turn allow us to establish the CPU time $CpuTimeOrig$ that the non-instrumented application would consume at any point of its execution. In the version of the J-RAF2 tool used in this paper, we do not account the bytecodes added for the instrumentation, neither in the rewritten application, nor in the rewritten JDK. This implies that the measured $BR_{exp}$ reflects the additional CPU time $CpuTimeInstr$ consumed, whereas it accounts for the same amount of bytecodes $BIC$ as executed by the original application. These relationships may be formulated as follows:

$$BR_{th} = BIC/CpuTimeOrig \tag{1}$$

$$BR_{exp} = BIC/CpuTimeInstr \tag{2}$$

We have shown in previous publications on J-RAF2 (e.g. [12], but in slightly different settings), that our bytecode instrumentation scheme results in a different execution time overhead $Overhead_{app}$ for each application. Knowing these overheads, the last equation can be expressed as follows in Relation 3. The approximation comes from the fact that overheads are computed from actual measurements, i.e. real (wallclock) execution times in the case of SPEC JVM98, respectively BOPS (business operations per second), the measurement metric of SPEC JBB2005; these benchmarks were however carefully conducted on dedicated machines, with the bare minimum system services running at the same time.

$$BR_{exp} \approx BIC/(CpuTimeOrig * Overhead_{app}) \tag{3}$$

Hence, the general formula we use to calculate $BR_{th}$ is:

$$BR_{th} \approx BR_{exp} * Overhead_{app} \tag{4}$$

The latter formula allows us to evaluate $BR_{th}$, as shown in Tables 5 and 6 for SPEC JBB2005, and Table 7 for SPEC JVM98.

Table 5: $BR_{th}$ and speedups in JBB2005 (Sun JDK 1.5.0, **Intel Pentium 2.6 GHz, Linux 2.6**)

| | Mode | $BR_{exp}$ avg | $\% \: Overhead_{app}$ | $BR_{th}$ avg | $BR_{th}$ speedup | Orig. BOPS | Orig. speedup |
|---|---|---|---|---|---|---|---|
| JBB2005 | Xint | 27 613 | 228.2% | 90 624 | – | 745 | – |
| JBB2005 | client | 395 073 | 71.2% | 676 239 | 7.46 | 5 260 | 7.06 |
| JBB2005 | server | 517 532 | 59.4% | 824 665 | 9.09 | 6 283 | 8.43 |

Table 6: $BR_{th}$ and speedups in JBB2005 (Sun JDK 1.5.0, **Sun Sparc 1.5 GHz, Solaris 10**)

| | Mode | $BR_{exp}$ avg | $\% \: Overhead_{app}$ | $BR_{th}$ avg | $BR_{th}$ speedup | Orig. BOPS | Orig. speedup |
|---|---|---|---|---|---|---|---|
| JBB2005 | Xint | 24 273 | 309.9% | 99 495 | – | 414 | – |
| JBB2005 | client | 421 648 | 76.1% | 742 294 | 7.46 | 3 535 | 8.54 |
| JBB2005 | server | 591 893 | 86.9% | 1 105 933 | 11.11 | 5 256 | 12.69 |

The third column recalls the $BR_{exp}$ average values already shown in Tables 1 through 4. The fourth column represents, for each application, the corresponding measured J-RAF2 overhead.

The computed $BR_{th}$ average is represented in the fifth column. We observe that 201_compress and 222_mpegaudio keep the highest average values, also after applying the respective corrective factor of the overhead. Moreover, 209_db is confirmed as a bad performer, having the lowest $BR_{th}$ in `client` mode.

Then, in column six, we calculate what we call the $BR_{th}$ *speedup* due to JIT optimizations; this is obtained by dividing, for each application, the $BR_{th}$ for `client` (and `server`) mode by the $BR_{th}$ for interpreted (`Xint`) mode. This speedup is a synthetic measure of the relative performance of the JIT compilation strategy inside the given JVM. Nevertheless, it must not be forgotten that the precision of this value depends on the quality of the underlying measurements, which cannot easily be evaluated. To address this issue, the last two columns are used for verifying the results obtained so far, as explained in the following.

