# SPECjbb2005 – A Year in the Life of a Benchmark

**Alan Adamson, IBM Canada Ltd., David Dagastine, Sun Microsystems, and Stefan Sarne, BEA Systems**

***Abstract*—Performance benchmarks have a limited lifetime of currency and relevance. This paper discusses the process used in updating SPECjbb2000 to SPECjbb2005 and presents some initial reflections on the implications and effects of the update now active.**

***Index Terms*—Performance, Benchmarks, SPEC, SPECjbb2000, SPECjbb2005**

## I. Benchmarks and their Role

Performance benchmarks play a number of roles.

For computer system customers, they can serve as tools to help make purchasing decisions, such as choosing what vendor to favor with hardware or software purchases, or determining how much hardware and software is required for their planned purposes.

Academic users can use this to design and evaluate prototype problem solutions, with the criterion of success being performance change achieved on the given benchmark. Such a criterion may play a key role in having the ideas from the research community taken seriously and implemented in some products.

For hardware or software vendors, there is also a variety of purposes, depending on the role played within the company. A development team will typically use benchmarks in one of two standard ways. Benchmarks will be used to help measure the change in performance of a product across releases, where typically the objective will be not to degrade any key benchmarks, and additionally probably to improve the performance of other benchmarks.

That same development team has likely been working to improve the performance on some key benchmarks, and hence using those benchmarks to guide its design and implementation effort.

Another key stakeholder in the performance benchmarking world is the performance marketing team at a vendor company. That team will be endeavoring to use performance benchmarking results using its products to convince customers to purchase its own products in preference to those of other vendors.

It is because of the importance and consequences of all these stakeholder behaviors that SPEC[1] was formed to emphasize the creation of "realistic, standardized performance tests", reflecting a key realization "that an ounce of honest data was worth more than a pound of marketing hype" (http://www.spec.org/spec/).

One other key realization in SPEC is that benchmarks have a lifetime, and the organization has emphasized regular updating and replacement of existing benchmarks. This paper discusses the concept of the lifetime of a benchmark. We use the specific example of SPECjbb2000, and its successor benchmark SPECjbb2005, released last year, as were all active developers of this successor benchmark.

The next section discusses the lifetime of SPECjbb2000 and its impact on all stakeholders during its lifetime. After that we describe why the Java subcommittee decided a revision for SPECjbb2000 was due, and what the goals of the effort were. We then describe how successfully the new benchmark met its objectives. Finally, we describe the impact of SPECjbb2005 on the marketplace and its stakeholders so far in its short life.

## II. The influence of SPECJBB2000

SPECjbb2000 was SPEC's first offering of a benchmark for server-side Java. It was based on an IBM internal benchmark, pBoB, itself based on an internal earlier benchmark intended to test a C++ runtime library.

The benchmark simulates a three-tier web application, with all of the clients, the middle tier, and the database, running on a single system in a single address space; the database is in-memory, and there is no application server, simply Java application code executing transactions, so the benchmark has quite different characteristics from SPEC's JEE benchmark SPECjAppServer2004 (see http://www.spec.org/jAppServer2004/). 366 SPECjbb2000 results were published in the benchmark's lifetime (24 in 2000, 58 in 2001, 57 in 2002, 52 in 2003, 39 in 2004, 122 in 2005, and 14 in the single review cycle in 2006). Overall

Manuscript received November 28, 2006.

Dr. Alan Adamson works on Java performance in the IBM Toronto Lab, 8200 Warden Ave., Markham, ON, Canada L6G 1C7 (e-mail: adamson@ca.ibm.com).

David Dagastine works on Java performance for Sun Microsystems in Burlington, MA. (e-mail: david.dagastine@sun.com).

Stefan Särne works on Java performance for BEA Systems in Stockholm, Sweden. (e-mail: stefan.sarne@bea.com).

[1] SPEC® and the benchmark names SPECjvm®, SPECjbb® and SPECjAppServer® are registered trademarks of the Standard Performance Evaluation Corporation. All results referred to in this paper are as of January 15, 2007 and can be found at http://www.spec.org.

leading scores were established in June 2000, April 2001, October 2001, January 2002, March 2002, May 2002, August 2002, May 2003, November 2004, January 2005, and January 2006, with scores varying form the initial 80,348 SPECjbb2000 bops, to the final 2,505,420 SPECjbb2000 bops.

