Volume 07 Issue 01 Published, February 19, 2003 ISSN 1535-766X

Intel® Technology Journal

Managed Runtime Technologies

This issue of Intel Technology Journal (Vol. 7, Issue 1, 2003) explores Intel's investigations into the behavior of dynamic managed runtime technologies as they relate to compilers, virtual machines, enterprise applications, performance analysis, security, wireless and mobile applications.

Inside you'll find the following papers:

More information, including current and past issues of Intel Technology Journal, can be found at:
http://developer.intel.com/technology/itj/index.htm

Volume 07 Issue 01 Published, February 19, 2003 ISSN 1535-766X

Intel® Technology Journal

Managed Runtime Technology

Articles

Preface Q1, 2003 ITJ
Lin Chao
Publisher

We often think of the technology business as a competition, a contest to win customers. But it's more than that. It's also a classroom for learning. This is particularly evident in Intel's research into the cross-platform technology known as dynamic, managed runtime technologies. Managed runtimes have an inherent abstraction layer that makes it possible to run on a wide range of devices such as personal computers, cellular phones, digital appliance, smart cards, and network servers. This abstraction layer characteristic carries interesting implications and opportunities.

This issue of Intel Technology Journal (Vol. 7, Issue 1, 2003) explores Intel's investigations into the behavior of managed runtime technologies. Papers in this issue discuss dynamic runtime environments as they relate to compilers, enterprise applications, performance analysis, Java virtual machines, security, wireless and mobile applications.

The three main topics are software technology, applications performance, and wireless mobility. In the first category, the paper by Cierniak, et al, describes the Intel Open Runtime Platform (ORP), an open runtime platform that is used extensively for Intel studies of runtime behavior and as a test-bed for new runtime technologies. It features exact generational garbage collection, fast thread synchronization, and multiple coexisting just-in-time compilers (JITs). A companion paper by Adl-Tabatabai, et al, describes one such technology known as the StarJIT, a just-in-time compiler for both the Java and C# programming languages that generates code for both IA-32 and Itanium® processors.

In the second category, three papers cover applications performance and security. The paper by Chow, et al, discusses the best practices for improving Enterprise Java performance, while the paper by Vorobiov, et al, looks at optimizing Web applications on the ASP .NET platform. Aissi's paper examines the security aspects of runtimes environments, which is another one of their compelling features.

In the third category, two papers look at the wireless mobility space where dynamic runtime environment have become very popular. Comp, et al, look at runtime abstractions for wireless and handheld devices, while Drew, et al, examines runtime environments for high performance mobile devices.

As the Intel Technology Journal begins its 7th year, we thank readers, authors, and referees who have contributed to it. This quarterly web publication is a refereed technical journal, which means that the integrity of each paper is ensured by peer reviews of the papers by recognized Intel experts. The papers, which are authored by the engineers and researchers who are actively working on the technology, are written for the technically aware readership worldwide, and give readers valuable insights into the purpose and intentions behind the technology.

You can read past issues at http://developer.intel.com/technology/itj/archive_new.htm.

You can also subscribe thru a simple registration form at
http://www96.intel.com/cme/showSurv.asp?formID=1019&actv=REG.

Foreword to the Q1 ’03 ITJ Issue on Managed Runtime Technologies

Justin Rattner
Intel Senior Fellow
Director, Microprocessor Research Laboratories

Microprocessor software is currently witnessing the most important behavioral change since the move from assembly language to high-level language programming. The transition to dynamic or so-called managed runtime environments, as exemplified by the Java virtual machine and, more recently, the .NET common language runtime (CLR), are the two most important language developments underlying this change. As a major microprocessor manufacturer, Intel has taken a special interest in these environments as they may lead to new architectural and microarchitectural elements in future designs. The papers in this special issue of the Intel Technology Journal reflect the broad nature of Intel’s investigations and the breadth of impact–from cell phones to clustered application servers–which dynamic runtime environments are having on the computer and communications industries.

A dynamic runtime environment is by no means a monolithic piece of software. While frequently referred to as a language virtual machine, with the Java* virtual machine (JVM) being the most familiar example, the actual virtual machine component is but one element of a modern runtime environment. In addition to the virtual machine, which interpretively executes a high-level, byte-encoded representation of a program, today’s runtimes include a garbage collector and a just-in-time compiler. The garbage collector provides automatic management of the address space by seeking out inaccessible regions of that space (i.e., no addresses point to them) and returning them to the free memory pool. The just-in-time compiler or JIT, as it is called, is used at runtime or install time to translate the byte code representation of a program into native machine instructions, which run much faster than interpreted code. The last-minute translation helps preserve program portability, a key feature of byte code, while maintaining acceptable application performance. Other features such as feedback-guided, dynamic optimization, which improves program performance during execution, are quickly becoming standard components of advanced runtime environments.

Dynamic runtimes are also changing the way programs are written and optimized for different platforms. An application written for a server will most likely deal with a lot of concurrency in the form of many simultaneous transactions. Multi-threading is thus an inherent aspect of server applications. An application written for a cell phone is more concerned about minimizing its memory footprint while providing good performance with limited processor, memory, and communication resources. Learning to deal with these vast differences in program structure and behavior has been a key part of Intel’s effort to fully characterize this new application paradigm.

The papers in this issue of the ITJ illustrate the exciting nature of this rapidly emerging technology, and its impact of both hardware and software design. They also illustrate Intel’s broad effort to fully comprehend these impacts and to apply that knowledge to future processor and platform designs.

The Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment

Michal Cierniak, Microprocessor Research Labs, Intel Corporation
Marsha Eng, Microprocessor Research Labs, Intel Corporation
Neal Glew, Microprocessor Research Labs, Intel Corporation
Brian Lewis, Microprocessor Research Labs, Intel Corporation
James Stichnoth, Microprocessor Research Labs, Intel Corporation

Index words: MRTE, Java, CLI, virtual machine, interface design

ABSTRACT

The Open Runtime Platform (ORP) is a high-performance managed runtime environment (MRTE) that features exact generational garbage collection, fast thread synchronization, and multiple coexisting just-in-time compilers (JITs). ORP was designed for flexibility in order to support experiments in dynamic compilation, garbage collection, synchronization, and other technologies. It can be built to run either Java* or Common Language Infrastructure (CLI) applications, to run under the Windows or Linux operating systems, and to run on the IA-32 or Itanium® processor family (IPF) architectures.

Achieving high performance in an MRTE presents many challenges, particularly when flexibility is a major goal. First, to enable the use of different garbage collectors and JITs, each component must be isolated from the rest of the environment through a well-defined software interface. Without careful attention, this isolation could easily harm performance. Second, MRTEs have correctness and safety requirements that traditional languages, such as C++, lack. These requirements, including null pointer checks, array bounds checks, and type checks, impose additional runtime overhead. Finally, the dynamic nature of MRTEs makes some traditional compiler optimizations, such as devirtualization of method calls, more difficult to implement or more limited in applicability. To get full performance, JITs and the core virtual machine (VM) must cooperate to reduce or eliminate (where possible) these MRTE-specific overheads.

In this paper, we describe the structure of ORP in detail, paying particular attention to how it supports flexibility while preserving high performance. We describe the interfaces between the garbage collector, the JIT, and the core VM; how these interfaces enable multiple garbage collectors and JITs without sacrificing performance; and how they allow the JIT and the core VM to reduce or eliminate MRTE-specific performance issues.

INTRODUCTION

Modern languages such as Java* and C# execute in a managed runtime environment (MRTE) that provides automatic memory management, type management, threads and synchronization, and dynamic loading facilities. These environments differ in a number of ways from traditional languages like C, C++, and Fortran, and thus provide a challenge both for language implementers and for the developers of high-performance microprocessors. This paper concentrates on language implementation challenges by describing a particular MRTE implementation developed at Intel Labs. Other articles in this issue of the *Intel Technology Journal* discuss the implications of MRTEs for microprocessors.

Intel Labs' Microprocessor Research Lab (MRL) has developed an MRTE implementation called Open Runtime Platform (ORP). ORP was designed to support experimentation with different technologies in just-in-time compilers (JITs), garbage collection (GC), multithreading, and synchronization. Over the past five years, researchers have used ORP to conduct a number of MRTE implementation experiments [15-17, 19-21, 23, 25]. At least three different garbage collectors and eight different

* Other brands and names are the property of their respective owners.

® Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

JITs have been developed and integrated with ORP. The version of ORP described in this paper is an internal research tool and is not publicly available.

Three characteristics of MRTEs provide the key challenges to their implementation. First, MRTEs dynamically load and execute code that is delivered in a portable format. This means that code must be converted into native instructions through interpretation or compilation. As a result, MRTE implementations typically include at least one JIT (and often several), and often an interpreter as well. In addition to the challenges of just-in-time compilation, dynamic loading adversely affects important object-oriented optimizations like devirtualization, which reduces the overhead of virtual method calls. Second, MRTEs provide automatic memory management and thus require a garbage collector. Since different applications may impose very different requirements on the garbage collector (e.g., raw throughput versus GC pause time constraints), garbage collector design becomes a significant challenge. Third, MRTEs are multi-threaded, providing facilities for the creation and management of threads, and facilities such as locks and monitors for synchronizing thread execution. The design of efficient locking schemes, given the modern memory hierarchies and bus protocols of microprocessors, is a significant challenge. In addition, the garbage collector must be designed for multiple threads and may very well need to be parallel itself.

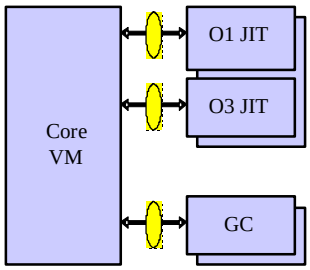


Figure 1: Block diagram of ORP

In order to provide the flexibility needed for JIT and garbage collector experiments, we designed interfaces to cleanly separate the JIT and garbage collector parts of ORP from each other and from the core virtual machine (VM). These interfaces are represented as ovals in Figure 1. Evaluating these experiments requires performance studies, which can be meaningful only if the interfaces impose insignificant overhead. As a result, one of the key contributions of ORP is the design of clean interfaces for JITs and garbage collectors that does not sacrifice performance. The MRTE implementation challenges described above may require cooperation between different components to achieve a good result. For example, devirtualization optimizations may require cooperation between JITs that do the optimization and the core VM that manages the class hierarchy. We had to balance the need for clean interfaces to support flexibility with the need for cooperation to overcome performance hurdles.

In the next section we elaborate on the nature of MRTEs and the challenges they provide to implementers. Then we describe ORP in detail, paying close attention to the design of interfaces that are clean and also lead to high performance.

MANAGED RUNTIME ENVIRONMENTS

In 1995, the Java programming language and the Java Virtual Machine (JVM*) [22] emerged as the first mainstream managed runtime environment (MRTE). In 2000, Java* was joined by Common Language Infrastructure (CLI) [10], and associated languages like C# [9], as the second major MRTE in the market. Both MRTEs have significant differences over C++ compilers and runtimes; yet they are similar to each other in most important ways. In this section, we describe the terminology and key features that distinguish MRTEs from traditional C++ systems, in particular those that may require new optimization techniques to gain full performance.

Key Features

MRTEs dynamically load and execute code. The code and other related data are loaded from *class files*, which can be read from disk, read from a network stream, or synthesized in memory by a running application. Each class file describes a single class, including its superclass, superinterfaces, fields, and methods. Concrete methods include *bytecodes* that specify what to do when that method is invoked. These bytecodes are machine independent, and are at a slightly higher level of abstraction than native instructions. As a result, MRTEs require some means to convert bytecodes into native instructions: an interpreter or a JIT. Because the MRTE controls how bytecodes are converted into native instructions, it may place additional requirements on this conversion that help it to perform functions such as garbage collection and exception throwing, which are discussed below. Because MRTEs tend to produce more

* Other brands and names are the property of their respective owners.

information about compiled bytecodes than just the native instructions, this code is referred to as *managed code*; implementations of native methods and the MRTE itself are called *unmanaged code*.

MRTEs manage type information, that is, they store information about all the classes, fields, and methods that they have loaded, and also about other types that they define or derive automatically, such as primitive and array types. MRTEs provide reflection facilities that allow application code to enumerate and inspect all this information about types, fields, and methods.

MRTEs provide automatic memory management. There is a region of memory belonging to the MRTE called the *heap*. When bytecodes request the instantiation of a class or the creation of an array, space for the new object is allocated in the heap. If the heap is full, the MRTE tries to reclaim the space of objects no longer in use, a process known as *garbage collection* (GC). The part of the MRTE that manages the heap, allocates objects, and performs GC is known as the *garbage collector*.

GC consists of three phases. In the first phase, the garbage collector must find all direct references to objects from the currently executing program; these references are called *roots*, or the *root set*, and the process of finding them all is called *root-set enumeration*. Within one stack frame of managed code, each native instruction may potentially have a different set of roots on the stack and in physical registers; for this purpose, a JIT usually maintains a *GC map* to provide the mapping between individual instructions and roots. In the second phase of GC, the garbage collector finds all objects reachable from the root set, as these might be used in the future; this is called *marking* or *scanning*. In the final phase, the garbage collector reclaims the space of objects not found in the first two phases.

Generational garbage collectors attempt to improve GC efficiency by only scanning a portion of the heap during a collection. Doing so requires additional support from the rest of the MRTE, particularly the JITs: a *write barrier* must be called whenever a reference type pointer in the heap is modified. The write barrier is part of the garbage collector's code and typically does a fast mark of a garbage collector data structure before completing the object field write.

MRTEs provide *exceptions* to deal with errors and unusual circumstances. Exceptions can be thrown either explicitly via a "throw" bytecode, or implicitly by the MRTE itself as a result of an illegal action such as a null pointer dereference. Each bytecode in a method has an associated list of exception handlers. When an exception is thrown, the JVM must examine each stack frame in turn, until it finds a matching exception handler among the list of associated exception handlers. This requires *stack unwinding*, the ability to examine stack frames and remove them from the stack one by one. Note that stack unwinding is also needed to implement security policies and during root-set enumeration, as individual stack frames may also contain roots.

