Automatic Application Performance Improvements through VM Parameter Modification after Runtime Behavior Analysis

Nicolas Neu
University of New Brunswick
Fredericton, New Brunswick
Canada
nneu@unb.ca

Kenneth B. Kent
University of New Brunswick
Fredericton, New Brunswick
Canada
ken@unb.ca

Charlie Gracie
IBM
Ottawa, Ontario
Canada
charlie_gracie@ca.ibm.com

Andre Hinkenjann
Hochschule Bonn-Rhein-Sieg
Sankt Augustin, North
Rhein-Westphalia
Germany
andre.hinkenjann@h-brs.de

ABSTRACT

This article describes an approach to rapidly prototype the parameters of a Java application run on the IBM J9 Virtual Machine in order to improve its performance. It works by analyzing VM output and searching for behavioral patterns. These patterns are matched against a list of known patterns for which rules exist that specify how to adapt the VM to a given application. Adapting the application is done by adding parameters and changing existing ones. The process is fully automated and carried out by a toolkit. The toolkit iteratively cycles through multiple possible parameter sets, benchmarks them and proposes the best alternative to the user. The user can, without any prior knowledge about the Java application or the VM improve the performance of the deployed application and quickly cycle through a multitude of different settings to benchmark them. When tested with the representative benchmarks, improvements of up to 150% were achieved.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance attributes; D.3.4 [**Programming Languages**]: Processors—*run-time environments,optimization*; D.4.2 [**Operating Systems**]: Storage Management—*garbage collection*

Keywords

Java Virtual Machine, Garbage Collection, Performance Optimizations

VALUETOOLS 2014, December 09-11, Bratislava, Slovakia
Copyright © 2015 ICST 978-1-63190-057-0
DOI 10.4108/icst.valuetools.2014.258196

1. INTRODUCTION

The times of JVM performance being a deterrent when the use of Java was discussed for a new project are long over. Advancements like Just-In-Time compilation or constant improvements of the garbage collection have propelled the JVM up to a point, where it's performance for a wide array of business and scientific applications are on par with their native equivalents. Once thought of as one the greatest weaknesses in the Java ecosystem, performance is now one of the JVM's strong suits. It has been adapted as a runtime environment not only for Java but also for a multitude of other languages, new and old, that compile to Java bytecode. Introducing a new layer of abstraction between the machine and the code might have been disadvantageous at one point, but the benefits developers and administrators gain by having easy access to the runtime environment and it's surrounding tooling outweigh these concern by far. One example is the parametrization of the VM which makes it possible to modify for example the size and layout of the heap or the algorithm used for the garbage collection policy. The IBM J9 JVM, which is used as the reference JVM throughout this article, has about 250 different parameters. Deciding which parameters to choose and what value to set them to for a given application requires intricate knowledge about both the JVM's internals as well as the application itself. This article presents a toolkit that enables users dealing with Java applications to rapidly prototype different VM parameters for an instant performance improvement without having to change programming code or other deployment details.

2. BACKGROUND

2.1 Garbage Collection In A Nutshell

Generally, a runtime providing a garbage collection service is composed of two elements. The mutator and the collector. The mutator is creating the data while the collector is responsible for cleaning up and guaranteeing available memory space. The collection can be performed in two ways:

1. *Stop-The-World:* All mutators have to be stopped and wait for the collector process(es) to finish before resuming work.
2. *Concurrent:* The collection itself or steps preparing the collection can be carried out in parallel to the mutator process.

Stop the World collectors work well with a lot of applications requiring high throughput as the collection time is mostly proportionate to the heap size. Stopping the mutator yet makes them unsuitable for work with systems with low latency requirements. [4]

All collectors independent from their working principle share two properties: Safeness and Completeness. No objects that could still be used at a later point are to be removed and all objects that are not in use anymore will be discarded eventually [3]. Objects still in use are referred to as alive, all other objects are considered dead. Alive objects can be distinguished through the following characteristics [2]:

1. The object is referenced from the stack, a local variable or a static field.
2. The object is referenced by another live object.
3. The object is referenced by the JVM.

2.2 Garbage Collection Mechanisms

2.2.1 Mark-And-Sweep

The garbage collector walks, starting from a root node, over the whole heap and marks all visited objects as alive. During the next step, the heap is traversed again and all non-marked objects are removed. As a third, optional step the heap can also be compacted to avoid fragmentation.