While very high scores were largely achieved by the use of very many processors on ever faster hardware, this strategy also created a requirement on the JVMs to be able to support efficient garbage collection on these larger systems, and much development effort saw significant improvement of garbage collectors.

The synchronization requirements in the benchmark saw further refinement of previous earlier strategies implemented by VM teams to mitigate the cost of locking.

On Intel-based systems, the one platform supported by all the three major VM technologies being used in published scores, the publication features a fairly aggressive history of two of the vendors producing successive results leapfrogging one another's results, while the other began to question the real value of the benchmark as measure of customer-relevant performance.

## III. What was wrong with SPECjbb2000

During 2004 there was a recognition that there would be benefit in updating SPECjbb2000 to encourage new JVM improvements, which would be useful for customers of all providers. There were clear problems with SPECjbb2000 that encouraged this move, with a goal of coming closer to real Java application usage.

SPECjbb2000's roots in C++ became more and more visible over the years, as more Java applications also required attention; the problems of optimizing more object-oriented Java code were not well reflected in the efforts spent on SPECjbb2000.

It was unreasonable for the financial calculations in the benchmark to be done in float when Java featured a BigDecimal library, part of the language intended to support exactly such computation. Also, rather than use Java's collection libraries to implement what were the collection types in the benchmark, the existing code used a roll-your-own persistence framework.

There was no XML processing in the benchmark, nor any standard logging of transactions.

SPECjbb2000 was unreasonably parallel in implementation too, as threads run in it with little dependence on shared data. Another thread-related problem was a specific 'fairness' requirement that was becoming more difficult to meet with the appearance of hardware multi-threading and multi-core systems.

There was also a requirement related to hitting peak performance that was becoming problematic, in that rather arbitrary indeterminacies could increase the length of a run unexpectedly; on large systems, this might mean it could take many runs to achieve a satisfactory result, simply adding to already existing problems creating submissions on large systems. Key among those was a simple hard limit on the number of cores used in a run at 128 (and even for that number a penalty in the scoring formula).

There was another serious problem related to garbage collection (especially on on large systems); this was the regular invocation of calls to System.gc() during a benchmark run, but not part of the measured parts of the run, so not being reflected in a final score. These calls allowed vendors to tune garbage collection specifically to this pattern, and hide expensive parts of garbage collection in the System.gc() calls.

A final concern was that on a large system the time to run the benchmark to produce a submission was very long (and this was becoming a serious burden when combined with the possible need to do a few runs to get a sensible result).

## IV. Objectives as we developed SPECjbb2005

The goals when developing SPECjbb2005 were to adjust the benchmark to current standards for a Java industry that has greatly matured. It should use up to date techniques and building blocks, both with representative code and use the extensive runtime libraries available in Java implementations.

A key goal was that it remain a Java server benchmark and so occupy the niche between SPECjvm98 and SPECjAppServer2004 and continue to be a load-and-go benchmark, easy to run and publish with.

A number of items where identified as things that would make us reach this goal.

More object-oriented design with a class and an interface hierarchy to reflect current techniques should be used. This pattern should apply through the various transactions performed in the middle tier and abstract the database tier behind a storage interface that will make it possible to change which map to use in the tier without knowledge from the application.

The internal database structures, for example longStaticBTree, should be replaced with commonly used collection classes in java.util in order to drive vendors to improve the JITs for frequently used code and in order to improve the class libraries themselves.

The required JDK level should change to 5.0, to use the recent standards and the common patterns the new language features introduce.

The representation for monetary calculations should change from float to BigDecimal, targeting both common practice and intended runtime library usage.

JSE standard logging should play a role in the benchmark, to enable easy development of the benchmark and more importantly, to execute code with logging inserted but not enabled, as the industry does. This would also help customers understand the benchmark code, should they care to.