### 5.1. Verification of the Theoretical $BR$

For the evaluation of the overhead due to the instrumentation, we use as reference values the original wallclock execution times ($RealTimeOrig$) for SPEC JVM98, and the original BOPS for SPEC JBB2005, as shown in the seventh column of Tables 5 through 7.

We do the verification of $BR_{th}$ by calculating $Orig.\ speedup$, which is the ratio, for each application, of the original (i.e., non-instrumented) performance in interpreted (`Xint`) mode to its original performance in `client` (and `server`) mode. For JBB2005, this ratio is inverted, because of the nature of the BOPS unit of measurement.

The *Original speedup* should in principle be identical to the $BR_{th}$ *speedup*, in the sense that both are a measure of the performance of the JIT compilation strategy inside the given JVM, but this time in the direct execution of purely non-instrumentated code. Hence, we obtain a verification scheme for $BR_{th}$ which is not linked to our instrumentation, and which can therefore be reproduced independently. By comparing columns six ($BR_{th}$ *speedup*) and eight (*Orig. speedup*), we observe linewise fairly close values, which we interpret as a good validation of our calculation of $BR_{th}$, and more generally of the proposed methodology. If these speedups were identical, we might conclude that our measurements are perfectly precise, which of course is not the case.

Observing these speedups, the differences may come from various issues. One possible source of imprecision is the accumulation of measurement errors, in all the steps leading to the $BR_{th}$ values, according to Relation 4. Another possible source of difference may reside in the influence that the instrumentation has on various components of the JVM. In other words, the JIT might compile a given method in its original form, but not in its instrumented form, because of its added length or complexity. Concerning GC, we estimate that the instrumentation does not influence the speedups, because only a very small number of additional objects are created. These different influences are hard to quantify, especially in the black-box approach that we try to follow, but the fact that the *Original speedup* and the $BR_{th}$ *speedup* are so close tells us that they do remain within well-defined boundaries.

## 6. Filtering and Continuous $BR_{exp}$

Having validated our approach in the previous section, we address here some issues related to the possible practical use of $BR_{exp}$ at run-time, in order to implement a portable means for estimating the CPU consumption of a given application.

We argued in Section 4 that the major influence on $BR_{exp}$ is attributed to the cost of the specific mix of bytecodes inside each application. To minimize this influence, we propose different optimizations, which might be further enhanced, depending on one's knowledge of the given application behavior.

Table 7: $BR_{th}$ and speedups in JVM98 (Sun JDK 1.5.0 in Xint and client modes, Intel Pentium 2.6 GHz, Linux 2.6)

| | Mode | $BR_{exp}$ avg | $\% Overhead_{app}$ | $BR_{th}$ avg | $BR_{th}$ speedup | RealTimeOrig | Orig. speedup |
|---|---|---|---|---|---|---|---|
| 201_compress | Xint | 54 225 | 115.7% | 116 963 | – | 1 057 035 | – |
| 202_jess | Xint | 25 970 | 222.4% | 83 727 | – | 22 134 | – |
| 209_db | Xint | 30 586 | 135.0% | 71 877 | – | 52 042 | – |
| 213_javac | Xint | 27 822 | 175.3% | 76 594 | – | 277 335 | – |
| 222_mpegaudio | Xint | 76 432 | 64.9% | 126 036 | – | 894 735 | – |
| 227_mtrt | Xint | 19 207 | 302.7% | 77 347 | – | 29 668 | – |
| 228_jack | Xint | 22 440 | 154.3% | 57 065 | – | 183 745 | – |
| 201_compress | client | 1 146 590 | 54.0% | 1 765 749 | 15.10 | 7 734 | 13.67 |
| 202_jess | client | 623 211 | 32.2% | 823 885 | 9.84 | 2 514 | 8.80 |
| 209_db | client | 194 540 | 7.8% | 209 714 | 2.92 | 15 493 | 3.36 |
| 213_javac | client | 395 946 | 33.0% | 526 608 | 6.88 | 49 035 | 5.66 |
| 222_mpegaudio | client | 1 608 147 | 34.3% | 2 159 741 | 17.14 | 5 193 | 17.23 |
| 227_mtrt | client | 673 549 | 84.2% | 1 240 677 | 16.04 | 1 826 | 16.25 |
| 228_jack | client | 245 388 | 28.6% | 315 569 | 5.53 | 4 076 | 4.51 |