Most of the significant differences between CLI and Java are due to additional features in CLI. Because CLI is largely a superset of Java, it is relatively straightforward to add Java support to an MRTE or JIT compiler that already supports CLI. One addition is that CLI has a richer set of types than Java. Key among these is *value types*. Value types resemble C structures and are especially useful for implementing lightweight types such as complex numbers. CLI also supports *managed pointers* that have many uses, including the implementation of call-by-reference parameters. These may point into the runtime stack, static fields, or into the interior of objects on the heap. These pointers are called "managed" because they must be reported to a garbage collector in order to prevent an object from being prematurely collected. Managed pointers require that a garbage collector properly deal with objects that are only referenced by a pointer into their interior; this means the garbage collector needs a mechanism to locate the start of such an object, based on the interior pointer. Other additions in CLI include support for *unsafe code* that may, for example, operate on pointers and the representation of objects. Such code is often required when accessing legacy libraries. In contrast to Java, CLI objects can be set to be *pinned*, guaranteeing that such objects will not be relocated; pinning may be required for some objects when interfacing with legacy code. CLI also supports a platform library invocation service that automates much of the work involved in calling native library routines.

Optimization Challenges

MRTEs (particularly Java systems) gained an early reputation for not performing as well as traditional languages like C or C++. In part, this reputation arose because the first implementations only interpreted the bytecodes. When JITs were introduced as a way to achieve better performance than interpretation, they were thought of as not optimizing code, but rather as quick producers of native code, with quick startup and response times being the driving requirements. Over time, JIT code quality has increased, due to more mature JIT technology, dynamic recompilation techniques, and a relaxation of the fast startup requirement, particularly for longer-running server-type applications.

Despite the general maturation of JIT technology, there still remain some fundamental issues that separate an MRTE JIT from a traditional C++ compiler. One set of

issues is the lack of whole-program analysis in an MRTE. Classes can be dynamically loaded into the system at any time, and new classes may invalidate assumptions made during earlier compilations of methods. When making decisions about devirtualization, inlining, and direct call conversion, JITs must take into account the possibility that a target method may be overridden in the future (even if at compile time, there is only one possible target), and that a target class may be subclassed (even if the class is currently not extended). This generally results in extra overhead for method dispatch or inlining than would typically be present in a C++ system.

Another set of issues is the safety checks required by MRTE semantics. For example, every array access must test whether the array index falls within the bounds of the array. Every type cast must test whether it is a valid cast. Every object dereference must test whether the reference is null. C and C++ lack these runtime requirements. To achieve competitive performance, therefore, JITs must employ additional techniques to minimize the overhead.

Further performance challenges relate to the garbage collector. Some batch-style applications may demand the highest possible throughput, while other interactive applications may require short GC pause times, possibly at the cost of some throughput. Such requirements have a profound impact on the design of the garbage collector. In addition, since the garbage collector is responsible for mapping objects into specific heap locations, it may also need to detect relationships between objects and ensure that related objects are collocated in memory, in order to maximize memory hierarchy locality.

Some of these JIT-related overheads can be reduced through compiler techniques alone. Others require some level of cooperation with the core virtual machine (VM). Throughout this paper, we identify such techniques and how they are implemented in ORP.

OVERVIEW OF THE OPEN RUNTIME PLATFORM

The Open Runtime Platform (ORP) is a high-performance managed runtime environment (MRTE) that features exact generational garbage collection (GC), fast thread synchronization, and multiple just-in-time compilers (JITs), including highly optimizing JITs. All code is compiled by these compilers: there is no interpreter. ORP supports two different MRTE platforms, Java* [22] and Common Language Infrastructure (CLI) [10].

Basic Structure

ORP is divided into three components: the core virtual machine (VM), just-in-time compilers (JITs), and the garbage collector. The core VM is responsible for class loading, including storing information about the classes, fields, and methods loaded. The core VM is also responsible for coordinating the compilation of methods to managed code, root-set enumeration during GC, and exception throwing. In addition, the core VM contains the thread and synchronization subsystem, although we are planning to split this into a separate component in a future version of ORP. JITs are responsible for compiling methods into native instructions. The garbage collector is responsible for managing the heap, allocating objects, and reclaiming garbage when the heap is full.

ORP is written in about 150,000 lines of C++ and a small amount of assembly code (this includes the core VM code, and excludes the JIT and garbage collector code). It compiles under Microsoft Visual C++ 6.0* and GNU g++, and it runs under Windows (NT/2000/XP*), Linux*, and FreeBSD*. ORP supports both IA-32 [7] and Itanium® processor family (IPF) [8] CPU architectures. ORP uses the GNU Classpath library [1], an open source implementation of the Java class libraries, and OCL [12], an open source implementation of the CLI libraries that is ECMA-335 [10] compliant.

ORP was originally designed with two JITs for Java. The *Simple Code Generator* (known as the *O1 JIT* [15]) produces code directly from the JVM bytecodes [22] without applying complex optimizations. Its optimizations include strength reduction, load-after-store elimination, and simple versions of common-subexpression elimination (CSE), eliminating array-bounds checks, and register allocation.

The *Optimizing Compiler* (known as the *O3 JIT*) converts JVM bytecodes to an intermediate representation (IR) that can be used for more aggressive optimizations. Besides the optimizations performed by the O1 JIT, O3 applies inlining, global optimizations (e.g., copy propagation, dead-code elimination, loop transformations, and constant folding), as well as more complete implementations of CSE and elimination of array-bounds checks.

* Other brands and names are the property of their respective owners.

® Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

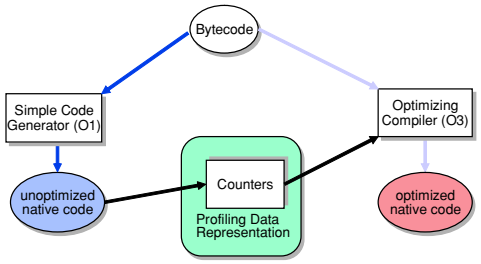


Figure 2: Structure of dynamic compilation

As shown in Figure 2, ORP can run in a mode that uses both the O1 and O3 JITs. In this mode, when a method is invoked for the first time, ORP uses O1 to compile the method in a way that instruments the generated code with counters that are incremented on every method call and on every back edge of a loop. When a counter reaches a predetermined threshold, ORP invokes O3 to recompile the method. The dynamic recompilation approach allows ORP to avoid the cost of expensive optimizations, while applying those optimizations to the methods where the payoff is likely to be high. It also provides the O3 JIT with profiling information that can help guide the optimizations.

ORP also supports a very simple JIT for CLI (currently only on the IA-32 platform), known as the *O0 JIT*. It does no optimizations and was designed for simplicity and to ease debugging. For each CLI bytecode instruction, it generates a sequence of machine instructions that is fixed for each set of operand types.

StarJIT [14] is a new JIT designed to plug into ORP. It supports Java and CLI, and it produces aggressively optimized code for IA-32 and IPF. It translates JVM and CLI bytecodes into a single common intermediate representation on which the rest of StarJIT operates. StarJIT includes an SSA-based optimizer and supports profile-based optimizations as well as dynamic optimizations that are based on continuous profiling and monitoring during program execution.

ORP has supported many different GC implementations over its lifetime, including a simple stop-the-world collector, an implementation of the Train Algorithm [18], and a concurrent collector [19]. There is support in the VM and JIT interfaces for moving collectors (in which objects can be relocated over their lifetimes) and for generational collectors (which require write barrier support from JITs and the core VM). ORP also supports dynamic linking of the GC module, making it possible to select a specific GC implementation via a command-line option.

Common Support for Java and CLI

CLI and Java are semantically similar enough that most of ORP's implementation is common to both runtimes. Both Java and CLI require approximately the same support for class loading, exception handling, threads, reflection, runtime, and low-level (non-library specific) native methods. Of course, CLI uses a different object file format than Java, so the object file loaders are different. Similarly, the class libraries for the two runtimes are different and require a different set of native method implementations. CLI's bytecode instructions are different, so there are differences in the JITs. However, these differences are relatively minor, and most of the code in the StarJIT is common. In general, the significant differences between CLI and Java are due to additional features in CLI. This means if an MRTE (or JIT) supports CLI, it is relatively straightforward to add support for Java.

ORP has relatively few Java-specific or CLI-specific source files beyond those that load classes and those that implement the native methods required by the different CLI and Java class libraries. The MRTE-specific source changes are mostly in short sequences of code that are conditionally compiled when ORP is built. We are currently refactoring ORP to share even more code, which will significantly reduce the need for conditionally compiled code sequences. For example, to indicate an attempt to cast an object to a class of which it is not an instance, a Java MRTE must throw an instance of *java.lang.ClassCastException*, whereas a CLI MRTE must throw *System.InvalidCastException*. Refactoring this part of ORP's implementation simply involves raising the exception stored in a variable that is initialized to the appropriate value.

THE CORE VIRTUAL MACHINE

The core virtual machine (VM) is responsible for the overall coordination of the activities of the Open Runtime Platform (ORP). It is responsible for class loading: it stores information about every class, field, and method loaded. The class data structure includes the virtual-method table (vtable) for the class (which is shared by all instances of that class), attributes of the class (public, final, abstract, the element type for an array class, etc.), information about inner classes, references to static initializers, and references to finalizers. The field data structure includes reflection information such as name, type, and containing class, as well as internal ORP information such as the field's offset from the base of the object (for instance fields) or the field's address in memory (for static fields). The method data structure contains similar information.

These data structures are hidden from components outside the core VM, but the core VM exposes their contents through functions in the VM interface. For example, when a just-in-time compiler (JIT) compiles an access to an instance field, it calls the VM interface function for obtaining the field's offset, and it uses the result to generate the appropriate load instruction.

There is one data structure that is shared across all ORP components, including JITs and garbage collectors, which describes the basic layout of objects. Every object in the heap, including arrays, begins with the following two fields:

```
typedef struct Managed_Object {
    VTable *vt;
    uint32 obj_info;
} Managed_Object;
```

No other fields of the Managed_Object data structure are exposed outside the core VM. The first field is a pointer to the object's vtable. There is one vtable for each class,[1] and it stores enough class-specific information to perform common operations like virtual-method dispatch. The vtable is also used during GC, where it may supply information such as the size of the object and the offset of each reference stored in the instance. The second field, *obj_info*, is 32 bits wide on both IA-32 and Itanium® processor family (IPF) architectures, and it is used in synchronization and garbage collection. This field also stores the instance's default hashcode. Class-specific instance fields immediately follow these two fields.

Garbage collectors and JITs also share knowledge about the representation of array instances. The specific offsets at which the array length and the first element are stored are determined by the core VM and are available to the garbage collector and JITs via the VM interface.

Another small but important piece of shared information is the following. The garbage collector is expressly allowed to use a portion of the vtables to cache frequently used information to avoid runtime overhead. This cached information is private to the garbage collector and is not accessed by other ORP components. Apart from the basic assumptions about object layout and this vtable cache, all interaction between major ORP components is achieved through function calls.

The VM interface also includes functions that support managed code, JITs, and the garbage collector. These functions are described as part of the discussion of the specific components, which we turn to next.

THE JUST-IN-TIME COMPILER INTERFACE

Just-in-time (JIT) compilers are responsible for compiling bytecodes into native managed code, and for providing information about stack frames that can be used to do root-set enumeration, exception propagation, and security checks.

Compilation Overview

When the core virtual machine (VM) loads a class, new and overridden methods are not immediately compiled. Instead, the core VM initializes the vtable entry for each of these methods to point to a small custom stub that causes the method to be compiled upon its first invocation. After a JIT compiles the method, the core VM iterates over all vtables containing an entry for that method, and it replaces the pointer to the original stub with a pointer to the newly compiled code.

The Open Runtime Platform (ORP) allows many JITs to coexist within it. Each JIT interacts with the core VM through the JIT interface, which is described in more detail below, and must provide an implementation of the JIT side of this interface. The interface is almost completely CPU independent (the only exception being the data structures used to model the set of physical registers used for stack unwinding and root-set enumeration), and it is used by both our IA-32 JITs and our Itanium® processor family (IPF) JITs. JITs can be either linked statically or loaded dynamically from a dynamic library.

As previously mentioned in the ORP overview, managed code may include instrumentation that causes it to be recompiled after a certain number of invocations. Another option is to have a background thread that supports recompiling methods concurrently with the rest of the program execution.

Native methods are also "compiled" in the following sense. When a native method is invoked for the first time, the core VM generates a custom wrapper for that native method, and installs it in the appropriate vtables. The purpose of the wrapper is to resolve the different calling conventions used by managed and native code.

[1] Because there is a one-to-one correspondence between a *Class* structure and a vtable, it would be possible to unify them into a single data structure. We chose to separate them to make sure that offsets to entries in the vtable that are used for method dispatch are small, and that instructions generated for virtual method dispatch can be encoded with shorter sequences. Also, the information in vtables is accessed more frequently, so collocating it improves spatial locality and reduces DTLB misses.

® Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Interface Description

The JIT interface consists of a set of functions that every JIT is required to export and a set of functions that the core VM exports. One obvious function in the JIT interface instructs the JIT to compile a method. The JIT interface also includes some not-so-obvious JIT-exported functions that implement functionality that is traditionally thought of as being part of the core VM. These include functions to unwind a stack frame and to enumerate all roots in a stack frame. Stack unwinding is required for exception handling, garbage collection (GC), and security. To allow exact GC, the JIT interface provides a mechanism to enumerate exactly the roots of a stack frame. Given an instruction address, the JIT consults the GC map for that method and constructs the root set for the frame. This is in contrast to some other JIT interfaces such as the Sun JDK 1.0.2* JIT interface [3] that assumes conservative scanning of the stack. Of course, if a conservative collector were used with ORP, this mechanism for root-set enumeration would never be used.

There are two basic solutions to providing stack unwinding and root-set enumeration from the stack:

1. A white-box approach in which the core VM and all JITs agree on a common format for GC maps. At compile time, JITs create GC maps along with native code, and then the core VM can unwind and enumerate without any further help from the JITs.
2. A black-box approach in which each JIT can store GC maps in an arbitrary format understood only by that JIT. Whenever the core VM unwinds the stack or enumerates roots, it calls back into the appropriate JIT for the frame in question, and the JIT decodes its own GC map and performs the operation.

ORP uses the latter scheme, the black-box approach. The advantage of ORP's approach is simplicity and flexibility in JIT design. For example, the O3 JIT supports GC at every native instruction [25], but the simpler O1 JIT only supports GC at call sites and backward branches. This is all possible through the same JIT interface.

