Better Performance or Better Manageability?

Mohammad A. Munawar
Shoshin Distributed Systems Group
Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ontario, Canada
mamunawa@shoshin.uwaterloo.ca

Paul A.S. Ward
Shoshin Distributed Systems Group
Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ontario, Canada
pasward@ccng.uwaterloo.ca

ABSTRACT

Competition among software providers creates enormous pressure on design and development teams to improve application performance. However, increased performance leads to systems whose behaviour is harder to predict. This in turn makes software harder to manage, or self-manage in the case of autonomic software. In this paper we elaborate on this problem, first in generic terms, and then taking memory-usage monitoring in a Java Virtual Machine as a specific example. We motivate the need for more research in developing monitoring techniques that can cope with the complexity of modern software systems. We finally present our own efforts in this direction.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling Techniques; D.3.4 [**Software Engineering**]: Processors—*Memory management (garbage collection), Run-time environments*

General Terms

Management, Performance

Keywords

Autonomic Computing, self-management, dynamic systems

1. INTRODUCTION

Historically, information technology has been used to support business-process implementation. In this capacity, the technology improved the efficiency of the relevant business process. The core business would still function if the software failed. Examples of such software include payroll systems, accounts receivable, *etc.* More recently, some businesses have become critically dependent on their information technology infrastructure. That is, their software *is* their business. Typical examples include eBay, Amazon, and Yahoo. These businesses require their systems to be up and running at all time, or they have no business. In spite of this, like all businesses, they desire an infrastructure that is as cost-effective as possible. These systems are usually complex, not only because they are required to adequately service a large population of users, but also because they need to provide increasingly-sophisticated functionality. This critical dependence of businesses on their infrastructure calls for continuous monitoring in order to respond promptly to changes. Traditionally, this problem has been tackled by employing enough human expertise. However, this solution is costly and becoming ineffective. This is explained by the difficulty in finding administrators with skills and knowledge required to cope with the complex nature of these information systems. This problem has lead to a surge in interest in self-managed systems, as reflected by the IBM Autonomic Computing [13] and HP Biologically Inspired Complex Adaptive Systems [1] efforts. While there is a widespread agreement as to the need for such autonomous systems, in practice, designing them has been difficult. In many instances the reason boils down to the problem of capturing the nature of the element to monitor. The difficulty in doing so will keep on growing, as information systems increase in size, functionality, heterogeneity, and distribution.

In this paper we elaborate on the tension that exists between changes that induce complexity and the need for predictability for the sake of manageability. In particular we elaborate on the constant pressure to improve performance of applications and how it impedes administrators' ability to manage them. Due to the competition that rages among software providers, it appears unlikely that performance will be sacrificed for the sake of easy management. Our thesis is that this trend will continue for the foreseeable future and existing approaches are far from sufficient to effectively monitor behaviour of complex software systems. It is thus necessary to develop more-intelligent monitoring solutions for enabling self-management.

2. BACKGROUND

In order to support scalable and distributed business applications, frameworks such as J2EE [3] and .Net [2] have been developed and widely adopted. These frameworks provide commonly-needed services, such as naming, directory, security, transactions, *etc.* These services are typically bundled as servers that can be shared by many applications. These application servers ease developers' work by letting them concentrate on the business logic and provide a robust

platform for the execution of the latter. Many facilities offered by application servers rely on services provided by the underlying operating systems or on virtual machines like the Java Virtual Machine (JVM). To limit the scope of our discussion, we will restrict ourselves to the domain of JVMs, which are critical to the performance of Java-based enterprise applications.

There exists a range of monitoring techniques to predict the performance (*e.g.* response time, throughput, *etc.*) of Java application servers. However, their behaviour has other facets, such as internal state (memory used, number of executing threads, sizes of various resource pools, *etc.*) and correctness. Performance has received a great deal of attention from the research community. The two principal approaches to monitor server performance either rely on past behaviour or use knowledge of the internals of the servers and associated applications. Examples of the first class of techniques include time-series analysis (*e.g.*, [11]), data-mining and pattern-analysis (*e.g.*, [17]), *etc.* Examples of the second approach include employing modeling techniques, such as queuing networks, Petri-nets, Markov chains (*e.g.*, [14, 15]). Some researchers have put forward the idea of using the resulting models to control the behaviour of software, for example by applying control theory (*e.g.* [5, 10]). As far as the correctness aspect is concerned, recently some researchers have started to apply statistical techniques to oversee correctness of a software system (*e.g.* [6]). Monitoring of software state and its evolution has received very limited attention, especially given the high degree of dynamism built in modern software. Examples of early on modeling and prediction resource consumption include work by Devarakonda and Iyer [7].

