Steps toward Understanding Performance in Java

Doug Lea
Computer Science Department
State University of New York at Oswego
Oswego NY 13126

Abstract

Java's design goals of portability, safety, and ubiquity make it a potentially ideal language for large-scale heterogeneous computing. One of the remaining challenges is to create performance models and associated specifications and programming constructs that can be used to reason about performance properties of systems implemented in Java.

1 Introduction

Java is the first mass-market concurrent, distributed, object-oriented language. To the extent that heterogeneous computing requires near-universal platform support for a given language ("Write once, run anywhere"), Java is currently the *only* answer for programming non-experimental heterogeneous systems. But is Java a *good enough* answer? Can it become good enough?

Java provides support for the common demands of system-wide heterogeneous computing: Concurrency via threads, locks and monitors; Distribution via Remote Method Invocation (RMI) and related frameworks; User interaction via the AWT; Persistence via serialization and database connections; Mobility via class loaders; and Security via principals, security managers, etc. Across all of these domains, as well as base language constructs, the primary design goals have surrounded portability, safety, historical precedent, and minimality. These goals have been traded off against performance, exploitation of machine-specific capabilities, and availability and real-time guarantees.

2 Performance Models

Reasoning about performance is an integral part of system-level development. Currently, system developers face more extreme versions of the kinds of problems that beset early developers of simple Java applets and applications. Developers could not be confident that a given Java Virtual Machine (JVM) would meet even the most minimal correctness and performance criteria needed for acceptable execution. The widespread deployment of Just-In-Time compilers, dynamic compilation, and more efficient run-time systems have alleviated some of these concerns. Others have been addressed by providing high-level, portable choices for mapping designs to implementations with different performance characteristics. For example, user interface programmers may now choose between "heavyweight" AWT components that are implemented directly by native windowing systems versus "lightweight" components that are implemented mainly in Java proper. Neither is always best with respect to performance and other design criteria.

However, these concerns become much more challenging at a systems level, and have yet to be addressed systematically by JVM implementors. Example issues include:

- How are threads mapped to different processors in SMPs?
- How is persistence mapped to high-performance random access devices (mainly disks), serial devices (mainly networks), transactional processing, etc?
- How is locality exploited in message-based remote communication?
- How is Java synchronization mapped to spinlocks versus JVM scheduling versus kernel scheduling?
- How are known regularities exploited for resource management?
- How can system-wide control and monitoring be extended, for example to include checkpointing and deadlock detection?
- How can soft-real-time requirements be used to influence scheduling?

Java is currently silent about most of these issues, leaving too much freedom in the hands of JVM and Java library and tool implementors, and hence too much uncertainty for developers to be able to reason

about performance. Right now, the only way for developers to deal with this is to build their own custom JVMs, support libraries, and/or tools. While the laxity of Java specifications allows this, it is an unacceptable solution in the long run since it allows developers to reason about performance only on particular implementations.

An alternative is to construct a portable system-level performance model for Java, that is honored by JVM, library, and tool implementors. Aspects of such models have been implicit in most in-the-small performance-related efforts. However, they must be made explicit to scale to systems-level concerns. The heart of a performance model is an abstraction of a computer system providing just enough detail to express mappings and choices among mappings, yet noncommittal enough to apply to JVMs residing on smartcards, supercomputers, and everything in between. Such a model could then be used to provide various styles of rules:

- System-specified mappings: If a capability exists, it will be mapped in a certain fashion.
- Default mappings: Rules that apply unless overridden by programmers.
- Programmer-specified hints: Constructions that allow programmers to heuristically influence or tune parameters of a mapping. These need not take the form of tuning APIs, but may instead for example associate performance properties with different programming constructions.
- Programmer-specified mappings: APIs that allow programmers to plug in control modules and the like.
- Multiple mappings: Different APIs with different performance characteristics, that programmers may choose among.
- Intentional opacity: Reserving the right of implementors to make any mapping choice, unknowable by programmers.

The main challenge is to identify those components of a performance model that significantly impact the ability to reason about performance, yet can be used as the basis of usable, portable, and readily implementable programming constructions. Members of the heterogeneous computing community have much to contribute toward such efforts.

Perhaps in an ideal world, all rules would be of the first type, requiring "optimal" mappings to system capabilities. However, the world is rarely this ideal. For example, the benefit of placing threads on different processors of an SMP generally varies inversely with communication rates among threads. It is hard to imagine placement strategies that would not benefit from information that reveals expected communication rates. Such hints would of course be ignored or used in some other heuristic fashion (for example to help choose between user-level versus kernel-level threads) when programs are run on uniprocessors.

And even in an ideal world, some mappings must remain opaque; for example those that would otherwise reveal information that would compromise safety and security properties.

JVM-level performance models may in turn give rise to application-level models. For example, a common Java programming dilemna surrounds how to map object communication to any of many available forms, including direct method invocations, notifications among threads, JavaBean-style events, structured RMI-style messages, applet-style class transport, serialized mobile-code-style commands, database transactions, and so on. While performance concerns are typically only one factor in such decisions, the ability to approximately predict the performance characteristics of different choices can lead to development of more usable and more useful Java-based systems.