XML processing should be included. Two areas where identified for this, the first is to replace the display screen which is a simulation of terminal output, with an XML document, created with DOM. The second place, which also tried to address the too parallel nature of the benchmark, was to split the database tier from the middle tier and communicate with XML messages via concurrent queues between them.

GC should no longer have “free time”, so System.gc() calls between iterations should be removed, and also warmup time per iteration, to address the pattern that applications are up and running and adjusting as work is loaded on the system. The runtime should also be adjusted, a longer measurement period, to prevent then concept of “lucky runs” is beneficial.

Object tracking done in the persistence layer in SPECjbb2000 should also be removed, since free() isn’t a concept of Java. The concept of multiple JVM mode, to improve benchmarking on large systems, by reducing the runtime, might be included.

## V. SPECjbb2005 Successes and Failures

As with any benchmark development project there are several successes and failures identified after release.

The refactoring of the benchmark to more closely follow the object oriented programming model was a success. A key part of this work was replacing the internal data structures in SPECjbb2000 (most notable the elusive longStaticBTree) with the Java SE collections HashMap and TreeMap. This had several interesting benefits, but most of all optimizations targeting these data structures benefit a large array of real-world applications, not just competitive benchmarks.

Targeting SPECjbb2005 as a Java 5 benchmark was a success. Having Java 5 features as part of the benchmark allowed performance issues surrounding these features to be quickly identified. This also helped to promote the Java platform and was an added requirement for JVM vendors to deliver a Java 5 solution quickly. Potential performance bottlenecks identified by using Java 5 features in SPECjbb2005 continue to have a direct impact on the performance of all Java 5 applications.

The removal of the System.gc() between each measurement interval was a significant success. Major garbage collections and GC performance are now part of the benchmark and prevents JVM vendors from targeting the explicit GC call for aggressive optimizations.

The modification of the benchmark runtime and the notion of expected peak has been a success. The new benchmark runtime with separate warm-up and measurement intervals more closely models the lifecycle of a Java server application and makes the benchmark run more stable and predictable. A favorite amongst Java performance engineers who frequently run this benchmark.

The SPECjbb2005 multi-VM mode can be viewed as both a success and a failure. It is successful at reducing the run time on large systems and providing a usage scenario that reflects customer use cases on large systems. Multiple JVMs have been traditionally used in large batch application deployments and application servers and the multi-VM mode in SPECjbb2005 allowed the benchmark to address this usage scenario.

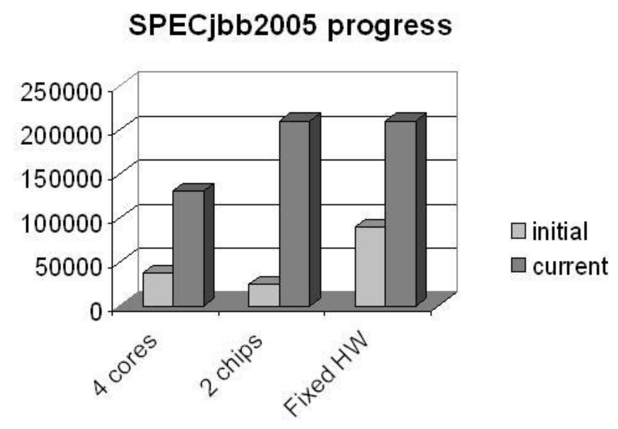
The failure of the multi-VM mode involves how it has been used for submissions. There are many benchmark submissions using the multi-VM mode on small system configurations with as little as 2 CPUs. Multi-VM submissions on small systems removes many aspects of JVM scaling from the workload. Unfortunately many of the submissions on small systems were done to avoid the memory latency overhead suffered by single JVM configurations on heavily NUMA systems. The configuration was also used to avoid GC overhead. The multi-VM mode of SPECjbb2005 provided an easy workaround for these problems and allowed fast benchmark results to be submitted in the near term. The bench-marketing concerns of the multi-VM mode were largely addressed by requiring both the SPECjbb2005 bops and the SPECjbb2005 bops/JVM metric, however the path in which the multi-VM mode has followed is disappointing since significant JVM optimizations have been avoided or postponed because marketing goals have been reached easily through use of the multi-VM mode.