2.2.2 Copying

The memory is divided into two spaces, a From and a To space. After an object in the From space is identified as alive, it is immediately copied to the To space. Then, the roles are switched and the old To space becomes the new From space and vice-versa. Data in the To space is simply overwritten. This approach avoids memory fragmentation in an efficient manner as there is no need to keep track of free memory gaps. Memory gaps with too little space for any object to fit in are also not possible since new data is written in a continuous manner. Constant copying also allows reordering of closely related objects for cache improvements.

2.2.3 Generational

As most young objects that are freshly created are very likely to die and objects that already reached a certain age have a low mortality probability [8], young and old objects are treated differently. Young objects are allocated in the nursery space and are only promoted to tenure space after they reach a sufficient age, i.e. they survived enough gc cycles. The tenure space needs to be collected less frequently.

2.3 IBM J9 JVM Garbage Collection Policies

Currently there are four different garbage collection policies implemented in the IBM J9 JVM from which the user can choose. Each of these implementations have a different working principle and can differ drastically from one another. All of them have been designed with a specific use case in mind.

2.3.1 Gencon

Uses a generational, partially concurrent, copying garbage collection algorithm. Most work is carried out concurrently, only utilizing stop-the-world phases when necessary. A rather inexpensive partial garbage collection only affecting the young space (called scavenge run) is carried out very frequently. Collections of the old space are much less common. The identification of dead objects can be done concurrently, the actual removal happens during a stop-the-world global collection including both the old and young space. Gencon is the default garbage collection policy and well suited for a wide range of applications.

2.3.2 Balanced

Like gencon, the balanced garbage collector segments the heap. However, the memory is divided into a lot more smaller regions, with each region being selfcontained and individually collected. The choice of which region exactly is garbage collected is made at the start of each new cycle and depends on the heap usage pattern of the application. A copying approach is used, with each free region being a potential candidate for a To region. It was designed to reduce performance drops during garbage collection and even exceed the average performance. Accepting a lower overall average performance in exchange for reduced peaks of performance deterioration intends to counter the effect growing heap sizes have on the garbage collector [5]. Bigger heaps mean a larger area for the garbage collector to monitor which results in longer collection pauses.

Similar object, for example objects of the same age are grouped together. Young objects are grouped together in eden regions that are collected more frequently. A halt of all mutator threads is generally required for a collection to take place. Marking of objects however can happen concurrently.

2.3.3 opthruput

This policy uses a nonsegmented heap layout and applies a mark-sweep garbage collection algorithm with an additional compact step if needed. No concurrent operations are taking place, allowing all computational power to be used by the mutator threads. This results in increased throughput for non collection phases. Longer garbage collection pauses are the result, especially for larger heap sizes.

2.3.4 optavgpause

Very similar to optthruput in its working mechanisms. While optthruput relies solely on a stop-the-world approach for its collections, optavgpause performs selected tasks concurrently. The overly long halts of the optthruput algorithm are reduced by this. During execution, GC threads concurrently mark dead objects. The following stop the world phase now takes significantly less time. While overall performance might be worse than with other policies, this policy is especially useful when the application context requires constant throughput. Pause times rarely exceed 0.2 milliseconds. For comparison: Gencon pauses take between one and ten milliseconds, balanced halts the execution regularly for even more than ten milliseconds.

3. APPROACH

To get a set of recommended parameters for the JVM, a rule based approach is used. The ruleset is made up of multiple rules. A rule provides a mapping between observed behaviour and proposed action. A single rule consists of two parts: condition and an action. The condition describes a possible performance problem an application exhibits as well as identifiers or indicators that can be used to activate this rule. The action defines which parameters or combination of parameters should be modified or added in order to avoid those kinds of problems. For parameters that require a quantitative part like the exact size of the heapspace, the algorithm to derive this number has to be part of the action as well. The data required to run these rules against is created by the VM itself in the form of a verbose logfile. This approach can be broken down into these three steps:

1. Execute the application with enabled logging. The J9 *-verbose* parameter family provides information about garbage collection, the heap constitution, class loading, etc.

2. Analyze the generated logfile and infer runtime patterns and possible problems.

3. Change the parameters accordingly and rerun the application. Measure the change of execution time. Repeat this process until desired results are reached.