Table 8: Variation of $BR_{exp}$ inside a sliding window of 50 samples (Sun JDK 1.5.0 in client mode)

| | Linux | | Solaris | |
|---|---|---|---|---|
| | Number of initially filtered samples | Maximal % std. error | Number of initially filtered samples | Maximal % std. error |
| 201_compress | 100 | 3% | 100 | 2% |
| 202_jess | 100 | 2% | 100 | 2% |
| 209_db | 25 | 12% | 25 | 6% |
| 213_javac | 100 | 12% | 100 | 2% |
| 222_mpegaudio | 25 | 2% | 100 | 2% |
| 227_mtrt | 100 | 4% | 100 | 3% |
| 228_jack | 50 | 3% | 25 | 3% |
| JBB2005 | 400 | 2% | 500 | 1% |

### 6.1. Filtering of Extreme Values

Knowing that the initialization workload at the beginning of most applications is rather important (class loading and initialization) and usually unrelated to its cruising speed workload, we may enhance the reliability and stabilization speed of $BR_{exp}$ by dropping a certain quantity of initial samples. Our measurements using SPEC JVM98 seem to indicate that a reasonable number of initial samples to ignore lies between 200 and 250 values. On our Linux/Pentium machine, this number of samples corresponds to a duration varying roughly between 0.5 and 5 seconds (depending on the $BR_{exp}$ of the application).

### 6.2. Continuous $BR_{exp}$ Measurements

Realizing that raw samples are difficult to exploit, and that the $BR_{exp}$ average does not give a sufficiently dynamic and localized view of $BR_{exp}$, we may set up a sliding window of contiguous samples and compute statistics on the values collected within each window. This way, we are able to obtain a smoothed, and at the same time representative instant view of $BR_{exp}$, that we call the *continuous BR* ($CBR_{exp}$). Depending on the application scenario, $CBR_{exp}$ may also be computed in real-time, provided that the window size is not too large. Extrapolating from our experiments with JVM98 and JBB2005, we found a window size of 50 samples to be a reasonable approximation of the behaviour of most applications: larger windows occupy more memory, whereas smaller windows yield much more irregular $BR_{exp}$ values.

Table 8 shows the maximal percentage standard error at any moment, after filtering the given number of initial values (according to Section 6.1), and using a sliding window of 50 samples.

## 7. Discussion

In this paper, we correlate the number of executed bytecode instructions with the measured CPU time. We have found $BR_{exp}$ to be a statistically predictable value for any program - be it in its entirety or in arbitrarily chosen subparts of it - executed in a specific environment. We attribute this positive result to the locality effect, which, as a rule of thumb states that a program spends 80% of its time executing only 20% of its code. Indeed, the percentage standard errors remain rather low, whereas $BR_{exp}$, respectively $BR_{th}$ differ for each application, suggesting an appreciable specialization of activities, as it is typically the case in the SPEC JVM98 benchmark suite. For SPEC JBB2005, the regularity is shown to be quite good as well.

### 7.1. Application-specificity of $BR_{exp}$

We have seen that $BR_{exp}$ is bound to a given execution platform (i.e. CPU speed, OS, JVM) and to the application itself. This dependency means that each application must be executed once, as a calibration phase, in order to obtain its $BR_{exp}$, and subsequently to estimate its CPU consumption at production time. This calibration requirement limits the current practical usability of the $BR_{exp}$. However, as exposed in the above section on $continuousBR$, it would be possible to have a weak form of prediction by running the application for a few seconds in order to compute its $BR_{exp}$, and then later extrapolate the CPU consumption from that value.