Application servers are already sophisticated software that are not well-understood by people who manage them. Given the importance of the application server market and thus the availability of many alternatives, fierce competition takes place among the server providers. Higher performance plays an important role in giving one provider a competitive edge over the others. The most popular application servers are therefore shipped with custom-designed JVMs. The resulting push for higher performance, leads to an increasingly complex software system, whose behaviour is difficult to predict at an reasonable performance cost. In the next section we present a concrete instance of this problem and discuss its implications.

3. CASE STUDY

Java application servers rely heavily on many features of the underlying Java language and the Java virtual machine (JVM). One such popular feature is the automatic management of memory (This feature also exists in the .Net framework). It shields the developers from having to explicitly manipulate memory when allocating and freeing objects. Despite this feature, it is still possible for developers to inadvertently keep references to objects that are no longer needed. This may be due to many reasons including lack of understanding of the language or the logic being programmed, intricacies involved when working with application frameworks, such as J2EE, *etc.* These objects would then linger in memory for the lifetime of the application. A garbage collector will not catch such leaks without having some understanding of the semantics of the application. These memory leaks deplete server memory, affecting server performance, and eventually resulting in its halt and of all applications that it supports. This problem represents an important issue and is experienced in production application servers [16]. It is therefore in the administrators' interest to monitor this aspect of an application server. This is especially true because applications running on a server can be updated, added, or removed dynamically. These changes may bring about new leaks or may create new conditions resulting in leaks. If the monitoring system (potentially itself) could detect these leaks, it could take a number of remedial actions. For example, it could inform other parts of the system dedicated for this purpose, or else, it could isolate the application that is causing the leak on a less critical server.

The performance of a JVM is heavily dependent on garbage collection, for a significant amount of time is spent on freeing up previously allocated, but now unused space [4]. In the simplest scheme when an object cannot be allocated for lack of memory, all threads are stopped and a garbage-collector thread is run to reclaim memory occupied by all unreferenced objects. Even this simple scheme makes the task of predicting expected memory usage of an application difficult.

The JVM provides an API to query the amount of free memory available at any time. Knowing the total size of the memory heap reserved by the JVM, we could compute the amount of memory in use at any point in time. This value, however, does not reflect the true amount of memory actually in use. This is due to the fact that used memory comprises objects that have ceased to be referenced and are waiting to be freed. A better alternative could be to collect memory usage information at the end of each garbage collection cycle. Let U_i be the memory in use at time i just after the garbage collection finishes and F_i be the amount of memory freed by the garbage collector. The amount of memory used between two garbage collection points can be computed as $U_{i+1} - U_i + F_{i+1}$. This approach requires us to profile the JVM.

Consider the simple garbage collection scheme mentioned above. If we monitor memory used via the normal JVM runtime interface when some load is being applied to the JVM, we would see a sawtooth pattern, whereby the upward trend corresponds to the increasing size of the memory occupied by objects and the sharp drop maps to the garbage collection. This behaviour can be seen in Figure 2, which depicts memory consumption of a Java application. Developing a predictive model for memory usage of a JVM with this simple garbage collection scheme would require processing the sawtooth signal using non-linear time-series modeling or signal-processing techniques. One way to simplify the problem would be to consider the average time between garbage collections as an indirect indicator of memory-usage evolution. We could learn this metric from past behaviour and predict the length of the next interval. This solution provides an approximation only and still relies on the profiling of the JVM.

The above model does not take into account the load applied to the JVM. Prediction presumes that future load will be comparable with the one seen in the past. In real life temporary irregularities may occur frequently, which would further increase errors of the predictive model. Incorporating the load in the model would require that we know the temporal aspects of the load and also its effect on memory consumption. While it is difficult to accommodate all these