The iterative aspect of this approach becomes important as the characteristics displayed by the JVM might change significantly with modified settings. For example, switching to a different garbage collection policy brings with it a change in the heap layout and how data allocation and cleanup is handled. More or less heap space is now required. A repeated run might be required to identify the exact extent. Provided that the application does not require any user input during its execution, this process can be completely automated. The application's execution is started through a toolkit which uses the default parameters with logging enabled. After the application terminated, this logfile is sent through a streaming parser extracting relevant characteristics and stores them for later retrieval. In the next step, the rules are applied. Each rule queries against the characteristics to determine whether the activation criteria have been met. If this is the case, the parameter is specified based on the characteristics and added to the parameter set. The application is then run again with the changed parameters. The process is stopped after all relevant parameters have been tried out. If two conflicting rules are triggered, both resulting parameters are added to the parameter set for later evaluation.

4. RULES

This section gives examples for the rules an application is tested for when it is run.

4.1 Choosing the Right Garbage Collection Policy

Given the right environment and testing conditions, each garbage policy is able to outperform the others. There is not a single best policy. Which policy to choose depends on the type of application and what the requirements regarding the performance are. A description of the use cases suited to each policy as well as indicators suggesting a switch to a certain policy are found below.

gencon: The gencon GC policy is suitable for a wide array of different applications for which it provides a good performance. A comparison of 52 benchmark results consisting of the SPECjvm2008 [7] and the dacapo [1] benchmark suite shows, that for varying heapsizes gencon always yielded the most instances of gencon outperforming competing GC policies. The gencon policy should be used for first iterations if the runtime behavior is not yet known.

balanced: Besides being very well suited for large heapsizes as each region can be collected separately over the course of multiple GC cycles, it demonstrates to be effective in the following cases:

- High average collection times can be countered by switching to the balanced garbage collection policy. Avoiding them was one of the main goals during the design of this policy. For the gencon policy, an average global collection time four times higher than the average time spent for a partial collection can be taken as an indicator.

- If the overall time spent in the garbage collection phase is over four percent, a switch to the balanced policy might cut this down.

- Unusually long collection times for single collection can also hint that a switch could be in order. For a segmented policy, a time of more than four seconds spent in a global collection cycle or more than two seconds spent on a partial collection can be considered overly long.

- The region based approach can help when dealing with a high number of large objects. The balanced policy can mitigate a high number of allocation failures caused by large objects. The percentage of allocation failures caused by objects larger than 200Kb can be used as an indicator for this: if it is higher than twenty, the region based approach might be a better fit.

optavgpause and optthruput: It is likely that both of these policies are outperformed by their counterparts using segmented memories. This becomes even more evident with increasing heap sizes. While maybe subpar in terms of overall performance, they excel when the abilities hinted at by their names are required: Low average pause times or a high data throughput with as few interruptions as possible. One exception are applications with a very low runtime. As both policies do not require extra time to set up multiple regions on the heap during startup, applications with a runtime under 2 seconds should be benchmarked with one of these policies. While the optavgpause policy performs slightly better on average, the same application run with the optthruput policy and an adjusted, application specific heap size usually exceeds these numbers.

Besides raw performance numbers, it is important to consider the context in which the application is deployed. Initially highly divergent performance for different garbage collection policies converge when additional measures like heap adjustments are performed. To quantify the degree, how much the performance for all four policies differs when they are fully optimized, the coefficient of variations can be applied. It measures the relative dispersion in a data set. In a test run with over 40 benchmarks, 90% of the benchmark scores had a coefficient of variations of less than 0.1. The results for all four policies are close together. Additional requirements, like consistently low pause times for soft real-time applications (*optavgpause*) can realistically be considered without sacrificing too much performance.

4.2 Finetuning the GC

To measure the effect a rule has on the performance of an application, extensive benchmarking was conducted. All performance benchmarks have been carried out using software from the SPECjvm and dacapo benchmark suites, 44 in total. Both benchmarksuites are comprised of open source applications with realistic, non-trivial workloads that only have been slightly altered to work as a benchmarking software. Every benchmark mimics a real world application that stresses different subsystems of the Java Virtual Machine.

All results are compared to the performance achieved with the JVM running default settings. Since some benchmarks require additional memory, the heap size has been set to one Gigabyte. All other parameters were left at the default setting. A speedup refers to the relative difference between running an application with the optimized parameters compared to running it with the default JVM settings. Not all rules are equally effective for all different kinds of programs. For some types the speedup can be drastic while only minor differences can be noted for other applications that might rely on different rules. When examining the effectiveness of these rules, the average change over all benchmark is noted, as well as the maximum improvement that was achieved.