Support for Multiple JITs

To support multiple JITs simultaneously, the core VM maintains an array of pointers to *JIT* objects that represent each JIT. The standard ORP/Java/IA-32 configuration includes two statically linked JITs, O1 and O3. Additional JITs may be specified on the command line by supplying the name of a library containing its implementation.

When a method is invoked for the first time, the custom stub transfers control to the core VM, which tries each JIT in turn until one returns success. If no JIT succeeds, ORP terminates with a fatal error.

Core VM Support for JITs and Managed Code

The VM interface includes functions to allocate memory for code, data, and JIT-specific information. The core VM allocates this memory, rather than JITs, which allows the space to be reclaimed when it is no longer needed (however, ORP does not currently implement unloading or GC of methods). The VM interface also includes functions to query the exception information provided in the application class files and to set the exception information for managed code. The core VM uses this latter information during exception propagation.

The core VM also provides runtime support functions for use by managed code. They provide functionality such as throwing exceptions, subtype checks, complex arithmetic operations, and other nontrivial operations.

Optimizations

As mentioned in the section on MRTEs, there are safety requirements and features such as dynamic class loading that can affect the applicability or effectiveness of traditional compiler optimizations. To get performance comparable to unsafe, static languages like C++, JITs must include optimizations that reduce or eliminate safety overheads, and that can work effectively even in the presence of dynamic loading. Some of these optimizations can be implemented entirely in the JITs, but some require cooperation from the core VM. Here we outline some of the key problems and their solutions, along with the additional interface functions that provide the needed cooperation.

Null-check elimination. Java and CLI semantics require null-pointer dereferences to throw an exception. As object dereferences are typically frequent in applications, this safety check might be costly if implemented naively. Compiler analysis can often prove that certain null checks are redundant, and thereby eliminate the checks: many null checks still remain, however.

Some help can be obtained from the core VM. It can instruct the operating system and the hardware to catch null-pointer dereferences and notify the core VM, which can then identify the offending instruction and throw the exception. The IA-32 version of ORP uses this technique, eliminating the need for most null checks by managed code.

The core VM is not always able to assist in this way, though. One frequent example involves devirtualization

* Other brands and names are the property of their respective owners.

of method invocations. A virtual dispatch typically involves dereferencing the object to extract its vtable, which implicitly contains a null reference check. A devirtualized call removes the implicit null check, and thus an explicit check must be added back to the managed code. (If this is not done, then program semantics could be changed if the null-reference exception is never raised, or if it is raised only after some visible side effect that should not have occurred.) In our experience with ORP, the vast majority of these explicit checks can be removed through simple compiler analysis, either by proving that the null check is dominated by a previous explicit or implicit null check, or that an implicit null check happens shortly thereafter, without any intervening side effects.

Array-bounds checking. Java and CLI semantics require out-of-bounds array accesses to throw exceptions. The core VM provides a function that tells JITs, at compilation time, the offset into the array at which the array length is stored, and the JIT is responsible for testing the array index and throwing an exception if necessary. Therefore, managed code does not have to execute a function call to determine the array length.

In a few cases, JITs can prove that all array accesses are within the array bounds. If the array is created within the same scope that it is accessed, it may be possible to symbolically prove that the array index is within bounds. If the application explicitly tests the array index against the bounds (for example, in a loop that explicitly iterates from the lower to the upper bound of the array), then the implicit bounds check can also be eliminated. Unfortunately, such instances of "clean" source code seem to be rare in practice.

In many cases, JITs can eliminate most array bounds checks through "loop cloning." The JIT generates two versions of the loop, one with bounds checks and one without. Loop prolog code is also created that tests the starting and ending conditions of the loop and determines which version of the loop to execute.

Note that both of these techniques are completely within the capabilities of JITs and require no cooperation from the core VM.

Fast subtype checking. Both Java and CLI support single inheritance and, through interfaces, multiple supertypes. An instance of a subtype can be used where an instance of the supertype is expected. Testing whether an object is an instance of a specific supertype is frequent: many thousands of type tests might be done per second during program execution. These type tests can be the result of explicit tests in application code (for example, Java's *checkcast* bytecode) as well as implicit checks during array stores (for example, Java's *aastore* bytecode). These array store checks verify that the types of objects being stored into arrays are compatible with the element types of the arrays. Although *checkcast*, *instanceof*, and *aastore* take up at most a couple of percent of the execution time for our Java benchmarks, that is enough to justify some inlining into managed code. The core VM provides an interface to allow JITs to perform a faster, inlined type check under some conditions that are common in practice.

Direct-call conversion. In ORP, devirtualized calls are still by default indirect calls. Even though the target method may be precisely known, it may not have been compiled yet, or it may be recompiled in the future. By using an indirect call, the managed code for a method can easily be changed after the method is first compiled, or after it is recompiled.

Unfortunately, indirect calls may require additional instructions (at least on IPF), and may put additional pressure on the branch predictor. Thus it is important to be able to convert them into direct calls. To allow this to happen, the core VM includes a callback mechanism to allow JITs to patch direct calls when the targets change due to compilation or recompilation. Whenever a JIT produces a direct call to a method, it calls a function to inform the core VM of this fact. If the target method is (re)compiled, the core VM calls back into the JIT to patch and redirect the call.

Devirtualization and dynamic loading. The O3 JIT performs class-hierarchy analysis to determine if there is a single target for a virtual-method invocation. In such cases, the compiler generates code that takes advantage of that information (for example, direct calls or inlining) and registers that class-hierarchy assumption with the core VM. If the core VM later detects that loading a class violates a registered class-hierarchy assumption, it calls back into the JIT that registered the assumption, to instruct it to deoptimize the code to use the standard dispatch mechanism for virtual methods. This is a variant of guarded devirtualization and does not require stack frame patching (see [17] for more details). The following functions in the JIT interface are used in this scheme:

- *method_is_overridden(Method_Handle m).* This function checks if the method has been overridden in any of the subclasses.
- *method_set_inline_assumption(Method_Handle caller, Method_Handle callee)* This function informs the core VM that the JIT has assumed that *caller* nonvirtually calls the *callee*.
- *method_was_overridden(Method_Handle caller, Method_Handle callee)* The core VM calls this function to notify the JIT that a new class that overrides the method *callee* has just been loaded.

This small set of methods, though somewhat specialized, was sufficient to allow JITs to implement an important optimization without requiring detailed knowledge of the core VM's internal structures.

Fast constant-string instantiation. Loading constant strings is another common operation in Java applications. In our original JIT interface, managed code had to call a runtime function to instantiate constant strings. We extended the interface to reduce the constant-string instantiation at runtime to a single load, similar to a load of a static field.

To use this optimization, JITs, at compile time, call the function *class_get_const_string_intern_addr()*. This function interns the string and returns the address of a location pointing to the interned string. Note that the core VM reports this location as part of the root set during GC.

Because these string objects are created at compile time regardless of which control paths are actually executed, there is the possibility that applying this optimization blindly to all managed code will allocate a significant number of unnecessary string objects. Our experiments confirmed this: performance of some applications degraded when JITs use fast constant strings. Fortunately, the simple heuristic of not using fast strings in exception handlers avoids this problem.

Native-Method Support

ORP gives JITs wide latitude in defining how to lay out their stack frames, and in determining how they use physical registers. As a consequence, JITs are responsible for unwinding their own stack frames and enumerating their roots, and must implement functions for this that the core VM calls. However, since a native platform compiler, not a JIT, compiles unmanaged native methods, the core VM cannot assume any such cooperation. As a result, the core VM generates special wrapper code for most native methods. These wrappers are called when control is transferred from managed to native code. They record enough information on the stack and in thread-local storage to support unwinding past native frames and enumerating Java Native Interface (JNI) references during GC. The wrappers also include code to perform synchronization for native synchronized methods.

In ORP, managed code can interact with native code using one of four native interfaces:

- Direct calls
- Raw Native Interface[2] (RNI)
- Java Native Interface (JNI)
- Platform Invoke (PInvoke)

CLI code uses PInvoke, and Java code uses RNI and JNI. For optimization purposes, native methods may be called directly. RNI, JNI, and PInvoke require a customized wrapper as discussed above. In Java most of the methods use JNI.

Interestingly, we also found JNI methods to be useful for implementing CLI's *internal call* methods. These are methods implemented by the MRTE itself that provide functionality that regular managed code cannot provide, such as *System.Object.MemberwiseClone*.

Native interfaces comparison. JNI and PInvoke are the preferred interfaces and are the only native-method calling mechanisms available to application programmers. However, a few native methods are called so frequently, and their performance is so time-critical, that ORP internally uses either a *direct* call interface or RNI for better performance.

The direct interface simply calls the native function without any wrapper to record the necessary information about the transition from managed code to native code. The lack of a wrapper means that ORP cannot unwind its stack frame. This means that the direct native interface can only be used for methods that are guaranteed not to require GC, exception handling, or security support.

For the PInvoke, RNI, and JNI interfaces, ORP generates a specialized wrapper for each method. This wrapper performs the exact amount of work needed based on the method's signature. This specialization approach reflects the general ORP philosophy of performing as much work as possible at compile time, so that minimum work is required at runtime. The wrapper first saves enough information to unwind the stack to the frame of the managed code of the method that called the native function (described in more detail below), performs locking for synchronized methods, and then calls the actual native method.

RNI and JNI are very similar; the only major difference between them is how references to managed objects are handled. In RNI, references are passed to native code as raw pointers to the managed heap. In JNI, all references are passed as handles. JNI handles incur additional overhead but they make writing and debugging native methods much simpler.

CLI's PInvoke is designed to simplify the use of existing libraries of native code. It supports the look up by name of functions in specified dynamic link libraries (DLLs). It handles the details of loading DLLs, invoking functions with various calling conventions, and marshalling arguments and return values. PInvoke automatically

[2] ORP's implementation of RNI is very close to but not identical to the original Raw Native Interface that is used in the Microsoft Java SDK [4].

translates (*marshals*) between the CLI and native representations for several common data types including strings and one-dimensional arrays of a small set of types.

Stack unwinding for native methods. Unwinding a thread's stack proceeds by first identifying, for each frame, whether it is managed or native. If the frame is managed, the corresponding JIT is called to unwind the frame. Otherwise, the core VM uses a last managed frame (LMF) list to find the managed frame nearest the native frame. Each thread (in thread-local storage) has a pointer to the LMF list, which links together the stack frames of the wrappers of native methods. Included in these wrapper stack frames and the LMF list is enough information to find the managed frame immediately before the wrapper frame, as well as the previous wrapper frame. Also included are the callee-saved registers and the instruction pointer needed to unwind to the managed frame.

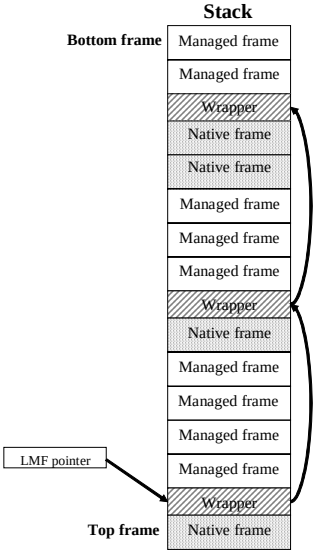


Figure 3: LMF List after the call to a native method

Figure 3 shows a thread stack just after a call to a native method. The thread-local LMF variable points to the head of the LMF list. During unwinding, the LMF list is traversed as each native-to-managed transition is encountered, and the wrapper information is used to unwind past native frames.

JNI optimizations. The core VM generates specialized JNI wrappers to support the transition from managed to native code. The straightforward implementation of these wrappers calls a function to allocate storage and initialize JNI handles for each reference argument. However, most JNI methods have only a small number of reference parameters. To take advantage of this fact, we use an inline sequence of instructions to allocate and initialize the JNI handles directly. This can improve by several percent the performance of applications that make many JNI calls.

Flexibility Versus Performance

For JITs, the performance impact of using interfaces is minimal, since interface functions are called infrequently during program execution. Naturally, the compilation interface is used once for every method that is compiled (including the wrapper generation for native methods), but the number of methods executed is typically orders of magnitude greater than the number compiled, and the compilation cost far exceeds the interface cost. Depending on the application, the number of calls related to exception unwinding and root-set enumeration may be much higher than the compilation-related calls. Once again, though, the cost of performing these operations generally greatly exceeds the cost of using the interface.

THE GARBAGE COLLECTION INTERFACE

The main responsibility of the garbage collector is to allocate space for objects, manage the heap, and perform garbage collection (GC). The GC interface defines how the garbage collector interacts with the core virtual machine (VM) and the just-in-time (JIT) compilers, and it is described in detail below. First we describe the typical garbage collection process in the Open Runtime Platform (ORP).

Overview of Garbage Collection

Typically, when the heap is exhausted, GC proceeds by stopping all managed threads at GC-safe points, determining the set of root references [26], performing the actual collection, and then resuming the threads. A garbage collector relies upon the core VM to enumerate the root set. The core VM enumerates the global references and thread-local references in the runtime data structures. Then it enumerates each frame of each thread stack, and calls the JIT that produced the code for the frame to enumerate the roots on that frame and to unwind to the previous frame.

The garbage collector is also responsible for allocating managed objects. As such, whenever the core VM, managed code, or native methods need to allocate a new object, they call a function in the GC interface. If the heap space is exhausted, the garbage collector stops all managed threads and performs GC as described above.

A generational garbage collector also needs support from the core VM and from managed code to execute a write

barrier whenever a reference field of a managed object is changed. In particular, this requires the JIT to insert calls to the write barrier function in the GC interface into managed code, where appropriate.

Overview of the Interface

Using an interface for GC potentially has a much greater performance impact than using a JIT interface, since a large number of objects are created and garbage-collected during the lifetime of a typical managed runtime environment (MRTE) application. Calling a core VM function to access type information would slow down common GC operations such as object scanning. A common solution to this problem is to expose core-VM data structures to the garbage collector, but this exposure increases the dependency between the garbage collector and the core VM.

The solution in ORP is to expose core-VM data structures only through a call interface (which provides good separation between the core VM and the garbage collector), but to allow the garbage collector to make certain assumptions and to have some space in vtables and thread local storage. In our experience, these non-call parts have been a very important feature of the GC interface. The following sections describe the explicit functions in the GC interface, as well as the implicit data layout assumptions shared between the core VM and the garbage collector.