From a longer-term perspective, we have shown that the portability of the BIC metric presents many important advantages in strongly heterogeneous environments [4]. Nevertheless, the ultimate goal is of course to find alternative approaches to avoid this calibration requirement.

### 7.2. $BR_{exp}$ Measurement Quality

In this subsection we want to emphasize that the major perturbations influencing $BR_{exp}$ are the J-RAF2 instrumentation of the code on one hand, and the presence of JIT compilation, GC, and native code on the other hand.

In this paper, instrumentation with J-RAF2 is done following its most simple, unoptimized scheme. Although this code transformation scheme results in very high overheads, the main reason for this choice is that the actual implementation of this scheme has remained stable over the years, leading to results that are easy to reproduce.

We recall that in our approach, the activities of the CPU manager itself are taken into account (and hence correctly excluded) in our measurements, since the CPU time consumed executing management tasks is deducted from the time consumed between each activation of the manager. This means that CPU manager activity has minimal impact at run-time. The influence of our scheme has been clearly demonstrated in previous papers, and the resulting overhead is known. We are therefore able to determine the perturbation it causes: our instrumentation scheme influences $BR_{exp}$ negatively, especially for object-oriented benchmarks, because method invocations are "punished".

Another possible source of perturbation, the JIT compiler is active in `client` and `server` modes of Sun's JVM. The JIT compiles the parts of the code that are executed the most frequently. The natural consequence of this is that $BR_{exp}$ will increase after each compilation phase. However, $BR_{exp}$ may temporarily seem to fall during the actual compilation phase in the case where the JIT executes in the same thread(s) as the measured application, because the JIT is native code using CPU time in a way that is not detectable with bytecode instrumentation. These accelerations and slowdowns will in turn artificially increase the standard deviation and percentage standard error of $BR_{exp}$. In `server` mode, compilation is even more intensive than in `client` mode, and will cause a more direct perturbation of $BR_{exp}$, as attested by the higher percentage standard error found in Tables 1 and 1.

GC is normally executed in a dedicated thread with Sun's JVMs, and since it also is implemented with native code, J-RAF2 cannot account for it. This means that we will see the same kind of perturbations as with the JIT.

In summary, all influences on $BR_{exp}$ correspond to the (known) J-RAF2 instrumentation overhead plus the (as yet not quantified) perturbations of the JIT, the GC and other portions of native code. Finding a portable way for estimating the amount of executed native code in part of our future activities.

### 7.3. Choosing the Sampling Frequency

J-RAF2 allows CPU managers to adjust the frequency at which threads invoke their CPU management code, by changing the value of the per-thread *granularity* variable. This value has to be carefully chosen in order to obtain useful samples. On the one hand, the granularity cannot be too low, because otherwise the dynamics of JIT compilation and garbage collection may distort the statistics, and the amount of memory required for the samples may perturbate the normal behavior of programs. On the other hand, if the granularity is chosen too high, there will not be enough samples to obtain insight into the fluctuations of $BR_{exp}$. Currently, J-RAF2 allows the value of the granularity to vary roughly speaking between zero (for "almost zero" delay between each sample, depending on the chosen kind of instrumentation scheme) and $2^{31}$ (for a delay of approximatively one second, on our Intel Pentium 4 at 2.6 GHz).

In order to experimentally determine the granularities that minimize perturbations, especially the distribution of samples, with Sun JVM 1.5.0 in its three different execution modes, we tested the following set of values: $2^{16}$, $2^{18}$, $2^{20}$, $2^{22}$ and $2^{24}$. Executions of a test program were launched 10 times with successively one, two, and three threads. Taking the average of the 10 runs as reference, the sampling distribution is here also expressed as a percentage standard error. The results for `client` mode are shown in Figure 3(b), and for `server` mode in Figure 3(c).

In `server` mode, we discover that the best choice for minimizing percentage standard error is a granularity of $2^{22}$ bytecodes. The percentage standard error is then lower than 0.5%. In `client` mode we found that, for a standard error lower than 1%, the same granularity is adequate.