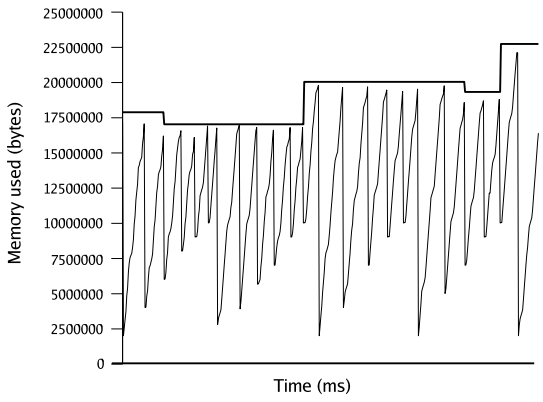

Figure 1: Memory consumption of a JVM without generations

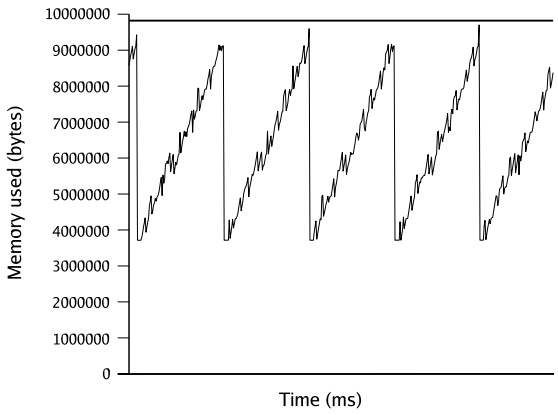

Figure 2: Memory consumption of a JVM with generations

factors into the model, with some efforts it should be possible to develop a model by characterizing the load, profiling the applications, and finding a correspondence between the application functionality exercised and the memory utilized.

Even with simplifications, such a system model is difficult to build. For typical JVMs, this task is even harder because they are much more sophisticated. To improve JVM performance, for example, a significant amount of research has taken place to optimize garbage collection [12]. This has resulted in more efficient garbage collection schemes, which have lead to improved the performance of Java-based applications. For example, based on the observation that many objects allocated in Java applications are short-lived, the use of generations has become part of most JVMs. In this scheme short-lived objects are kept in a young generation, whereas long-lived objects are moved to a tenured generation. Small-scale garbage collections take place more frequently in the young generation, which allows full-scale collections to take place further apart.

To illustrate the difficulty involved in monitoring memory-usage of a JVM, we developed a simple multi-threaded application, in which worker threads allocate a mixture of short-lived and long-lived objects. We then ran the same application on two different JVMs: one that employs generations and one that does not. Figures 1 and 2 show five-second snapshots of the memory usage behaviour for the two JVMs. The upper curve in the two figures depict the amount of memory reserved by the JVM for object allocation. The memory-used value is the difference between the reserved memory and the free memory as reflected by the JVM runtime.

As we can see, the use of generations gives a different picture of the system behaviour in terms of memory consumption. Generational garbage collection makes the task of monitoring and predicting memory-usage at a fine granularity even harder. The memory behaviour depicted in **??** can be traced to the separate behaviour in each generation and thus calls for more elaborate models that can combine different sub-models. Moreover, if we were to use the monitoring scheme where we collect memory usage information at the end of each garbage collection cycle, using a JVM with generations would lead to high overhead. One can now easily imagine how other changes, like incremental garbage collection, could render memory-usage behaviour even more volatile.

As the above example shows, the drive towards faster execution is impeding our ability to know what to expect from the system. Considering our example, simplistic approaches to monitor memory will not give good results at a fine granularity unless our model takes both small-scale and large-scale garbage collections into account.

Beside memory management, we can make similar arguments about other aspects affecting performance in dynamic environments like the JVM. For example, the JVM has evolved to incorporate just-in-time (JIT) compilation. Different providers can develop and tune JIT-related policies to suit characteristics of their software. This again makes predicting application performance at a fine granularity very difficult. Performance is the main factors driving the JVM evolution [9]. JVMs have evolved quickly, but our ability to manage them has not followed the movement. There is a need for bridging this gap.

4. OUR WORK

In order for a system to be able to self-manage, it must know its state, how it reached this state, and what the correct state should be. Our work revolves around modeling system state and evolution of behaviour in dynamic software systems, especially as reflected by its continuous state (*e.g.* via resource consumption information). A concrete example is to devise models that can be employed to characterize the *normal* usage of resources. These models can either be used for resource provisioning purposes or more importantly for anomaly detection. At present, we are looking at memory-usage consumption of Java application servers, in particular IBM WebSphere application server. We are looking at a number of issues in this regard. We are concerned with building workload models that are based on clustering that is sensitive to memory-usage. Similarly, we are interested in workload generators that reflect typical requests seen in production systems. Most of the existing benchmarks have been designed with performance appraisal in mind with the purpose of evaluating peak perform ace. Metrics reflecting the same aspect of a system may differ in quality and cost. We are thus evaluating the effectiveness of various metrics as indicators or approximation of mem-

ory consumption. We are also studying ways to model the internal dynamics of application servers (*e.g.* pool size of threads and database connections, cache sizes, *etc.*) that affect memory-usage and relate these to workload models. To this effect, we have started to look at stochastic models such as Dynamic Bayesian networks (DBN).