4.2.1 Adjusting the Heap Size

It seems intuitive that running the JVM with too little memory is detrimental to performance. The garbage collection process has to be triggered unnecessarily often slowing down the whole application. Excessive memory usage can also lead to an application crash if the amount of memory required exceeds the available memory. Performance concerns can also arise in the adverse case with too much heap space available. Garbage collection is avoided for a longer period of time until the whole space is filled up. Subsequent collection cycles now take up more time as a larger space has to be monitored. Setting the heap size to a fixed value in a way, so that at the moment of maximum heap consumption the heap is at a 70% occupancy gives the JVM enough space for dynamic allocations without constantly needing to free memory while staying at an adequate size which can easily be covered by the garbage collector. Setting the heap to a fixed size also relives it from the overhead introduced by automatic memory resizing. Doing this is especially efficient when using the balanced policy. When compared to the performance using default settings, benchmarks could be sped up by up to a factor of 2.82 with an average of 1.22.

4.2.2 Region Sizing

Different applications exhibit different rates of object creation and decay. When using the gencon policy, objects that are discarded quickly after their creation stay in the the nursery without ever being moved to the tenure space. While the JVM has some means of resizing the different segments, its extent is limited. If factors suggesting a high child mortality rate are identified, a manual adjustment can improve the performance by making more room for new objects using formerly unused tenure space memory. There are two indicators to be found in the logs that point to a nonoptimal heap partition:

- *Nursery Space Occupation:* If the nursery's occupation is constantly found to be above 70%, increasing its size becomes necessary. The magnitude of the increase is determined by the occupation and higher occupancies require a bigger increase. In general, it is desirable to keep the maximum occupation around 70% again. Performance improvements of up to 33% were registered for some benchmarks compared to the default settings, with the average improvement of 5%.

- *Tenuring Rate:* The rate of tenuring for a given application can be determined through the following formula:

  $$\frac{data\ copied\ to\ tenured\ space}{data\ occupying\ nursery\ space}$$

  A high value means that large proportions of the nursery are tenured during a collection. Long nursery and frequent tenure space collections are the consequence as many young objects are prematurely tenured. Increasing the nursery space shifts the proportion between young and old area towards a larger young area. If the computed rate exceeds two percent, an increase of the nursery space should be considered. A higher rate requires a bigger increase. A 31% performance increase can be witnessed for selected benchmarks with an average increase of 3%.

It is recommended to combine both measures, resizing the heap as well as readjusting the heap segments when using the generational concurrent policy. When doing so it has to be kept in mind that both parameters are interdependent and one can not be changed without affecting the other. Decreasing the heap size while increasing the nursery can lead to a situation where one component of the heap is larger than the whole heap: The JVM will refuse to start. New values have to be adjusted after all recommendations have been created to retain the proportions between the different components.

4.2.3 Setting the Tenure Age

The tenure age specifies the number of collection cycles an object survives before being promoted to tenure space when using the gencon policy. This value is normally automatically adjusted by the VM over the runtime of the application. It can be set to a fixed value. If the tenure age falls below three, objects are quickly tenured. Setting the tenure age to a fixed low value right from the start of the execution can increase performance as the optimal tenuring strategy is applied earlier. Values of three and lower indicate, that objects are generally tenured very quickly after their creation. Performance improvements of up to 34% were observed after this rule was applied, an average improvement of 3% was achieved.

5. AUTOMATING THE PROCESS

Applying these rules manually is error prone and requires significant effort. By automating this process, there is no need for the user to interact during the benchmarking process. As running an application might take up to several hours, this frees the user to constantly check results and allows the user to focus on other tasks. It is also well suited for the iterative approach as multiple settings can be prototyped successively. It makes it also easier, to run the application repeatedly with the same settings to mitigate possible variations in the benchmark score.

The analysis program is written in Python, using *libxml* as a parsing library and *PySide* to display a graphical user interface. To start the analysis, the user just has to select the executable jar file. The application is then executed multiple times (can be user defined) with logging enabled. After each iteration, the logfile generated by the Java JVM is analyzed to aggregate relevant information. This includes information about the number and frequency of different times of garbage collection, the reasons for the collections and the memory layout at the time of the garbage collection kickoff. The execution time is also recorded. The aggregate data that was created during multiple iterations is now averaged and compared with the rules. If the datapoint lies within the area defined by one of the rules, the corresponding parameter is added to an execution queue where it waits for benchmarking. If for example the average time for all iterations spent in garbage collection is over four percent, a benchmark run with the balanced garbage collection policy will be queued. Adjusting balanced garbage collection specific options will be done once this run has been finished. To remove duplicates, elements that are to be added to the queue are compared to the items in a list of past runs. This avoids running a certain set of parameters multiple times. After the maximum number of different parameter settings is reached, the benchmarking process is stopped. In the result list, the parameters along with the achieved runtime are displayed. The user can now take the parameters with the lowest runtime results to either use it for the deployment of the tested java software or as a starting point for further manual testing and tweaking.