The `Xint` mode completely deactivates the JIT compiler, and exhibits as expected even lower distributions, as

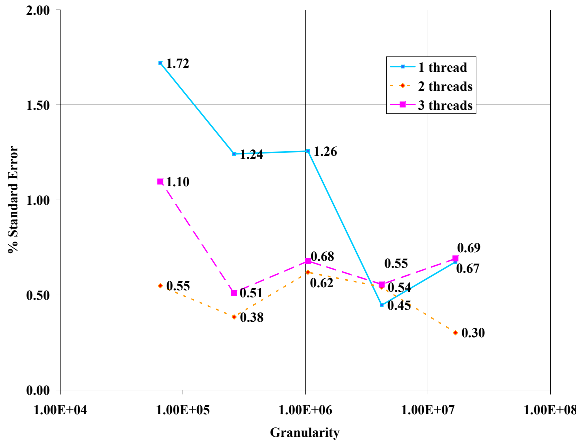
shown in Figure 3(a). It confirms the results of the other two modes, even if the discrepancy between the different number of threads seems much more evident.

## 8. Related Work

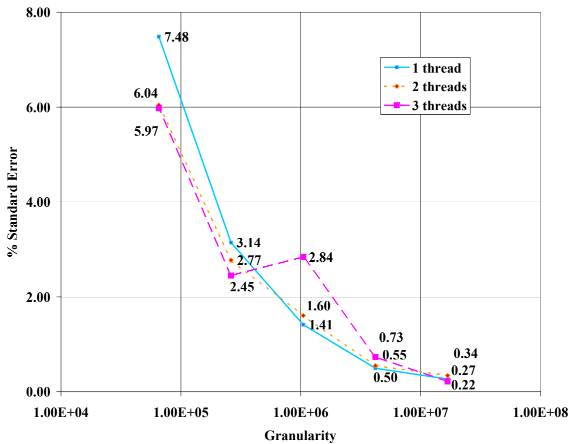
Although many researchers have studied the effect of Java programs on processor-level instruction and data caches [11, 15], as well as the level of repetition of method invocations [10], with the aim of enhancing adaptive compilation schemes, we are aware of no previous publication on the present correlation between bytecodes and CPU time. In [15] the authors mention the number of 25 as the average amount of Sparc machine instructions required for implementing a single bytecode on the older Sun JDK 1.1.6 platform, but this kind of global ratio is obviously too imprecise for our purpose. In [21] a system of vectors is proposed to characterize the performance of each JVM and of each application; these vectors are determined through an extended set of micro-benchmarks, and the achieved prediction quality seems to be fairly good (no precise numbers are provided); however, the JVMs employed are now rather old. In [20], a performance prediction scheme for Java is developed, based on a combination of detailed platform description (including the underlying hardware), of static analysis and of offline calibration of well-chosen subsets of the application code. In contrast, our goal is to try to determine how far it is possible to progress in terms of prediction quality, by following a black-box approach, and, hopefully a simpler methodology.

## 9. Conclusion

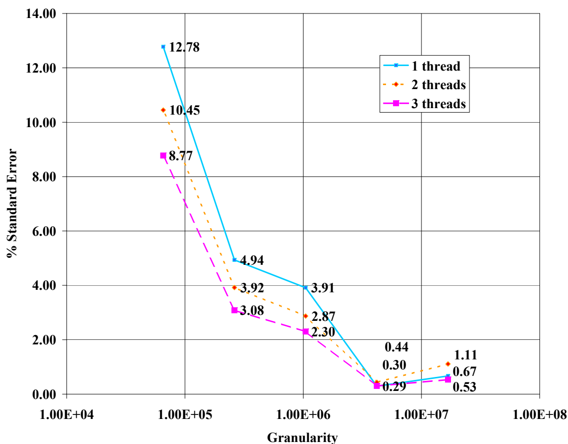
In this paper, we studied the correlation between BIC and CPU time. We showed that for all of the tested platforms there exists a stable, application-specific ratio of bytecodes per unit of CPU time – the *experimental bytecode rate* ($BR_{exp}$) – that can be used for translating a BIC value into the corresponding CPU consumption. Another contribution resides in the description of a new, very fine-grained benchmarking methodology based on portable bytecode instrumentation instead of less precise standard profiling APIs. The validity of our approach is assessed by the establishment of similarities with the performance of non-instrumented programs run on the same platforms.




