Theory versus Practice in Real-Time Computing With the Java™ Platform

William Foote
Sun Microsystems
EmbeddedJava™ VM Group
bill.foote@eng.sun.com

Abstract

The discipline of Computer Science has always featured a large gap between theory and practice. For example, both the object-oriented paradigm and garbage collection techniques go back to at least the 1960's, [1] [2] but they have only seen widespread commercial acceptance this decade. The Java™ programming language has been an important factor leading to the commercial popularization of these techniques.

The field of real-time computing has suffered from a particularly acute disconnect between theory and practice. Java™ technology can be applied to the building of real-time systems through a set of standard extensions to the Java™ platform. A set of extensions that bring the core benefits of the platform to real-time will advance the commercial state of the art. It will help to narrow the gap between theory and practice.

This paper explores different aspects of computer systems theory as applied to real-time systems. It analyzes if they are ready to be reduced to practice and standardized.

1. Introduction

From the dawn of Computer Science, theory has been well ahead of practice. This can be dismaying at times, but it is natural and to be expected in a field as young as the study of the construction of software systems. Computer programming is at a stage of its evolution where it is still very much more an art than a science. [3]

Academic investigation into the nature of systems building is, of course, of fundamental and profound importance. Due to the nature of the endeavor, those who choose this path can expect many false starts, negative results, and other frustrations. It can be irksome when a good idea goes unnoticed or unadopted by the commercial software community.

Two striking examples of this are automatic memory management techniques ("garbage collection" or GC), and object-oriented design and programming. Each technique has a long academic heritage that goes back to at least the 1960's, [1] [2] but neither saw widespread commercial adoption until the 1990's. The Java™ programming language has contributed to the adoption of both in important ways.

Must all good ideas in Computer Science incubate for over 30 years before they become a widespread part of the practice? One would hope not, but it is important to remember that simple inertia is not the only reason why these techniques did not become widespread. A large motivation for their adoption is that hardware has become powerful enough to compensate for their overhead, and that software complexity (driven in part by higher hardware capacity) has increased in such a way as to require them.

It is tempting to use standardization built around Java™ technology to drive new, experimental ideas to center stage. This would seem to be ill-advised. Standard technologies can advance the commercial state of the art, but they are most successful in areas that are well understood, and around which substantial consensus has been achieved.

This paper examines different ways that the commercial state of the art might be advanced, thus narrowing the gap between theory and practice. It considers these ideas in terms of whether it would be best to address them in a standard for real-time extensions to the Java™ Platform.

2. Object-Oriented Programming

Excellent support for the object-oriented paradigm is so central to the Java programming language and culture that it's easy to overlook. In the world of commercial real-time systems, we find that a large percentage of systems are developed in C, without the benefit of object-oriented design, programming, or language support. A widespread transition to object-oriented techniques in the development of real-time systems will be a paradigm shift, and will advance the state of the art considerably.

Object orientation does not come to real-time without cost. For example, it is more difficult to analyze the worst-case execution time of a system in the presence of polymorphism and extensibility. Despite the difficulties, the Java language's support for object oriented programming is very well understood, and techniques for applying object orientation to real-time systems are well developed. [4]

3. Garbage Collection

Garbage collection is fundamental to the Java programming language and platform. Reliance on GC is pervasive in libraries, and there are many useful things that it is difficult to do without any allocations. The Java platform's memory management policy, built on automatic memory management and no explicit stack allocation, provides many important benefits. It has been pointed out that explicit memory management breaks encapsulation,[5] as any experienced C++ programmer can attest. Additionally, the absence of explicit memory management is necessary for safe pointers. This property, in turn, eliminates two classes of pernicious bugs: dangling references and storage leaks. Furthermore, pointer safety is necessary for security.

Garbage collection presents special challenges when applied to real-time systems. [6] Hard real-time requires that small-grained operations of an algorithm complete within a guaranteed, short amount of time. Most popular garbage collectors cannot guarantee such a bound. For example, a simple mark/sweep GC stops all threads for a time proportional to the total heap size, which introduces large, unpredictable delays (on the order of tens or hundreds of milliseconds, or even seconds). Even advanced generational collection algorithms frequently suspend all threads for tens of milliseconds, and cannot guarantee a hard upper bound on thread suspension.