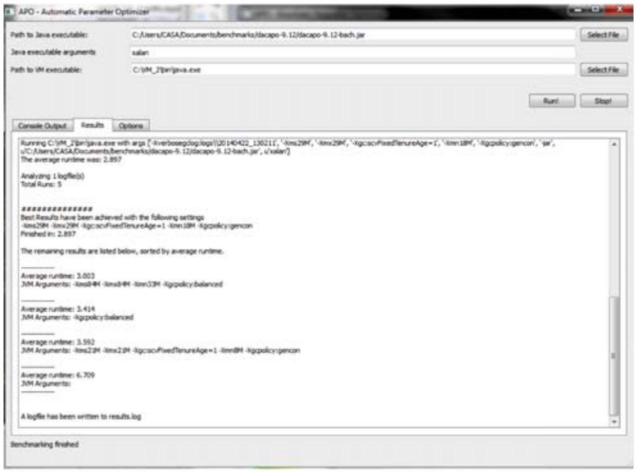
Benchmarks can measure performance in two different ways: Constant time and varying workload or a constant workload with varying execution times. In the first case, the system tries to execute as many operations during a given time frame as possible. The second type measures, how long it takes to perform a predefined number of operations. Since the execution time is used as a performance indicator, only benchmarks belonging to the second category can be used to verify the efficiency of the toolkit. Applications which do not have a predefined set of work which is handled before the applications stops have to be adjusted. A server application would for example need a light wrapper that emulates a predefined workload. After this workload is processed, the application is stopped and the time can be taken. The benchmarks from the dacapo benchmark suite fulfil this requirement. Instead of the actual benchmark score displayed by dacapo [1], the runtime was used as a performance metric which means that set up and tear down times are included in the measurement. This was necessary as to enable the automatic assessment of the runtime characteristics. The results can be found in Table 1. The benchmark scores were obtained by taking the average results of 25 runs. A maximum of 15 different parameter sets were tested. These settings can be changed by the user.

benchmark	default avg.	optimal avg.	improvement	optimal reached
avrora	8.339	7.150	16.6%	2
batik	5.566	4.664	19.3%	11
eclipse	131.739	131.739	0%	1
fop	2.753	2.487	10.7%	11
h2	15.997	13.980	14.4%	3
jython	45.924	44.2491	3.8%	4
luindex	5.8512	5.8512	0%	1
lusearch	6.185	2.467	150.7%	3
pmd	7.294	6.880	6.0%	8
sunflow	7.949	5.591	42.2%	3
tomcat	20.782	17.494	18.8%	15
xalan	3.832	3.061	25.2%	7
Average			25.65%	5.75

Table 1: Performance results for automated tuning evaluation. Average results in seconds. Lower numbers are better. Optimum reached describes the number of runs until an optimal set was found.

For most benchmarks, the best result could be found during the first couple iterations. For three, eleven or more different parameter sets were applied before arriving at the setting that would yield the best performance. Parameters improving the performance can often be found after several iterations. These results are however often improved upon during subsequent iterations. Specifying the number of iterations per run and the maximum number of parameter sets is always a trade off between the time that is spent on the optimization process and the quality of the results. For most applications with a stable execution time five to ten iterations and a maximum number of 15 different parameter sets should suffice to gain satisfying results. If more time can be allotted to the benchmarking process, the results might improve significantly.

Figure 1 shows a screenshot of the application after finishing the benchmarking process for a test application.