Self-management requires that a system has all the pertinent information necessary for it to reason about its state. We are thus also exploring the problem of scalable monitoring. Because data collection for monitoring has an impact on performance, we would like to keep this cost as low as possible when we do not suspect any anomaly. The monitoring system should thus automatically adjust to the appropriate level of data collection given present appraisal of the system. Related work in this area include [8]. Approaches that do a lot more than just adjusting sampling rate are needed in systems where one has the possibility of enabling collection of an arbitrary amount of information (*e.g.* via dynamic byte-code instrumentation). Not only it is imperative to know what is the most relevant information, but also when it should be collected to minimize performance costs. The value of monitored events may differ according to the context. Therefore, it is important to choose the most pertinent information to collect at any given time.

5. CONCLUSION

We believe that it would be unrealistic to try to restrict performance improvement given the very competitive nature of the business applications market. As such, the increasing complexity trend is likely to persist for the foreseeable future. Instead of controlling changes required for higher performance, we need to increase efforts in dealing with the resulting complexity. This in turn requires us to move beyond traditional deterministic modeling techniques, such as queuing networks. One direction thus is to incorporate stochastic models in the mainstream software monitoring tools. Much work has already taken place in the area of machine learning by researchers from various communities. There is a need to apply and adapt existing techniques for software monitoring purposes. In addition, new techniques may need to be developed to accommodate the specific nature of software.

6. ACKNOWLEDGMENTS

This work is supported in part by an IBM Centre of Advanced Studies (CAS) fellowship.

7. REFERENCES

[1] HP Labs. BICAS: Biologically Inspired Complex Adaptive Systems. Available at http://www.hpl.hp.com/research/bicas/.

[2] Microsoft Corp. .NET Platform. Available at http://www.microsoft.com/net/.

[3] Sun Microsystems Inc. J2EE 1.4 Platform Specification. Available at http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf.

[4] Sun Microsystems Inc. Tuning garbage collection with the 1.4.2 Java virtual machine (2003). Available at http://java.sun.com/docs/hotspot/gc1.4.2/.

[5] T. F. Abdelzaher, J. A. Stankovic, C. Lu, R. Zhang, and Y. Lu. Feedback performance control in software services. In *IEEE Control Systems Magazine*, volume 23, June 2003.

[6] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*.

[7] M. Devarakonda and R. Iyer. Predictability of process resource usage: A measurement-based study on UNIX. *IEEE transactions on Software Engineering*, 15:1579–1586, December 1989.

[8] Z. Fu and N. Venkatasubramanian. Adaptive parameter collection in dynamic distributed environment. In *Proceedings of the The 21st International Conference on Distributed Computing Systems (ICDCS)*, 2001.

[9] W. Gu, N. A. Burns, M. T. Collins, and W. Y. P. Wong. The evolution of a high-performing Java virtual machine. *IBM Syst. J.*, 39(1):135–150, 2000.

[10] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004.

[11] J. L. Hellerstein, F. Zhang, and P. Shahabuddin. Characterizing normal operation of a web server: Application to workload forecasting and problem detection. In *Proceedings of Computer Measurement Group*, December 1998.

[12] R. Jones and R. Lins. *Garbage Collection: algorithms for automatic dynamic memory management*. John Wiley and Sons, 1999.

[13] J. Kephart and D. Chess. The vision of Autonomic Computing. In *IEEE Computer*, volume 36, pages 41–50, January 2003.

[14] S. Kounev and A. Buchmann. Performance modelling of distributed E-Business applications using queuing petri nets. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2003.

[15] D. A. Menascé and M. N. Bennani. On the use of performance models to design self-managing computer systems. In *International Computer Measurement Group Conference*, pages 1–9, 2003.

[16] N. Mitchell and G. Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *Proceedings of European Conference on Object-Oriented Computing*, July 2003.

[17] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 2003.