"Incremental" garbage collection algorithms interleave garbage collection work with the execution of the program, [7] commonly called the mutator. Effective incrementalism helps to bound the maximum pause time the mutator will experience. Unfortunately, incremental algorithms suffer from many limitations. The overhead associated with them is typically large, and can result in up to a 2x or 3x slowdown of the mutator. Incremental techniques typically have high space overheads, as well.

Another problem with incremental techniques is that they usually can't provide a small, constant upper bound on small mutator operations. Eventually, a full root scan must be performed atomically.[6] This is often a function of the total number of threads in the system, the total number of global variables (both native and due to static data members), and other factors that are difficult to analyze. Incremental garbage collection techniques that are implemented purely in software are the realm of research.

Hardware assisted garbage collection techniques provide a potential way out in theory, but their adoption is hindered by high cost. Historically, hardware-assisted techniques have not been economically viable. A programming model that requires hardware assist to effectively support would hinder deployment on off-the-shelf hardware.

Garbage collection is a necessary part of the Java™ platform, but the current state of the art of garbage collection techniques admits no easy solution. It is likely that some kind of compromise will be necessary. This will likely involve some degree of cooperation between the application and the garbage collector, perhaps as outlined in Roger Henriksson's PhD thesis. [8]

4. Real-Time Components

The word "component" is used to mean many things. For this paper, a component is defined to be a self-contained, reusable software unit that can be composed into composite components, or used within an application. Components are usually thought of as black boxes, with well-defined external interfaces.

The Java™ programming language provides extensive support for building and using components. Specific features that enable componentry include encapsulation, reflection, and extensibility. These features work quite well in real-time systems, so in a sense we get real-time components "for free."

Execution time constraints are central to real-time systems. Unfortunately, the time complexity of a computation tends to cut across object-oriented (and component) encapsulation boundaries. Succinctly stated, timeliness breaks encapsulation. To determine the worst-case execution time bound of a computation, the behavioral characteristics of relevant components (and objects) must be considered. Good formalisms for expressing behavioral characteristics are an area for research.

Looked at another way, CPU time requirements are just a special case of a more general problem. In general, it is necessary to analyze the resource requirements of some unit of computation. For example, one can imagine expressing resource requirements via an API. It is not clear that the component is the appropriate level of abstraction for addressing resource requirements in this way. Further, expressing and negotiating resource requirements is not well understood. This is an interesting area for research.

5. Downloaded code

The ability to download pieces of code from a variety of sources is an outstanding property of the Java platform. This extends quite naturally to real-time. The process of actually loading the code might be quite time-intensive, and difficult to analyze. Fortunately, this is usually done during a non-time-critical set-up phase of a system.

One related problem is class initialization. The Java Language Specification states that a class is not initialized until it is used. [9] Class initialization involves running user-provided initialization code, and this code might have side effects. It would be semantically incorrect to run it eagerly.

Software that is not written to take this into account might have surprising results, the first time it is executed during a given program run. This could lead to missed deadlines. Fortunately, it's already possible to address this problem, by ensuring that each time-critical class has an active use [9] during the initialization phase of a program. In the future, a mechanism for more elegantly and conveniently expressing this initialization might be developed.

6. Security

The Java platform provides excellent data security. [10] It has a fine-grained security mechanism that determines access to APIs and system resources based on the protection domain associated with code. The access level associated with a protection domain is determined by the security policy of the runtime, which can be expressed in a text file.

Real-time code will benefit from this security model. It will be an easy matter to ensure that real-time code from a given source can only access resources that it has permission to access. Conversely, the security model makes it easy to ensure that untrusted, non-real-time code does not have access to real-time services, like high priority threads.

The security model relies on checking for permission to carry out certain operations. [11] Many of these checks can be carried out at classloading time, but some of them may have to happen at runtime. If any such checks are necessary in real-time code, it will be important that the overhead of these checks be bounded in time and space.