Data Layout Assumptions

Part of the GC interface consists of an implicit agreement between the core VM and the garbage collector regarding the layout of certain data in memory. There are four classes of memory assumptions in the interface.

First, the garbage collector assumes the layout of objects described previously, in terms of the *Managed_Object* data type. This allows it to load an object's vtable without calling into the core VM. In addition, it can use the *object_info* field for certain purposes such as storing a forwarding pointer while performing GC. However, this field is also used by the synchronization subsystem, so the garbage collector must ensure it does not interfere with those uses.

Second, the core VM reserves space in each vtable for the garbage collector to cache type information it needs during GC. This cached information is used in frequent operations such as scanning, where calling the core VM would be too costly. When the core VM loads and prepares a class, it calls the GC function *gc_class_prepared* so that the garbage collector can obtain information it needs from the core VM through the VM interface and store it in the vtable.

Third, the core VM reserves space in thread-local storage for the garbage collector, and during thread creation it calls *gc_thread_init* to allow the garbage collector to initialize this space. The garbage collector typically stores a pointer to per-thread allocation areas in this space.

Fourth, the garbage collector assumes arrays are laid out in a certain way. It can call a VM function to obtain the offset of the length field in an array object, and for each array type, the offset of the first element of arrays of that type. It can further assume that the elements are laid out contiguously. Using these assumptions, the garbage collector can enumerate all references in an array without further interaction with the core VM. Note that the two offsets can be cached in vtables or other garbage collector data structures.

Initialization

The GC interface contains a number of functions that are provided to initialize certain data structures and state in the core VM and the garbage collector at specific points during execution. These points include system startup, as well as when new classes are loaded and new application threads are created.

At the startup of ORP, the core VM and the JITs call the GC interface function *gc_requires_barriers* to determine what kinds (if any) of write barriers the garbage collector requires. Write barriers are used by some generational, partial collection, and concurrent garbage-collection techniques to track the root sets of portions of the heap even in the presence of updates to those portions. If the garbage collector requires write barriers, then JITs must generate calls to the GC function *gc_write_barrier* after code that stores references into an object field.

As previously mentioned, the core VM calls *gc_class_prepared* upon loading a class, and *gc_thread_init* upon creating a thread. Also, the core VM calls *gc_init* to initialize the garbage collector, *gc_orp_initialized* to tell the garbage collector that the core VM is sufficiently initialized that it can enumerate roots, and thus that GC is allowed, and *gc_next_command_line_argument* to inform the garbage collector of command line arguments.

Allocation

There are several functions related to allocating space for objects. The function *gc_malloc* is the main function, and it allocates space for an object given the size of the object and the object's vtable. There are other functions for special cases such as pinned objects. These allocation functions are invoked by the core VM or by the managed code.

Root-Set Enumeration

If the garbage collector decides to do GC, it first calls the VM function *orp_enumerate_root_set_all_threads*. The core VM is then responsible for stopping all threads and enumerating all roots. These roots consist of global and thread-local object references. Global references are found in static fields of classes, JNI global handles, interned constant strings, and other core VM data structures. Thread-local references are found in managed stack frames, local JNI handles, and the per-thread data structures maintained by the core VM. The core VM and the JITs communicate the roots to the garbage collector by calling the function *gc_add_root_set_entry(Managed_Object**)*. Note that the parameter points to the root, not the object the root points to, allowing the garbage collector to update the root if it moves objects during GC.

After the core VM returns from *orp_enumerate_root_set_all_threads*, the garbage collector has all the roots and proceeds to collect objects no longer in use, possibly moving some of the live objects. Then it calls the VM function *orp_resume_threads_after*. The core VM resumes all threads; then the garbage collector can proceed with the allocation request that triggered GC.

Flexibility Versus Performance

Relatively few interface functions need to be called during GC, largely as a result of the cached type information. However, within managed code, there are potentially many GC interface crossings. The majority of these are object allocation (both of objects and of arrays) and write barriers. The write barrier sequence consists of just a few straight-line instructions with no control flow, and the extra call and return instructions have not proven to be a performance issue in practice. For object and array allocation, the extra call and return instructions are also not a significant source of overhead for MRTE applications (but the same is not true in functional languages). However, if future benchmarks warranted it, the JIT and GC interfaces could be extended to allow inlining of the fast-path of allocation into managed code.

PERFORMANCE OF THE OPEN RUNTIME PLATFORM

For our work to be relevant to other groups that we work with, and to Intel as a whole, the Open Runtime Platform (ORP) must perform as well as commercial Java* virtual machines (JVM*s). As a result, we have put significant effort into designing our interfaces to impose minimal overhead. The purpose of this section is not to provide any in-depth analysis of ORP's performance, but merely to show that ORP is comparable with commercial JVMs on a set of standard benchmarks. A more extensive performance analysis appears in another study [24].

Many commercial JVMs have been developed for the IA-32 platform. A few examples include IBM JDK 1.3.1* [2], Sun HotSpot JDK 1.4.0* [11], and BEA JRockit JVM 1.3.1* [13]. We compare ORP with Sun HotSpot JDK 1.4.0* [11] for SPEC JVM98 [6][3] which is a set of benchmarks that are designed to reflect the workload on a client machine.

The comparison appears in Figure 4. These numbers are taken on a 2.0 GHz dual-processor Pentium® 4 Xeon™ machine without Hyper-Threading, with 1GB of physical memory, and running RedHat Linux 7.2*. We set the initial and maximum heap sizes to the same value of 48 MB for both VMs by using the –Xms and –Xmx command line options.

We are unable to strictly follow the official run rules for these benchmarks because, for example, the Java class library we use, GNU Classpath, does not support AWT and thus cannot run the applets that are required for a conforming SPEC JVM98 run. We have tried to approximate as closely as possible the conditions required for conforming runs within the limits of our research infrastructure. We use unmodified benchmarks, each of which is run from the command line.

Performance numbers are presented in a relative fashion so that the performance of ORP is normalized to 1, and numbers greater than 1 indicate better performance than ORP (the graph shows the inverse of the execution time). ORP was run in its default configuration (all methods were compiled by the O3 JIT), and the only parameter we modified was the heap size.

---

* Other brands and names are the property of their respective owners.

[3] As a research project, the information based on the components of SPEC JVM98 are published per the guidelines listed in the SPEC JVM98 Run and Reporting rules section '4.1 Research Use' (http://www.spec.org/jvm98/rules/runrules-20000427.html#Research). As such these results do NOT represent SPEC JVM98 metrics but only run times and are not directly comparable to any SPEC metrics. Also, as such, enough information is being provided to allow people to reproduce the results.

® Pentium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

™Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

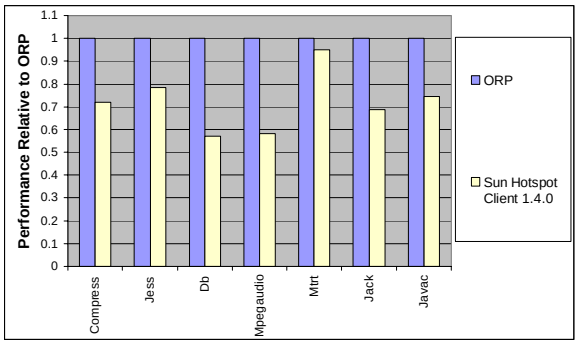


Figure 4. Relative performance to Sun HotSpot Client

ORP performance compares well with Sun HotSpot on these benchmarks. We believe that this performance comparison demonstrates that using interfaces can be consistent with good performance.

CONCLUSION

Along with a general overview of the Open Runtime Platform (ORP), we have described our use of strict interfaces between the core virtual machine (VM) and other components, in particular just-in-time compilers (JITs) and the garbage collector. These interfaces have allowed us and others to construct new JITs and garbage collectors without having to understand or modify the internal structure of the core VM or other components. Contrary to conventional wisdom, we are able to provide this level of abstraction and yet still maintain high performance. The performance cost of using interfaces is minor for the JITs, where interface crossings are infrequent. For the more heavily crossed interface of the garbage collector, we maintain high performance by exposing a small, heavily used portion of the Java object structure as part of the interface and allowing caching of frequently used information. Our experience has shown that this approach is effective in terms of both software engineering and performance.

Our experience with ORP's component design has been positive and has encouraged us to modularize our implementation further. We are currently developing interfaces for other managed runtime environment (MRTE) components such as ORP's threading and synchronization subsystems, to simplify experimentation with other runtime technologies.

ACKNOWLEDGMENTS

This work would not be possible without contributions from the entire Open Runtime Platform (ORP) development team. We also thank ORP users outside of Intel for their contributions and GNU Classpath developers, for providing an open-source class library for Java.

REFERENCES

[1] GNU Classpath, http://www.classpath.org

[2] Jikes Research Virtual Machine. IBM, http://www.research.ibm.com/jikes

[3] The JIT Compiler Interface Specification, Sun Microsystems, http://java.sun.com/docs/jit_interface.html

[4] Microsoft SDK for Java. Microsoft Corp., http://www.microsoft.com/java/

[5] SPEC Java Business Benchmark 2000, Standard Performance Evaluation Corporation, http://www.spec.org/jbb2000

[6] SPEC JVM98, Standard Performance Evaluation Corporation, http://www.spec.org/jvm98

[7] *Intel Architecture Software Developer's Manual*, Intel Corp., 1997.

[8] *IA-64 Architecture Software Developer's Manual*, Intel Corp., 2000.

[9] "C# Language Specification," ECMA-334. ECMA, 2002.

[10] Common Language Infrastructure, ECMA-335. ECMA, 2002.

[11] Java 2 Platform, Standard Edition (J2SE) v. 1.4.1, Sun Microsystems, 2002.

[12] Open CLI Library (OCL). Intel Corp., 2002, http://sf.net/projects/ocl

[13] WebLogic JRockit JVM Version 1.3.1, BEA, 2002, http://commerce.bea.com/downloads/weblogic_jrockit.jsp

[14] Adl-Tabatabai, A.-R., Bharadwaj, J., Chen, D.-Y., Ghuloum, A., Menon, V., Murphy, B., Serrano, M.J. and Shpeisman, T., "StarJIT: A Dynamic Compiler for Managed Runtime Environments," *Intel Technology Journal*, 7 (2003).

[15] Adl-Tabatabai, A.-R., Cierniak, M., Lueh, G.-Y., Parikh, V.M. and Stichnoth, J.M., "Fast, Effective Code Generation in a Just-In-Time Java Compiler," *ACM Conference on Programming Language Design*

and Implementation, Montreal, Canada, 1998, pp. 280-290.

[16] Cierniak, M., Lewis, B.T. and Stichnoth, J.M., "Open Runtime Platform: Flexibility with Performance using Interfaces," *Joint ACM Java Grande - ISCOPE 2002 Conference*, Seattle, Washington, 2002.

[17] Cierniak, M., Lueh, G.-Y. and Stichnoth, J.M., "Practicing JUDO: Java Under Dynamic Optimizations," *ACM Conference on Programming Language Design and Implementation*, Vancouver, British Columbia, 2000.

[18] Hudson, R. and Moss, J.E.B., Incremental Collection of Mature Objects, *International Workshop on Memory Management*, 1992.

[19] Hudson, R. and Moss, J.E.B., Sapphire: Copying GC Without Stopping the World, *Java Grande*, 2001.

[20] Hudson, R., Moss, J.E.B., Sreenivas, S. and Washburn, W., "Cycles to Recycle: Garbage Collection on the IA-64," *International Symposium on Memory Management*, 2000.

[21] Krintz, C. and Calder, B., "Using Annotations to Reduce Dynamic Optimization Time," *ACM Conference on Programming Language Design and Implementation*, 2001.

[22] Lindholm, T. and Yellin, F., *The Java Virtual Machine Specification, Second Edition*, Addison-Wesley, 1999.

[23] Shpeisman, T., Lueh, G.-Y. and Adl-Tabatabai, A.-R., "Just-In-Time Java Compilation for the Itanium Processor," *International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*, Charlottesville, Virginia, 2002.

[24] Shudo, K., "Performance Comparison of JITs," January 2002. http://www.shudo.net/jit/perf

[25] Stichnoth, J.M., Lueh, G.-Y. and Cierniak, M., "Support for Garbage Collection at Every Instruction in a Java Compiler," *ACM Conference on Programming Language Design and Implementation*, Atlanta, Georgia, 1999, pp. 118-127.

[26] Wilson, P.R., "Uniprocessor Garbage Collection Techniques," in revision (accepted for ACM Computing Surveys). ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps

AUTHORS' BIOGRAPHIES

Michal Cierniak is a senior staff researcher in the Programming Systems Lab. He joined Intel in 1997 and has over ten years of experience in compiler design and over six years of experience in managed runtime design. Michal has a Ph.D. degree from the University of Rochester, an M.S. degree from the University of Edinburgh, and an M.S. degree from the Silesian University of Technology. His e-mail address is michal.cierniak@intel.com.

Marsha Eng is a researcher in the Programming Systems Lab. Marsha joined Intel in 2001, with an M.S. degree in Computer Engineering from the University of California, San Diego, and a B.S. degree, also in Computer Engineering, from the University of Washington. Her e-mail address is marsha.eng@intel.com.

Neal Glew is a staff researcher in the Programming Systems Lab. He received a Ph.D. degree in Computer Science from Cornell University in January 2000. His e-mail address is neal.glew@intel.com.

Brian Lewis is a senior staff researcher in the Programming Systems Lab. Brian joined Intel in 2002. He previously worked at Sun, Olivetti Research, and Xerox. While at Sun Microsystems Laboratories, Brian worked on the development of virtual machines for several languages. He also worked on techniques for binary translation as well as portions of the Spring research operating system. Brian received a Ph.D. and M.S. degree in Computer Science and a B.S. degree in Mathematics from the University of Washington. His e-mail address is brian.t.lewis @intel.com.

James Stichnoth is a senior staff researcher in the Programming Systems Lab. Jim joined Intel in 1997, with a Ph.D. degree in Computer Science from Carnegie Mellon University. Jim has worked extensively on both the core VM and the JITs in the Open Runtime Platform (ORP). His e-mail address is james.m.stichnoth@intel.com.