The high allocation rate in SPECjbb2005 is also both a failure and a success. Its a success because it models a common bottleneck in many real-world Java applications. Many Java applications and benchmarks (SPECjappserver2004) have high allocation rates, and including this bottleneck in SPECjbb2005 has allowed JVM developers to address this issue. The high allocation rate is a failure mostly because it can be isolated to a small amount of code and is ripe for aggressive point optimizations that may have little positive impact on customer applications.

## VI. The Year and a Half of SPECjbb2005 Currency

SPECjbb2005 has become an interesting benchmark for two reasons. It has been the common competitive ground for the major JVM vendors and results from different vendors have topped each other over the year, the throne has been passed on more than once. The benchmark has also served an important role in product advertisement, especially for hardware vendors where product launches quotes the SPECjbb2005 score and models are compared with results. The reason it has been able to get this role is that it combines both ease of use and relevance in the workload.

The performance on the benchmark has significantly improved in this one and a half years and the benchmark has been an important driver for this in itself. Comparing four-core results to see the improvement on the benchmark, the first submission was at 37,034 SPECjbb2005 bops, while current greatest 4-core result is at 130,589 SPECjbb2005 bops, an improvement of more than 250%. Another comparison, in one way less apples to apples, but in in another way more relevant, is a 2 socket comparison, both being entry level servers. In this space the first submission was at 24208 SPECjbb2005 bops, while current lead in this category is at 210065 SPECjbb2005 bops, 105033 SPECjbb2005 bops/JVM, which yields a performance increase of more than 750%.



The performance increase is not all JVM performance and to state all was thanks to SPECjbb2005 would be to be even further from the truth. But the truth is that significant JVM performance improvements have been driven by this benchmark, improvements that yield increases on other benchmarks as well, sometimes not as much, sometimes more.

There are several important improvements that have added up to the performance increase, many of them can be found in similar fashions in all JVMs, some in a subset of them. Below a few of them are mentioned with a slight description of them. Garbage collection (GC) improvements are key, with a focus on GC throughput to achieve shorter pause times both for young space and old space collection, driving parallelization of the GC phases and also forcing a tricky balance on what is worth to spend time on inside a collection and what is not, regarding object layout, locality and heap fragmentation. It is interesting to note that both single-spaced and generational collectors have performed well on this workload.

Large page utilization, taking advantage of the possibility to use multiple page sizes in the same process is an ideal optimization for a JVM, since it manages a large amount of memory in process, shared by application threads. By using large pages the TLB cache misses are reduced, which in turn yields performance by reduced stalls. This is a technique used prior to SPECjbb2005, but was made a common option for it and is equally beneficial for workloads outside this benchmark.

The use of a noncontiguous heap, accepting a heap split up in several parts, to be able to use a larger heap on platforms where the address space is split up from the process start, has been of value.

BigDecimal library class improvements to the library class, which specially treat values that fit in a long while possible and then fall back to the default representation if needed, have produced significant improvements in this benchmark, and are of clear general value.

HashMap library class improvements, in order to shorten execution path, have proven to be significant.

Allocation prefetching, a memory enhancement driven by the memory intensive workload with different impact on different architectures, significantly improves performance for applications with high allocation rate.

Locking improvements, reflecting the common case when locks are not contended, allow the Java runtime the possibility to adapt. In this common case a lock is not released until another thread is trying to acquire it. While acquiring a lock at this stage is more expensive, the benefit is that a lock that is acquired again without interruption by the same thread does not need any atomic instructions, an improvement ideal for the multi core world.

Compressed references is another feature that is driven by the memory intensive nature of the benchmark. In order not to have to work with a with a full 64-bits reference on a 64-bit platform, a 32-bits version is used to reduce the memory overhead, which is possibly under the conditions that the heap is less than or equal to the space of what that the reference still can span. This optimization combines the benefits both from a 32-bit platform and a 64-bit platform to some extent.

And next to the explicit features mentioned, friction and more friction are removed by the JITs in this workload, friction that exists in applications out there.

The conclusion is that the benchmark has driven through a lot of changes in only one year, good changes that are here to stay.