7. Deadline-based Scheduling

The general topic of deadline-based scheduling takes many forms in the literature. Perhaps the simplest example is earliest deadline first scheduling, or EDF. This is a very simple algorithm, but it has characteristics that are undesirable for many systems. Besides EDF, there are many variants of deadline-based scheduling. It appears that there is not a single approach that it would be appropriate to legislate for all real-time code written in the Java programming language.

The goal of deadline-based scheduling is to capture timeliness requirements in a form that is close to the problem domain. This is laudable, but it is quite a bit more complicated than simple deadlines. Systems based on pure deadline-based scheduling do not degrade gracefully in the presence of overload. To truly achieve the goal, it appears that a more complex discipline is necessary. This might involve utility curves for task completion, or maybe even a formalism for describing other resource requirements. This is an interesting area for research.

The Java Platform's threading model is priority-based. Adding a facility for priority-based threads with semantic guarantees appropriate to real-time is a natural extension that fits well with the rest of the platform. Priority-based scheduling is well understood, and it is the way that most commercial real-time systems are built.

One interesting idea is to provide some kind of low-level scheduling mechanism, which by default provides priority-based scheduling. This mechanism, perhaps some kind of pluggable scheduler, could be extended by the programmer to provide interesting scheduling policies. This is an intriguing idea that merits further study.

8. Summary

This paper discusses real-time extensions to the Java™ platform. Different aspects of computer systems theory are analyzed for the practicality of reducing them to practice in the near term. In this way, we get an interesting perspective on the challenges of applying theory in practice in commercial real-time systems. Some areas that it may not be practical to address in the short term are formalisms for expressing resource requirements, and deadline-based scheduling.

A real-time standard for the Java™ platform would significantly advance the state of the art. It would bring excellent support for the object-oriented paradigm to the real-time programmer. The security, robustness and simplicity of garbage collection and safe pointers would become a reality in real-time systems. Downloaded code, and object-oriented components would be well supported. The strong security guarantees of the platform will transfer naturally. Of course, other benefits of the Java™ platform would also be brought to real-time, such as true binary portability, excellent tool support, and an intuitive and pleasing programming model.

References

[1] H. Schorr and W. Waite. "An efficient machine independent procedure for garbage collection in various list structures." Communications of the ACM, 10(8):501-506, August 1967.

[2] Dahl, O.J. and K. Nygaard "SIMULA- An Algol Based Simulation Language." Communications of the ACM, 9(9):671-678, 1966.

[3] Donald E. Knuth, "The Art of Computer Programming," Volume 1, Addison-Wesley, 1997. ISBN 0-201-89683-4.

[4] Bran Selic, "Animated Structures: Real-Time, Objects, and the UML." Proceedings of the Real-Time Systems Symposium, IEEE Computer Society, Technical Committee on Real-Time Systems, December 2-4, 1998, pp. 165–167.

[5] Bertrand Meyer, "Object-Oriented Software Construction," second edition, pp. 301. Prentice Hall PTR, 1997. ISBN 0-13-629155-4.

[6] Paul R. Wilson, "Uniprocessor Garbage Collection Techniques," Submitted to ACM Computing Surveys,
ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps

[7] Richard Jones and Rafael Lins, "Garbage Collection, Algorithms for Automatic Dynamic Memory Management," John Wiley and Sons, 1996. ISBN 0-471-94148-4.

[8] Roger Henriksson, "Scheduling Garbage Collection in Embedded Systems," PhD Thesis, Department of Computer Science, Lund Institute of Technology / Lund University, Lund, Sweden.

[9] James Gosling, Bill Joy, Guy Steele, "The Java™ Language Specification," section 12.4. Addison Wesley, 1996. ISBN 0-201-63451-1.

[10] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2." In Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California, December 1997, pp. 103–112.

[11] L. Gong and R. Schemers, "Implementing Protection Domains in the Java Development Kit 1.2." In Proceedings of the Internet Society Symposium on Network and Distributed System Security, San Diego, California, March 1998, pp. 125–134.