The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments

Ali-Reza Adl-Tabatabai, Microprocessor Research Labs, Intel Corporation
Jay Bharadwaj, Microprocessor Research Labs, Intel Corporation
Dong-Yuan Chen, Microprocessor Research Labs, Intel Corporation
Anwar Ghuloum, Microprocessor Research Labs, Intel Corporation
Vijay Menon, Microprocessor Research Labs, Intel Corporation
Brian Murphy, Microprocessor Research Labs, Intel Corporation
Mauricio Serrano, Microprocessor Research Labs, Intel Corporation
Tatiana Shpeisman, Microprocessor Research Labs, Intel Corporation

Index words: Just-in-time compiler, JIT, Java, Common Language Runtime, virtual machine, dynamic optimization

ABSTRACT

Dynamic compilers (or Just-in-Time [JIT] compilers) are a key component of managed runtime environments. This paper describes the design and implementation of the StarJIT compiler, a dynamic compiler for Java Virtual Machines and Common Language Runtime platforms. The goal of the StarJIT compiler is to build an infrastructure to research the influence of managed runtime environments on Intel architectures. The StarJIT compiler can compile both Java* and Common Language Infrastructure (CLI) bytecodes, and it uses a single intermediate representation and global optimization framework for both Java and CLI. The StarJIT compiler is designed to generate optimized code for the major Intel architectures and currently targets two Intel architectures: IA-32 and the Itanium® Processor Family.

In this paper, we describe the overall architecture (bytecode translators, global optimizer, and code generators) of the StarJIT compiler and the design of its intermediate representation, global optimizer, Itanium Processor Family code generator, and dynamic optimization framework. We present implementation details on the single static assignment (SSA)-based global optimizations [1], the Itanium Processor Family trace scheduler, and the profile-driven dynamic optimization framework.

INTRODUCTION

Programs targeted to managed runtime environments (MRTEs), such as the Java Virtual Machine and the Common Language Runtime, are distributed in a machine-neutral bytecode format and need to be compiled to native machine code by a dynamic compiler. The performance of managed applications depends on the quality of optimizations and code generation performed by the dynamic compiler. Dynamic compilers, or Just-in-Time (JIT) compilers, are thus a key component of MRTEs.

Because final native code generation happens as part of an application's execution, MRTEs pose several challenges to the dynamic compiler:

1. The dynamic compiler must be sensitive to the time and space efficiency of its optimization algorithms – compilation overheads become overheads on the application's execution. For example, a slow compiler can slow down an application's load time, making the system feel less responsive to the user. A dynamic compiler, therefore, must be designed to balance compilation overhead with code quality.
2. Bugs in the dynamic compiler can become security holes that can be exploited by hackers. MRTEs partially rely on the dynamic compiler to enforce security; for example, the dynamic compiler enforces

* Other brands and names are the property of their respective owners.

® Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

memory safety by inserting checks for type casts and out-of-bound array accesses. Bugs in the dynamic compiler can compromise the safety guarantees provided by the MRTE. A dynamic compiler, therefore, must not only be efficient but also robust.

These challenges are particularly difficult for architectures that rely on compiler optimizations for performance. For example, the Itanium® Processor Family architecture relies heavily on expensive and sophisticated code-generation optimizations (such as global scheduling and control speculation) for performance. A dynamic compiler must implement these optimizations robustly and efficiently, and also be flexible, to allow balancing of compilation overhead and code quality.

In comparison to traditional, statically compiled programs, however, MRTEs also provide new performance optimization opportunities:

1. Because native code generation occurs during an application's execution, MRTEs are an ideal environment for dynamic profile-guided optimization. This is important for the Itanium Processor Family, which relies on profile-guided optimizations (such as inlining and trace scheduling) for performance. Dynamic profile-guided optimization also enables the dynamic compiler to concentrate expensive optimizations only on those regions of the program that have the biggest payoffs, thus limiting optimization overhead.
2. The dynamic compiler can tailor the generated code to the platform on which the application is executing. The dynamic compiler can detect platform parameters (such as microarchitecture generation, cache size, and memory size) and tailor the code to the platform parameters. Thus it can deal effectively with "legacy binary" issues.
3. MRTEs provide metadata (such as type information) that can be used for optimization. Metadata gives the compiler precise information about control flow and types used by a program, which the compiler can exploit for optimization (e.g., type-based alias analysis).

We have built the StarJIT compiler as a research infrastructure to investigate these challenges and opportunities on Intel architectures.

The rest of this paper is organized as follows. In the next section, we describe the overall architecture of the StarJIT compiler. We then describe the design of the global optimizer, including the single static assignment (SSA)-based intermediate representation, global optimization phase structure, and SSA-based global optimization algorithms. We then describe the design of the Itanium Processor Family code generator, including the code generation phase structure and trace scheduler.

THE ARCHITECTURE OF THE STARJIT COMPILER

The StarJIT compiler is designed to provide a common strongly typed substrate in which code distributed for various managed runtime environments can be safely optimized and targeted to Intel architectures. A further design goal is to enable dynamic profile-driven optimization and recompilation. These goals are reflected directly in the topological organization of the architecture, illustrated in Figure 1. Paths exist connecting every language front-end with every architecture-specific back-end, propagating type information from the source bytecodes through to the architecture-specific back-ends. Furthermore, an additional path for annotating the intermediate representation (IR) used by the global optimizer with profile information from execution of the generated native code enables the seamless injection and use of dynamic information for recompilation.

If virtual machine (VM) support exists, supporting a new hardware architecture for all of the supported languages requires only that a single StarJIT compiler back-end is implemented for that hardware. Similarly, supporting a new language across the supported Intel architectures requires only that a new language front-end be implemented. The primary architectural features of the StarJIT compiler that enable this are divided into language- and architecture-specific portions and language- and architecture-independent portions, both of which are described in this section.

The process for StarJIT compilation follows a single path in this architectural schema, determined by the source language and target architecture. The managed runtime environment (MRTE) bytecode is translated into the global optimizer's IR by the individual front-ends for each source language supported. The language- and architecture-independent portion comprises the global optimizer and the profile feedback manager. The global optimizer is built on an IR called STIR (StarJIT IR). After optimization, architecture-specific code generators translate STIR into architecture-specific IRs, perform architecture-specific scheduling and register allocation, and finally emit the generated native code. A dynamic feedback loop is created through the use of profile information by the Profile Feedback Manager to selectively recompile and guide global and architecture-specific optimization decisions.

® Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

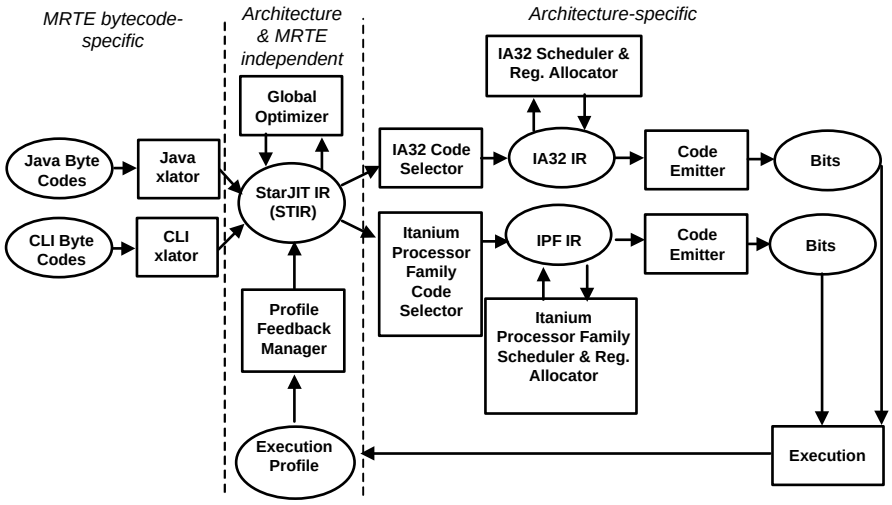


Figure 1: StarJIT compiler architecture

The interface of the StarJIT compiler to a specific MRTE's VM is implemented in a layer that abstracts out the required set of interactions between the JIT and the VM in any MRTE. These include, among other transactions, enumeration of live pointer information for garbage collection, allocation of objects, and metadata queries.

Java* and Common Language Infrastructure Bytecode Translators

The initial compilation step is the translation of portable bytecode into STIR. Currently, the StarJIT compiler has bytecode translator front-ends for Common Language Interface (CLI) and Java*.

Bytecode translation has two phases: the first phase establishes basic block boundaries and exception handling regions, and it recovers type information for variables and operators. There are two major differences in the type information contained in the CLI and Java bytecodes. First, CLI variables are annotated with exact type information whereas Java variables do not have a fixed type and may be reused with different types at different points in the program. Second, CLI operators are untyped whereas Java operators are typed. The first phase of translation reconciles these differences to generate type information for both variables and operators: the Java translator performs type propagation to recover type information for variables, and the CLI translator performs type propagation to recover type information for the operators.

The second translation phase generates STIR and performs simple optimizations, including inlining, constant and copy propagation, folding, strength reduction, type check elimination, devirtualization, elimination of class initialization checks, and value numbering-based redundancy elimination across extended basic blocks.

The bytecode translators generate low-level operators to expose as many calculations as possible to the later global optimization phase. For example, a load of an object field is broken up into component operations that perform a null check of the object reference, load the base address of the object, compute the address of the field, and load the value at that computed address. The front-end translators, however, can be configured to use higher-level operators, which minimizes the need for later-stage coalescing, to take advantage of IA-32's rich addressing modes.

STIR: The StarJIT Compiler's Intermediate Representation

The StarJIT compiler's intermediate representation (IR) (STIR) is a traditional two-level IR, with control-flow represented as a graph and instructions represented as triples [16].

At a high level, STIR is a control flow graph consisting of nodes and edges. The StarJIT compiler also maintains dominator and loop structure information on this level of IR for use in optimization and code generation. STIR represents both conventional control flow due to jumps and branches, and exceptional control flow due to thrown and caught exceptions, so that the global optimizer and code generators both account for and optimize exceptions and exception handlers. STIR models conventional control flow via basic block nodes and edges, which represent jumps and conditional branches between basic

* Other brands and names are the property of their respective owners.

block nodes. STIR models exceptional control-flow via dispatch nodes: a thrown exception is represented by an edge from a basic block node to a dispatch node, and a caught exception is represented by an edge from a dispatch node to a block node.

In managed runtime implementations, compiler-generated code generally does not implement exceptional control flow. Instead, the underlying system implicitly handles the exception throws and catches. The StarJIT compiler generates a system call instruction for each throw and registers a handler for each catch. By modeling exceptional control flow explicitly in the control flow graph, the compiler can optimize across throw-catch boundaries. For locally handled exceptions, the compiler replaces expensive throw and catch combinations with cheaper direct branches.

At a lower level, each basic block node consists of a list of instructions, where each instruction is a tuple consisting of an operator and a set of static single assignment (SSA) operands [10]. The operators are low level in order to expose finer-grain operations to the optimizer. SSA form provides explicit use-def links between operands and their defining instructions, which simplifies and speeds up global optimizations. STIR is designed to address both exclusive and dissonant implementation semantics of Java and CLI.

Each STIR instruction and operand is annotated with detailed type information. STIR instructions retain all type information explicit or implicit in the original Java and CLI bytecodes. Optimization passes preserve and update this type information, and they propagate it through to the architecture-specific back-ends for their use. Type information is needed in the code generator to support exact garbage collection (GC), which requires enumeration of the root set at GC safe points. Type information also greatly improves the quality of the compiler analyses by enabling type-based memory disambiguation at various optimization and code-generation stages.

Itanium® Processor Family and IA-32 Code Generators

The StarJIT compiler currently supports both the Itanium® Processor Family and IA-32 family architectures through distinct back-end code generators. The compiler enables adaptation of new code generators, such as for the Intel® XScale™ family, through a software interface that allows the optimizer to transparently perform the necessary callbacks to the code generator to construct each code-generator's IR with the appropriate type information.

The propagation of STIR type information and access to metadata provide the code generators with the critical ability to disambiguate memory accesses relatively inexpensively, avoiding aliasing conflicts that would otherwise defeat many code optimizations and transformations. Metadata also allow the code generators to generate sufficient GC information so that the StarJIT compiler can enumerate the root set of live pointers when requested to do so by the garbage collector at runtime.

The implementations of the code generators are completely independent because each architecture family requires a different set of optimizations and code-generation passes and utilizes very different IRs. For example, the Itanium Processor Family code generator performs aggressive trace scheduling; the IA-32 code generator does not need to do this because of its instruction set architecture and its microarchitectural implementation.

Dynamic Profile-Guided Optimizations

The StarJIT compiler supports dynamic profile-guided optimization (DPGO) as part of its dynamic compilation framework. Modern static compilers have used profile-guided optimization (PGO) to achieve significant performance improvement [5] [7]. The performance benefit from PGO on the Itanium Processor Family architecture is even more profound, with a speedup of approximately 20% observed on certain integer benchmarks. Traditional static PGO requires an initial compilation and execution run to collect an execution profile for use in a final compilation step. The three-step process – compiling with instrumentation, executing with representative inputs, and re-compiling with PGO – requires manual involvement, and it adds a significant burden to the usually time-constrained software development cycle. Moreover, this process requires the software vendor to develop a training workload that represents the end-user's workload.

In contrast, DPGO is automatic and transparent to the end user and software vendor. At the center of the StarJIT compiler's DPGO framework is a module called the Profile Manager, which resides in the virtual machine. The Profile Manager manages the collection and processing of the execution profile, and it selects hot methods for recompilation. The first time it compiles a method, the compiler uses lightweight, fast-path optimizations and prepares additional information to support profiling (which depends on the profiling mechanisms used). When a method is executed, its

® Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

® Intel XScale is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

execution profile is collected online. Periodically, the Profile Manager examines the execution profile of each method to determine which methods are hot enough to warrant recompilation. Once the Profile Manager selects a method for recompilation, it tells the compiler to recompile the method with a higher level of optimizations. The Profile Manager also preprocesses the execution profile of the method and provides it to the compiler so that the compiler can apply PGO during recompilation.

This recompilation yields higher performing code for hot methods. The Profile Manager continues this profiling-recompilation process throughout the execution of the application. Since DPGO collects execution profiles and triggers recompilation on-the-fly, it can re-optimize hot methods when there is significant change in their execution profile, allowing the MRTE to adapt to different execution or usage patterns of an application.