Path to Java executable: C:/Users/CASA/Documents/benchmarks/dacapo-9.12/dacapo-9.12-bach.jar

Java executable arguments: xalan

Path to VM executable: C:\VM_2\bin\java.exe

```
Running C:\VM_2\bin\java.exe with args ['-Xverbosegclog:logs\\20140422_130211', '-Xms29M', '-Xmx29M', '-Xgc:scvFixedTenureAge=1', '-Xmn18M', '-Xgcpolicy:gencon', '-jar',
u'C:/Users/CASA/Documents/benchmarks/dacapo-9.12/dacapo-9.12-bach.jar', u'xalan']
The average runtime was: 2.897

Analyzing 1 logfile(s)
Total Runs: 5


###############
Best Results have been achieved with the following settings
-Xms29M -Xmx29M -Xgc:scvFixedTenureAge=1 -Xmn18M -Xgcpolicy:gencon
Finished in: 2.897

The remaining results are listed below, sorted by average runtime.

----------
Average runtime: 3.003
JVM Arguments: -Xms84M -Xmx84M -Xmn33M -Xgcpolicy:balanced

----------
Average runtime: 3.414
JVM Arguments: -Xgcpolicy:balanced

----------
Average runtime: 3.592
JVM Arguments: -Xms21M -Xmx21M -Xgc:scvFixedTenureAge=1 -Xmn8M -Xgcpolicy:gencon

----------
Average runtime: 6.709
JVM Arguments:
----------


A logfile has been written to results.log
```

Benchmarking finished

Figure 1: Screenshot of the parameter optimizing tool after running a set of benchmarks for an application.

6. FUTURE WORK

Future work might include the creation of an application specific database which enables the lookup of a set of optimal parameters for a given platform. Given either the name and version of the application or a list of characteristics obtained during the execution of a Java application, a set of proven and verified parameters are returned that can then instantly be applied to the VM without the need of running the application again.

The toolkit is currently limited to applications for which the performance can be assessed by measuring the elapsed runtime. Applications like servers that do not fall into this category have to be modified so that they can be used. To avoid having to modify an applications code or writing a wrapper that might skew the benchmark results, a plugin like system can be devised that allows for a custom defined performance assessment. This might for example include the parsing of the applications output which could then be translated in specific performance metrics.

Due to the high number of parameters, new rules and combinations of different modifications are very likely to be found. The process of incorporating them into the already existing ruleset has been made as easy and straight forward as possible, so that an all-encompassing set can be created over the course of continued J9 JVM research. This can also include the modification of the JVM to extract additional characteristics which can then be fed into the tuning toolkit. One example is the inclusion of trace events that can record and make JVM execution details accessible [9].

7. CONCLUSION

This article presented a set of rules that can be applied to all applications running on the IBM J9 JVM, changing its behavior and improving the performance. By automatically applying a combination of these rules to a given application, a user can gain performance improvements without the need to touch any existing code or change the current deployment in any other way. Benchmarks showed, that this can result in performance improvements of 150% for selected applications without any interaction necessary.

8. REFERENCES

[1] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann, *The dacapo benchmarks: java benchmarking development and analysis*, SIGPLAN Not. **41** (2006), no. 10, 169–190.

[2] Joshua Engel, *Programming for the java virtual machine*, Addison-Wesley Professional, 7 1999.

[3] Daniel John Frampton, *An investigation into automatic dynamic memory management strategies using compacting collection*, (2003).

[4] Georgios Gousios, Vassilios Karakoidas, and Diomidis Spinellis, *Tuning java's memory manager for high performance server applications*, Proceedings of the 5th International System Administration and Network Engineering Conference SANE 06 (Alexios Zavras, ed.), NLUUG, Stichting SANE, May 2006, pp. 69–83.

[5] IBM, *Garbage collection in websphere application server v8, part 2: Balanced garbage collection as a new option*, `http://www.ibm.com/developerworks/websphere/techjournal/1108_sciampacone/1108_sciampacone.html`, Accessed: 2013-05-30.

[6] IBM, *Java technology, ibm style: Garbage collection policies*, `http://www.ibm.com/developerworks/java/library/j-ibmjava2/`, Accessed: 2013-02-08.

[7] Standard Performance Evaluation Corporation, *Specjvm2008 (java virtual machine benchmark)*, `http://www.spec.org/jvm2008/`, Accessed: 2013-06-010.

[8] David Ungar, *Generation scavenging: A non-disruptive high performance storage reclamation algorithm*, Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments (New York, NY, USA), SDE 1, ACM, 1984, pp. 157–167.

[9] Y. Wang, G. Johnson, and K. B. Kent, *Improving J9 Virtual Machine with LTTng for Efficient and Effective Tracing*, Software: Practice and Experience, `http://dx.doi.org/10.1002/spe.2282`, no. 2282. issn 1097-024X.