(a) Xint mode




(b) Client mode




(c) Server mode

Figure 3: Granularity vs. % Standard Error with Sun JDK.

## References


[1] R. Berrendorf and H. Ziegler. PCL – The Performance Counter Library: A common interface to access hardware performance counters on microprocessors (version 2.2). Technical report, Central Institute for Applied Mathematics, Research Centre Juelich GmbH, 2003. `http://www.fz-juelich.de/zam/PCL/`.

[2] W. Binder. A portable and customizable profiling framework for Java based on bytecode instruction counting. In *Third Asian Symposium on Programming Languages and Systems (APLAS 2005)*, volume 3780 of *Lecture Notes in Computer Science*, pages 178–194, Tsukuba, Japan, Nov. 2005. Springer Verlag.

[3] W. Binder and J. Hulaas. A portable CPU-management framework for Java. *IEEE Internet Computing*, 8(5):74–83, Sep./Oct. 2004.

[4] W. Binder and J. Hulaas. Using bytecode instruction counting as portable CPU consumption metric. In *QAPL'05 (3rd Workshop on Quantitative Aspects of Programming Languages)*, volume 153(2) of *ENTCS (Electronic Notes in Theoretical Computer Science)*, Edinburgh, Scotland, Apr. 2005.

[5] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. *ACM SIGPLAN Notices*, 35(5):13–26, May 2000.

[6] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. *ACM SIGPLAN Notices*, 38(11):149–168, Nov. 2003.

[7] J. Dujmovic and C. Herder. Visualization of Java workloads using ternary diagrams. *Software Engineering Notes*, 29(1):261–265, 2004.

[8] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java language specification*. Java series. Addison-Wesley, Reading, MA, USA, second edition, 2000.

[9] D. Gregg, J. Power, and J. Waldron. Benchmarking the java virtual architecture, April 2002.

[10] D. Gregg, J. F. Power, and J. Waldron. A method-level comparison of the java grande and spec jvm98 benchmark suites. *Concurrency and Computation: Practice and Experience*, 2005.

[11] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 251–269. ACM Press, 2004.

[12] J. Hulaas and W. Binder. Program transformations for portable CPU accounting and control in Java. In *Proceedings of PEPM'04 (2004 ACM SIGPLAN Symposium on Partial Evaluation & Program Manipulation)*, pages 169–177, Verona, Italy, August 24–25 2004.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[14] T. Ogasawara, H. Komatsu, and T. Nakatani. A study of exception handling and its dynamic optimization in Java. *ACM SIGPLAN Notices*, 36(11):83–95, Nov. 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

[15] R. Radhakrishnan, J. Rubio, and L. K. John. Characterization of java applications at bytecode and ultra-sparc machine code levels. In *ICCD '99: Proceedings of the 1999 IEEE International Conference on Computer Design*, page 281, Washington, DC, USA, 1999. IEEE Computer Society.

[16] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of java workloads: a structured view and opportunities for optimizations. In *SIGMETRICS 2001/Performance 2001*, pages 194–205, Cambridge, MA, June 2001.

[17] Sun Microsystems, Inc. Java Virtual Machine Profiler Interface (JVMPI). Web pages at `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/`.

[18] The Standard Performance Evaluation Corporation. SPEC JBB2005 (Java Business Benchmark). Web pages at `http://www.spec.org/osg/jbb2005/`.

[19] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. Web pages at `http://www.spec.org/osg/jvm98/`.

[20] J. D. Turner. *A Dynamic Prediction and Monitoring Framework for Distributed Applications*. Phd thesis, Department of Computer Science, University of Warwick, UK, May 2003.

[21] X. Zhang and M. Seltzer. Hbench:java: An application-specific benchmark for jvms. In *the ACM 2000 Java Grande Conference, San Francisco, CA, USA*, June 2000.