Because of the high overhead in collecting an execution profile, DPGO in today's MRTEs is typically constrained to collect a method invocation profile for identifying hot methods and a dynamic call graph for making inlining decisions. As an advanced research platform, the StarJIT compiler provides a much more extensive set of profiles in its DPGO framework. Two types of profiling mechanisms are supported in the StarJIT DPGO framework: one is instrumentation-based and the other is sampling-based.

Instrumentation-based profiling inserts profile-collecting code in the dynamically generated native binary when the compiler first compiles a method. The inserted code increments counters when execution goes through the control flow of the method [2]. The StarJIT compiler currently supports the collection of method invocation and control flow edge execution counts. The inserted code maintains these counters in buffers that are accessible to the Profile Manger. The instrumentation code incurs significant overhead during execution; therefore, the compiler does not generate instrumentation when it recompiles a method with DPGO.

Sampling-based profiling collects an execution profile by collecting samples during the program execution. Instead of simply taking an instruction pointer (IP) sample, the StarJIT compiler utilizes the Performance Monitoring Unit (PMU) of a microprocessor to get a better execution profile of an application. On the Itanium Processor Family architecture, the PMU can monitor and provide a rich set of events and execution information. For example, the Itanium Processor Family PMU has a Branch Trace Buffer (BTrB) that can capture information on the last few branches executed. The branch trace information includes the IP address of a branch instruction and the target IP address of the branch. The information in the BTrB thus captures a short trace fragment during the execution of the program. By taking enough BTrB samples, the Profile Manager is able to construct an execution profile that approximates the edge profile.

The PMU samples contain virtual IP addresses. To map the sampling profile into a control flow profile, the compiler must map IP addresses into IR at the feedback point. To facilitate such IP-to-IR mapping, the StarJIT compiler emits a basic block mapping table when it dynamically compiles a method. The table allows the mapping of an IP address to a basic block and the branch-target pair of IPs to a control flow edge in the optimizer IR.

The Profile Manager can adjust the overhead of sampling-based profiling by changing the sampling rate. Hence once the compiler has recompiled the majority of hot methods with DPGO, the Profile Manager can tune down the sampling rate to lower the profiling cost. The dynamic adjustment of the sampling rate allows non-stop, low-overhead monitoring of the application, making continuous profiling and recompilation feasible in MRTEs.

When profile information is available, the StarJIT compiler feeds the profile information into the IR, and it selects an optimization path consisting of aggressive profile-guided optimizations. The optimizer propagates the profile information to the Itanium Processor Family code generator so that the code generator can use the profile to guide basic block layout, trace selection, instruction scheduling, and other transformations. We discuss the details of profile usage in later sections.

GLOBAL OPTIMIZER

The StarJIT compiler uses a single optimization framework for Java* and Common Language Infrastructure (CLI) programs. The StarJIT global optimizer applies a set of classical, object-oriented, and profile-guided optimizations to the method representation, balancing the aggressiveness of optimizations with their compile-time cost.

Figure 2 shows the high-level flow of the StarJIT global optimizer. The optimizer has two primary phases. The first phase consists of fast optimizations performed every time the StarJIT compiler is invoked. This phase improves code quality and performance without substantial compile-time cost. It carries out a baseline set of optimizations on all generated code. It is deterministic: it uses no contextual information (such as profiling) that may change in a later recompile. If no profile information is available (i.e., this is the first time the StarJIT compiler

* Other brands and names are the property of their respective owners.

is compiling a method) and the Profile Manager is using instrumentation-based profiling, then the StarJIT optimizer instruments the intermediate representation (IR) before invoking the code generator.

If profile information is available (i.e., the method is a hot method that the Profile Manager has selected for recompilation), the optimizer annotates it into the STIR after the first phase and runs the second optimization phase. This phase applies more aggressive optimizations and takes advantage of profile information in the annotated STIR. Through this second phase, the StarJIT compiler focuses compilation time on methods and regions that are most critical to overall performance.

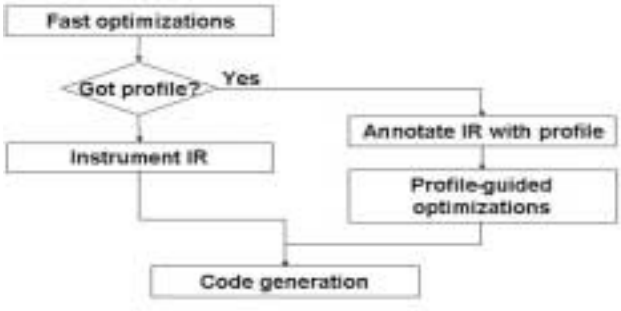


Figure 2: The StarJIT global optimizer

Each of the two optimization phases performs the same basic set of optimization passes. These passes are grouped into four categories. *Scope Enhancement* passes, *Privatization* passes, and *Redundancy Elimination* passes are performed in sequence, while *IR Simplification* passes are performed at multiple points to clean up the method representation between passes. In the first phase, all passes use conservative settings to run quickly. In the second phase, the passes use profile information and more aggressive settings. In this manner, the StarJIT compiler balances compile time with performance by concentrating expensive optimizations only on methods that are hot. The remainder of this section describes the optimization passes in more detail.

Intermediate Representation Simplification Passes

IR simplification passes are a set of very fast optimization passes that the StarJIT optimizer performs several times on the IR. These optimizations reduce the size and complexity of the IR. In addition to improving the code quality, this reduction improves the efficiency of other, more expensive optimizations. IR simplification consists of three passes.

The first pass involves propagation and folding. This pass performs constant, type, and copy propagation over the entire method following the static single assignment (SSA)-form use-def links. As it does this, it also simplifies and folds expressions such as arithmetic on constants or runtime checks for null references that are proven non-null (e.g., a reference defined by a new allocation). When branch conditions or instructions that can potentially raise an exception are folded, the corresponding edges are also removed from the control flow graph, and any unreachable code as a result of the edge deletion is skipped (effectively performing conditional constant propagation [19]).

The second pass eliminates unreachable and useless code. It does the former by testing reachability via traversal from the control flow graph entry; it does the latter by using a sparse liveness traversal over SSA-form use-def links.

The third pass performs fast global value numbering to eliminate common subexpressions [3]. This pass does an in-order depth-first traversal of the dominator tree (instead of the more expensive iterative dataflow analysis done by traditional common subexpression elimination). At any given program point, SSA-form expressions computed earlier within the same basic block are considered available. In addition, expressions that are available at the end of dominating blocks are also available. Expressions that may be killed (such as loads from memory) or have side-effects (such as calls) are ignored.

Global value numbering is effective in eliminating redundant address computation and check instructions (e.g., chkzero, chknull, and chkcast that are redundant or guarded by explicit conditional branches). Later optimization passes eliminate redundant memory accesses (which require alias analysis and kill information) and array bounds checks (which are difficult to remove in a single forward pass because they require arithmetic reasoning and propagation of dataflow facts across loop back edges).

Together, the IR simplification passes can be thought of as a single cleanup pass. This cleanup is performed at a number of points in the optimization process.

Scope Enhancement Passes

The global optimizer begins with a set of transformations designed to enhance the scope of later optimizations. The first scope enhancement pass normalizes control flow by removing critical edges (a critical edge is an edge from a node with multiple successors to a node with multiple predecessors), and factoring entry and back edges of loops. These transformations prepare the intermediate representation for later optimization; for example, loop

normalization simplifies the implementation of peeling, and critical edge removal is necessary for redundancy elimination.

After normalization, the optimizer performs a set of loop transformations. These include loop inversion, peeling, and unrolling. The first optimization phase is conservative, and performs only loop inversion and limited partial peeling. The second profile-driven optimization phase is more aggressive, and performs profile-driven peeling and unrolling of hot loops. Note that loop peeling, in combination with global value numbering, provides a cheap mechanism to hoist loop-invariant computation and runtime checks.

The third scope enhancement optimization is guarded devirtualization of virtual method calls. Virtual method calls are prevalent in managed runtime environment (MRTE) applications. They differ from direct calls in that the actual call target must be resolved at runtime by examining an object's virtual method table. The costs of this extra level of indirection include the runtime expense of extra code to invoke a virtual method and potentially poorer branch prediction in hardware for that call, as well as the compile-time expense of impeded interprocedural analysis and inlining.

In cases where the optimizer has exact type information, the IR simplification pass is able to devirtualize a virtual call by converting it into a more efficient direct call. In other cases, the target of a virtual method may be highly predictable. In these cases, the scope enhancement pass devirtualizes the call by guarding it with an inexpensive runtime test that checks whether the predicted method is in fact the target. If performed accurately, guarded devirtualization alleviates the runtime costs associated with virtual method calls and enables the compiler to inline targets of virtual method calls. The first phase performs guarded devirtualization conservatively using simple static heuristics. The profile-driven phase performs guarded devirtualization aggressively using block execution and call graph profiles.

The centerpiece of the scope enhancement passes is the inliner. Inlining removes the overhead of a direct call and specializes the called method within the context of its call site. The inliner consists of an iterative process built around the other scope enhancement and IR simplification passes. In the first pass through this cycle, scope enhancement and IR simplification transformations are performed on the original intermediate representation. At this point, the inliner examines each direct call site in the IR (including those exposed by guarded devirtualization), heuristically assigns a benefit to it, and, if it exceeds a certain threshold, registers it in a priority queue. The top candidate, if any, is then selected for inlining. The translator generates IR for the inlined method, and the cycle is repeated upon the new IR. The inliner then processes the new IR for further inlining candidates (updating the priority queue), splices it into the existing IR, selects a new candidate, and repeats the cycle. The inliner halts once the queue is empty or after the IR reaches a certain size limit. When inlining is completed, the global optimizer performs a final IR simplification pass over the entire intermediate representation.

Privatization Passes

The privatization passes optimize accesses to memory locations. The privatization phase first performs alias and escape analyses on memory accesses, and then performs synchronization removal [18] and scalar replacement [16]. Alias analysis yields information about which load/store addresses may affect each other [16]. The StarJIT optimizer uses the type information about object fields for alias analysis. For example, accesses to two object fields cannot refer to the same location if the object types, field names, or field types differ. A store cannot alias with a final or read-only field of an object (except in the object constructor). The StarJIT optimizer also uses the definition point of an object reference for alias analysis: a reference to an object that is a method parameter may not alias with a reference resulting from an object allocation.

Escape analysis determines the extent to which accessed memory locations are visible outside the current method [6][13]. Escape analysis determines this information with a sparse SSA-based analysis of each object referenced in a method. An object that is allocated in the body of the method is initially assumed to be private to the method (i.e., non-escaping). Any object passed in as an argument or a return value, passed as an argument to another method, returned as a result, or stored into a static field escapes by definition. Moreover, any object stored as a field in an escaping object transitively escapes. Finally, any object that potentially aliases an escaping object (because of a copy or a merge at an SSA phi node) also escapes. The StarJIT optimizer's current escape analysis algorithm is intra-procedural and relies on the prior inlining pass to expose privatization opportunities. We plan on augmenting the escape analysis pass with inter-procedural information.

Once escape analysis is done, synchronization removal eliminates synchronization operations (which are explicit in STIR) on objects that do not escape a method or that escape only via a return. Scalar replacement promotes object fields and array elements to SSA variables that are amenable to further optimization passes [9]. This pass takes advantage of the alias and escape analysis information to disambiguate memory references.

Redundancy Elimination Passes

The final set of optimization passes comprises optimizations to eliminate redundant and partially redundant computations. These passes include loop-invariant code motion, bounds-check elimination, and strength reduction [16]. They are deferred until the largest possible program scope is available and the most memory locations have been promoted to scalar variables.

The StarJIT optimizer uses a demand-driven array bounds-check elimination analysis based upon the previously published ABCD algorithm [4]. It first inserts Pi nodes into the IR to split variable live ranges based on branch conditions. Pi nodes capture information gleaned about a variable based on branch conditions. From each variable's definition, the analysis then derives inequality constraints upon that variable's value, which can be used to prove redundancy of bounds checks involving that variable. Unlike the original ABCD algorithm, the StarJIT optimizer's bounds-check elimination implementation does not construct a separate constraint graph, but uses the SSA graph directly to derive constraints during an attempted proof. We also have added handling of symbolic constants to allow check elimination in slightly harder cases, commonly encountered in practice.

To facilitate load hoisting in the code generator, the check-elimination transformations track conditions used to prove that a check can be eliminated. The code generation interface passes this information to the code generator. The scheduler uses this information to determine which branches guard the safety of a given load, and marks the load as speculative if it hoists the load above a guarding branch.

The StarJIT optimizer performs strength reduction to transform expensive operations, such as multiplication by an induction variable in a loop, into simpler operations such as addition. The implementation is based upon the operator strength reduction optimization described in [8], extended to also reduce the strength of memory address computations. The strength reduction pass performs linear function test replacement to eliminate uses of the original loop induction variable in tests (e.g., loop exit tests). This optimization is effective in transforming an iteration through the elements of an array into a series of pointer increments and pointer comparisons, and eliminating the original array index. In cases where the loop index is live after the loop, this pass rematerializes the index on loop exits to still allow removal of the loop index computation from the loop. For the Itanium® Processor Family architecture, the strength reduction pass can also transform loops with invariant trip counts into counted loops.

The optimizer must be careful during strength reduction because of overflow issues: the optimizer cannot transform a 32-bit integer induction variable used as an array index into a 64-bit pointer (strength reducing the indexing operations) unless it can prove that additions to the 32-bit index will not overflow (and wrap around to a negative number, as required by Java* bytecode semantics) because adding to the 64-bit pointer will not overflow in the same cases. While unlikely to occur in real code, the induction variable range is checked for possible overflow before such a strength reduction transformation. The range analysis makes use of the same demand-driven bounds-check analysis used for array bounds-check elimination.

THE ITANIUM PROCESSOR FAMILY CODE GENERATOR

The Itanium® Processor Family code generator is responsible for generating native code for a program represented by STIR. It lowers the program representation to the machine level, performs architecture-dependent optimizations such as register allocation and scheduling, computes the information necessary to support garbage collection (GC), and emits the bits that are directly executed by the processor.

Figure 3 shows the structure of the code generator. The first code generation phase is code selection. During this phase the code generator lowers STIR operations into Itanium Processor Family code sequences and performs simple optimizations such as immediate operand folding, operator folding, and strength reduction. It uses predication [15] to avoid generating additional control flow for complex STIR operations such as `instanceOf`. The Itanium Processor Family instruction sequences generated from STIR usually contain many operations that move data between temporaries, variables, incoming and outgoing arguments, and return values. The code selector makes a pass over the intermediate representation to coalesce the sources and destination operands of moves, and to remove the resulting redundant moves.

® Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

* Other brands and names are the property of their respective owners.

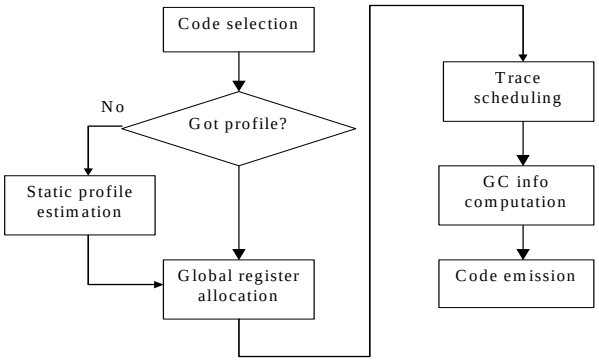


Figure 3: Itanium Processor Family code generator phases

The optimizer drives code selection through a code-generation interface. This interface abstracts the information that the optimizer should communicate to a code generator from the details of STIR implementation and allows the code generator to be used with any front-end that supports the code generation interface. The subsequent code generation phases require profile information to guide optimizations. When dynamic profile information is not available, the code generator estimates the profile using static heuristics [1].

The ordering of register allocation and code scheduling is a classical phase-ordering problem [12]. Register allocation performed before scheduling introduces additional anti and output dependencies that restrict scheduler freedom to reorder the instructions. Register allocation performed after code scheduling may require an additional scheduling pass to accommodate generated spill code. In addition, register allocation quality may suffer because of increased register pressure. The code generator chooses a middle-ground approach. It divides all operands into two categories: local and global. An operand is local if it has a single definition and its live range does not span a loop boundary. All other operands are global. Only global operands require iterative data flow analysis to compute their liveness. The liveness of local operands can be computed with a single reverse pass over the IR. The global operands are assigned registers during the global register allocation phase that occurs before scheduling. This introduces only a few data dependencies, as most of the operands are local. The local register allocator is integrated with scheduling. The scheduler keeps track of the register pressure, and materializes and schedules spill code as needed.

The code scheduler is the most complex component of the code generator. In addition to scheduling instructions using trace scheduling [11], it performs code layout and local register allocation. The design of the trace scheduler is described in the next section.

After scheduling, the code generator computes the information necessary to support GC. For each call instruction, it computes the set of registers and stack locations that contain live references and interior pointers (pointers to the middle of the objects allocated on the heap), and it records this information in a data structure called the GC map table. During garbage collection, the garbage collector enumerates the root set by iterating over the set of frames on each thread's runtime stack. For each frame, the garbage collector makes a callback into the JIT compiler asking it to enumerate the set of live references for that frame and to unwind to the previous frame. The JIT compiler computes the set of live references for the frame using the GC map information.

The final code emission phase emits the native Itanium Processor Family binary code into memory for execution. This phase also emits the GC map table, exception handler tables (for dispatching exceptions), stack unwinding information (for root set enumeration, exception unwinding, and runtime security checks), and the IP-to-IR mapping tables (for profile gathering).

Trace Scheduler Design

The modular trace scheduler design facilitates managed runtime environment (MRTE) research work, retargetability to other micro-architectures, and portability for use in other virtual machine (VM) or compilation systems. The various components of the trace scheduler are shown in Figure 4. The components have been designed as independent modules with clear interfaces so that they can be applied to each trace selectively.

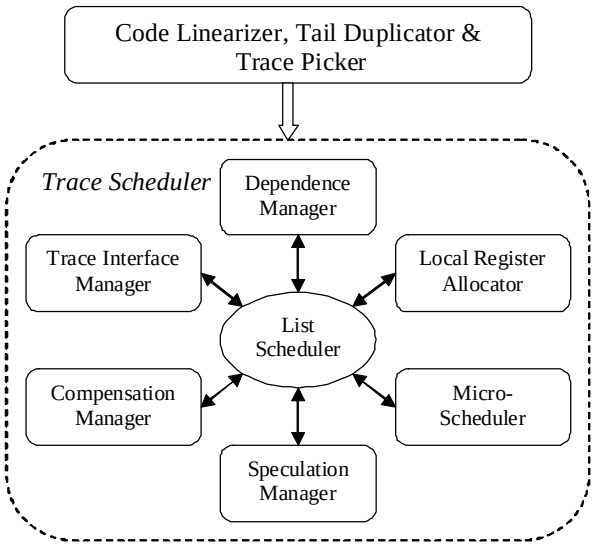


Figure 4: Trace scheduler components

The first pass of the trace scheduler is the code linearizer, tail duplicator, and trace picker. Trace selection is important because it defines the scheduling scope. Trace selection and scheduling are best done after code (basic block) layout, to schedule unconditional branches introduced by code linearization, and to form cross-block bundles and cycles. Tail duplication is useful to eliminate side entries into traces, but must be done before code layout decisions are finalized. Tail duplication decisions, however, are best made with input from trace formation. Therefore, there is a cyclic phase ordering dependency between trace picking, code layout, and tail duplication.

The StarJIT Itanium Processor Family code generator uses a novel scheme that performs all three together. The code layout technique is a top-down scheme similar to that described by Pettis & Hansen [17]. Code layout, trace formation, and tail duplication decisions all benefit from any available branch profile information. Code layout uses profiles to improve cache locality and reduce taken branches. Along hot paths the trace picker picks longer traces, and the tail duplicator is more aggressive in removing cold side entries, while on cooler paths shorter traces are picked with little or no tail duplication. Finally, a few compensation blocks are added on some critical edges. After scheduling, the code generator eliminates useless compensation blocks, which have no compensation code moved into them.

At the core of the trace scheduler is the list scheduler, which schedules one trace at a time. The list scheduler schedules instructions from a data-ready list. It uses several heuristics to choose between data-ready candidates. These include critical path length, slack (a measure of the freedom to delay an operation without delaying the overall schedule), register and resource availability and future needs, code size and code motion usefulness metrics, and effects of any required compensation, or speculation. The heuristics are profile sensitive: their basic goals are to generate high-performance code at hot traces and to enable fast generation of compact code at cold traces. The list scheduler heuristics also guide multiway branch generation. MRTE safety checks, such as null pointer, array bounds, and type checks, result in a large number of branch operations. It is therefore important to bundle multiple branches together to reduce code size, control height, and mispredicted branches.

The list scheduler uses a micro-scheduler to schedule instructions within a cycle. The micro-scheduler models resources and dispersal rules, and makes compact bundling decisions. For the Itanium Processor Family, it is important to integrate scheduling with bundling because the bundling choices influence dispersal. The micro-scheduler is based on the Open Research Compiler's micro-scheduler [14]. It abstracts away the machine details and reads the Itanium micro-architecture definition from a knobs file.

The dependence manager tracks all register data dependencies, memory dependencies, and control dependencies while trying to avoid transitive dependencies, for efficiency reasons. It uses MRTE metadata to avoid creating false memory and control dependencies. Memory disambiguation is based on the properties of pointers to memory locations such as type, memory region (heap, stack, static), and access semantics (e.g., field, array element). The dependence manager uses the safety semantics of MRTE memory operations to avoid unnecessary control dependencies. A load is safe (i.e., can be issued without making it speculative) everywhere except before its corresponding safety checks (chknull, chkbounds, and/or chkcast). When the optimizer combines or eliminates any of these checks based on control or data flow implications, it keeps around enough information to allow the dependence manager to recognize control dependencies of such loads on the appropriate check and/or branch instructions. The dependence manager also enables the list scheduler to use predication to convert a control dependency on a branch to a data dependency on the associated predicate-generating compare. This allows the list scheduler to predicate a block partially, thus reducing the need for speculation, check, and recovery generation.

The speculation manager uses the Itanium Processor Family control speculation feature to schedule loads before the branches on which they are control dependent. It keeps track of the speculative loads and dependent speculative instructions that should be included in the recovery code. After all traces have been scheduled, the speculation manager materializes the recovery code and schedules it using a local scheduler.

When instructions are moved above a trace side entry or below a trace side exit, the compensation manager inserts copies of these instructions in the off-trace blocks. The scheduler performs code motion, only when heuristics suggest that the good done to the on-trace path is not outweighed by any harm done by compensation code to the off-trace path. Code motion requiring compensation insertion into previously scheduled traces is not permitted. Profile information (which determines the order in which traces are scheduled), therefore, guides compensation code decisions. The compensation manager also avoids compensation code when control and data dependence relationships indicate that it is unnecessary. For example, compensation code is not needed at intermediate side entry points when an instruction is moved to a dominating point in the trace and the instruction's operands are not modified on any off-trace path.

The trace interface manager models liveness and data flow latency across trace boundaries (trace main entry/exit and side entries/exits), thus maximizing scheduling freedom and improving performance at trace interfaces.

The StarJIT trace scheduler has an integrated local register allocation module (as mentioned earlier, global operands are allocated registers prior to scheduling). This module monitors liveness of local temporaries and allocates registers to them when their definitions are scheduled. A local temporary has a single definition that dominates all its uses. The scheduler exploits this property to model register pressure during scheduling, and to materialize and schedule spill code on-the-fly, thus performing efficient and optimized register allocation.

CONCLUSION

Managed Runtime Environments (MRTEs) depend on dynamic compilation for performance and security. The strict runtime requirements of dynamic compilation pose new challenges to compiler engineers. These requirements also provide new dynamic optimization opportunities involving both the compiler and the hardware.

In this paper, we have described the design of the StarJIT compiler. Built upon a framework that enables dynamic recompilation for a range of MRTEs and Intel architectures, this research infrastructure enables heretofore intractable research opportunities in implementation tradeoffs of managed runtimes and hardware architectures.

ACKNOWLEDGMENTS

The authors thank members of the Open Research Platform (ORP) VM team, Michal Cierniak, Neal Glew, Rick Hudson, Brian Lewis, James Stichnoth, Sreenivas Subramoney, and Weldon Washburn. The StarJIT compiler would not have been realized without their support and efforts in making the ORP VM robust and high-performing. We thank Youfeng Wu, Roy Ju, and Sun Chan for providing valuable feedback on the IPF trace scheduler design and details on the ORC micro-scheduler. The authors also thank Jesse Fang for his guidance and continuing support of this work, Ken Lueh for his early contributions to the StarJIT source code, and Youngsoo Choi for his contributions to the Itanium Processor Family PMU driver.

REFERENCES

[1] T. Ball and J.R. Larus, "Branch Prediction for Free," in proceedings of *ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993, pp. 300-313.

[2] T. Ball and J.R. Larus, "Optimally Profiling and Tracing Programs," Conference Record of the *Nineteenth ACM Symposium on Principles of Programming Languages*, January 1992, pp. 59-70.

[3] P. Briggs, K.D., Cooper and L.T. Simpson, "Value Numbering. Software-Practice and Experience," vol. 27(6), June 1997, pp. 701-724.

[4] R. Bodik, R. Gupta, and V. Sarkar, "ABCD: Eliminating Array-Bounds Checks on Demand," in proceedings of the *SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000, pp. 321-333.

[5] P.P. Chang, S.A. Mahlke and W.W. Hwu, "Using Profile Information to Assist Classic Code Optimizations," *Software-Practice and Experience, vol. 21(12)*, Dec. 1991, pp.1301-1321.

[6] J.-D. Choi, M. Gupta, M.J. Serrano, V.C. Sreedhar and S.P. Midkiff, "Escape Analysis for Java," in proceedings of the *Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA), Denver, Colorado, 1999, pp. 1-19.

[7] R. Cohn, D. Goodwin and P.G. Lowney, "Optimizing Alpha Executables on Windows NT with Spike," *Digital Technical Journal, vol. 9, No. 4*, 1997, pp. 3-20.

[8] K.D. Cooper, L.T. Simpson and C.A. Vick, "Operator Strength Reduction," *ACM Transactions on Programming Languages and Systems, vol. 23, no. 5*, September 2001, pp. 603-625.

[9] K.D. Cooper and L. Xu, "An Efficient Static Analysis Algorithm to Detect Redundant Memory Operations," *ACM 2002, Workshop on Memory System Performance* (MSP '02), Berlin, Germany, June 16, 2002.

[10] R. Cytron, J. Ferrante, B. Rosen, M. Wegman and F. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems, vol. 13, No. 14*, October 1991, pp 451-490.

[11] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers, C-30(7),* July 1981, pp. 478-490.

[12] S.M. Freudenberger and J.C. Ruttenberg, "Phase Ordering of Register Allocation and Instruction Scheduling," in proceedings of *the International Workshop on Code Generation*, May 1991, pp. 146-172.

[13] D. Gay and B. Steensgaard, "Fast Escape Analysis and Stack Allocation for Object-Based Programs," *9th*

International Conference on Compiler Construction, (CC '2000), *Springer-Verlag, Vol. 1781*, 2000, pp. 82-93.

[14] R. Ju, S. Chan, F. Chow, X. Feng and W. Chen, "Open Research Compiler (ORC) Beyond Version 1.0," tutorial presented at *PACT-2002*, September 22, 2002.

[15] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank and R.A. Bringmann, "Effective Compiler Support for Predicated Execution Using the Hyperblock," in proceedings of the *25th International Symposium on Microarchitecture* (MICRO 25), Dec. 1992, pp. 45-54.

[16] S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, CA, 1997.

[17] K. Pettis and R.C. Hansen, "Profile Guided Code Positioning," in proceedings of the *ACM SIGPLAN 90 Conference on Programming Language Design and Implementation*, White Plains, N.Y., June 20-22, 1990, pp. 16-27.

[18] E. Ruf, "Effective synchronization removal for Java," in proceedings of the *ACM SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, British Columbia, June 2000, pp. 208-218.

[19] M. Wegmen and F. Zadeck, "Constant Propagation with Conditional Branches," *ACM Transactions on Programming Languages and Systems, vol. 13, No.2*, April 1991, pp. 181-210.

AUTHORS' BIOGRAPHIES

Ali-Reza Adl-Tabatabai is a senior staff researcher in the Programming Systems Lab. He received a B.Sc. degree from UCLA in Computer Science & Engineering and a Ph.D. degree from Carnegie Mellon University in Computer Science. His research interests include dynamic compilation and optimization, managed runtimes, memory hierarchy design, and compression. His e-mail is ali-reza.adl-tabatabai@intel.com

Jay Bharadwaj is a senior staff researcher in the Programming Systems Lab. He received a B.S. degree in Mechanical Engineering from IIT Madras, India and M.S. degrees in Computer Science and Mechanical Engineering from Rensselaer Polytechnic Institute and SUNY Stony Brook, respectively. His research interests include managed runtimes, hardware software cooperation, and compilation techniques. Other interests include activities requiring use of hand or power tools. His e-mail is jay.bharadwaj@intel.com

Dong-Yuan Chen is a staff researcher with the Programming Systems Lab. He received his Ph.D. degree in Computer Science from Yale University in 1995. He has worked on back-end compiler optimizations, including software pipelining and machine modeling, and various microarchitectural performance studies for the Itanium Processor Family architecture. His current interests include lightweight online profiling mechanisms and dynamic profile-guided optimizations in managed runtime environments. His e-mail is dong-yuan.chen@intel.com

Anwar Ghuloum is a senior staff researcher in the Programming Systems Lab. He received a B.Sc. degree from UCLA in Computer Science & Engineering and a Ph.D. degree from Carnegie Mellon University in Computer Science. His research interests include managed runtime environments, memory hierarchy design, and compression. Other pursuits include cycling, tri, building bikes, painting, and the uses of coherent light. His e-mail is anwar.ghuloum@intel.com

Vijay Menon is a staff researcher in the Programming Systems Lab. He received a B.S. from the University of California, Berkeley in Electrical Engineering and Computer Science and a Ph.D. from Cornell in Computer Science. His current research interests include program analysis, dynamic compilation, and managed runtime environments. His e-mail is vijay.menon@intel.com.

Brian R. Murphy is a Just-In-Time researcher at Intel Labs. He has done analysis of functional languages, automatic parallelization of Fortran code, development of advanced program analysis techniques, programming language design and implementation, Unix and Linux systems programming and administration, and Web site development and management. He received S.B. and S.M. degrees from M.I.T., and a Ph.D. degree from Stanford University. His e-mail is brian.r.murphy@intel.com

Mauricio Serrano received his Ph.D. degree in Computer Engineering from the University of California Santa Barbara in 1994, an M.S. degree from Rensselaer Polytechnic Institute, and a B.S.E.E. from Javeriana University, Bogota, Colombia. Before joining Intel, he spent several years working with IBM T.J. Watson/New York and STL/San Jose, where he worked in several compiler areas including program restructuring, retargetable code generation, and Java performance optimizations. His other interests are computer architecture and performance modeling. He published the first dissertation on SMT (Simultaneous Multithreaded Processors) in 1994, although at that time he called it

SMS (Simultaneous Multistream Superscalar Processors). His e-mail is mauricio.j.serrano@intel.com

Tatiana Shpeisman is a staff researcher in the Programming Systems Lab. She received her B.Sc. degree from the Leningrad Electrical Engineering Institute in Applied Mathematics and M.S. and Ph.D. degrees from the University of Maryland, College Park, in Computer Science. Her research interests include compilation techniques, managed runtimes, and sparse matrix computations. Her other interests include hiking in the Sierras, ballroom dancing, and classical ballet. Her e-mail is tatiana.shpeisman@intel.com.

Enterprise Java Performance: Best Practices

Kingsum Chow, Software and Solutions Group, Intel Corporation
Ricardo Morin, Software and Solutions Group, Intel Corporation
Kumar Shiv, Software and Solutions Group, Intel Corporation

Index words: Enterprise Applications, Application Servers, Java Performance, Java 2 Enterprise Edition, J2EE

ABSTRACT

This paper discusses best practices for maximizing the performance of enterprise Java* workloads. First, we introduce the importance of performance of enterprise Java applications. We then describe our top-down, data-driven, and closed-loop approach to characterize where the problems are. We examine the performance of the software/hardware stack, first from the system-level perspective (topology, I/O, network), then from the top software layer (application level), through the middle layer (Java Virtual Machine), and down to the platform layer (processor, memory). We conclude by summarizing our recommendations for attaining the best performance on enterprise Java applications.

INTRODUCTION

Managed runtime environments such as Java have proven to be a very attractive platform for developing and deploying enterprise applications. Accessible object orientation, programming safety, and automatic memory management features deliver a highly productive foundation for business application development. In addition, the platform independence offered by managed runtime environments provides unprecedented investment protection, which is appealing to Information Technology (IT) managers, as enterprise applications tend to have a long life span.

Advanced Just-In-Time (JIT) compilation, memory management, and garbage collection technologies have effectively addressed initial concerns raised about the poor performance of Java-based applications. Today's Java Virtual Machines (JVM*) take full advantage of a variety of target platforms, and keep up to date with the performance of the latest hardware and operating system advances as they evolve over time.

As Java [1] gained popularity in the development of server-based applications, standardized, robust, and scalable application support frameworks became a must. Enter Java 2 Enterprise Edition (J2EE) [2], a comprehensive specification for application servers, a class of system software designed to relieve application developers from creating and re-creating the "plumbing" necessary to support enterprise applications, including component models and life-cycles, object models, database access, security, transactional integrity, and safe multi-threading.

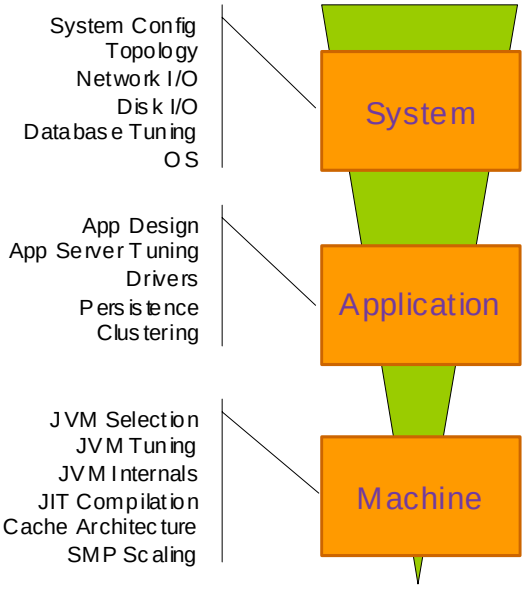


Figure 1: Performance optimization considerations at the three levels of the top-down stack: system-level, application-level, and machine-level

Since the emergence of J2EE, application servers have grown to become important IT infrastructure components of many enterprises [3]. They support complex, multi-tier configurations with well-defined separation of functions

(user interface, business processing, and database access), often including multiple servers arranged in clustered configurations, as well as back-end relational database management systems and legacy applications, integrated in the overall design.

As applications move from development to production, performance becomes a critical life-cycle requirement. Applications must not only meet stringent performance requirements upon deployment, but they must be able to gracefully scale with varying usage patterns and increased demand. Performance optimization and management in this environment is a difficult task, as performance is affected by many interrelated elements.

In this paper, we describe an iterative, data-driven, top-down methodology and the tools needed to systematically optimize the performance of application-server-based applications. We also describe performance optimization considerations at the three levels of the top-down stack: system-level, application-level and machine-level (see Figure 1).

At the system level, we identify performance and scalability barriers such as input/output (I/O), operating system and database bottlenecks, and we discuss techniques to overcome those barriers. At the application level, we discuss application design considerations and application server tuning. At the machine level, we discuss JVM implementations and hardware-level performance considerations such as processor frequency, cache sizes, and multi-processor scaling.

Throughout the paper, we introduce several case studies to illustrate the application of the techniques presented.

APPLICATION SERVERS

Application servers provide a solid foundation for developing and deploying enterprise applications. They implement a large collection of Application Program Interfaces (API) and a set of capabilities specified in the J2EE suite of standards, which support the development of multi-tier applications.

Application-server-based applications are arranged in multi-tier configurations: client tier, Web interface tier, business tier, and enterprise information systems tier. The client tier represents the service requestors, and it is usually associated with the user interface. The Web interface tier provides services required to process Web-based forms and Web services, and it dynamically assembles the resulting HTML and/or XML. The business tier is used to implement computation, business processing, and business rules. The enterprise information systems tier includes persistence back-ends, based on relational databases and legacy applications such as mainframe-based information systems.

As depicted in Figure 2, application server functionality is organized around the concept of containers, which provide groupings of related functions, and are typically layered on top of the Java 2 Standard Edition (J2SE*) platform, which includes a Java Virtual Machine (JVM) and the corresponding suite of APIs. For instance, the two application server-based containers are the Web container and the Enterprise JavaBeans (EJB) container. Web containers are used to support Web-based user interface components, such as Servlets and Java Server Pages (JSP). EJB containers are used to support business components, which include Session Beans, Entity Beans, and Message-driven Beans. Session Beans provide access to independent business components in two flavors: Stateful Session Beans, used when state information is required between service calls, and Stateless Session Beans, used when individual service calls are independent of each other and do not require state information to be preserved. Entity Beans provide persistence services through connectivity to relational databases. Message-driven Beans provide the ability to implement business components that take advantage of asynchronous messaging capabilities.

In addition to the core container APIs, application servers provide additional support to APIs such as naming and directory services (JNDI*), database connectivity (JDBC*), messaging (JMS*), XML processing (JAXP*), transactions (JTS*), and connectivity to legacy systems (JCA*).

Most application servers also provide the ability to transparently cluster multiple containers in order to enable fault tolerance and multi-node scalability.

To provide runtime support for this comprehensive set of functionality, application servers need to implement a number of key services, including state management, life-cycle management, thread pooling, transactions, security, persistence, fault tolerance, and load balancing.

Examples of commercial application servers include BEA's WebLogic* [4], IBM's WebSphere* [5] and Oracle's 9i AS* [6]. In addition, there are a number of open source implementations, including JBoss* [7] and JOnAS* [8]. Additional information about J2EE* and application servers can be reviewed in [9], [10], and [11].

* Other brands and names are the property of their respective owners.

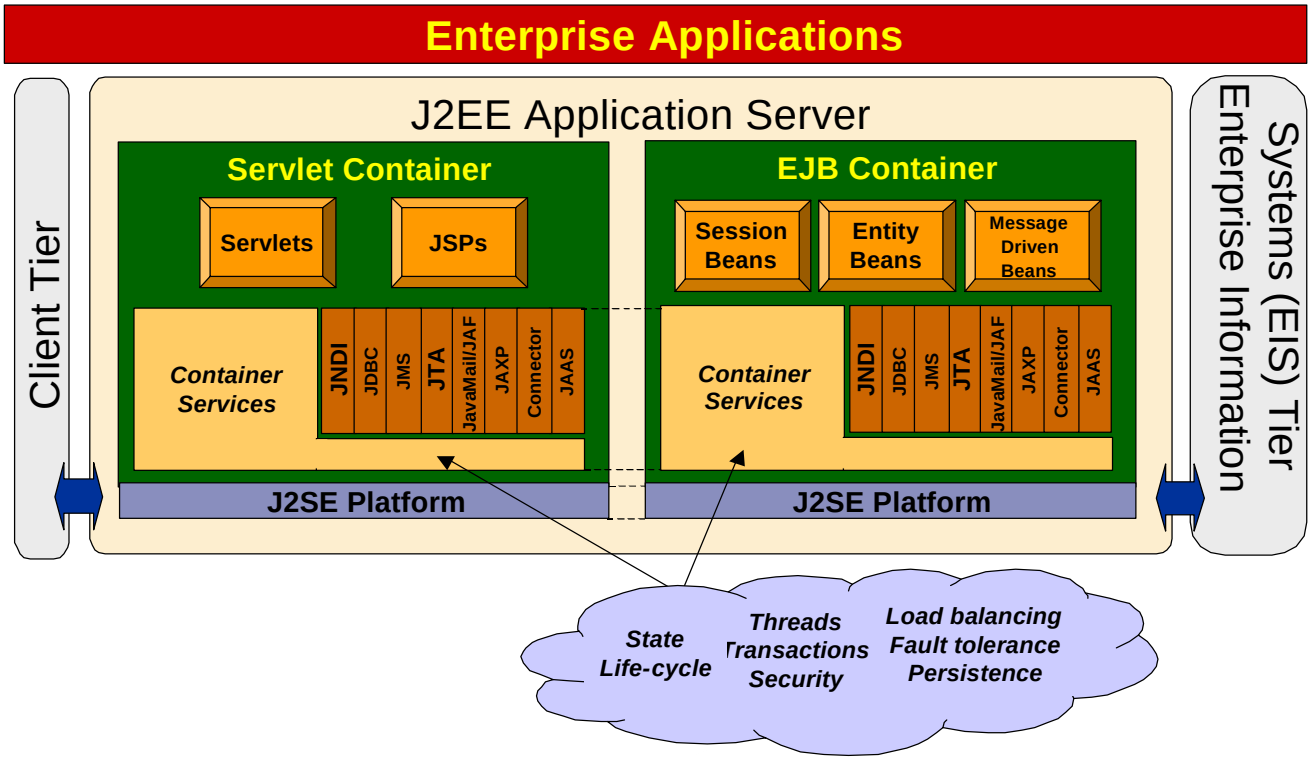


Figure 2: The two server-based containers are the Web and Enterprise JavaBeans (EJB) containers. Web containers support Web-based user interface components, such as Servlets and Java Server Pages (JSP). EJB containers support business components, which include Session Beans, Entity Beans and Message-driven Beans.

PERFORMANCE TUNING METHODOLOGY

Application server configurations involve multiple computers interconnected over a network. Given the complexity involved, ensuring an adequate level of performance in this environment requires a systematic approach. There are many factors that may impact the overall performance and scalability of the system. Examples of these performance and scalability factors include application design decisions, efficiency of user-written application code, system topology, database configuration and tuning, disk and network input/output (I/O) activity, Operating System (OS) configuration, and application server resource throttling knobs.

The first and foremost element an application implementer needs to keep in mind to achieve the desired level of performance is ensuring that the application architecture follows solid design principles. A poorly designed application, in addition to being the source of many performance-related issues, will be difficult to maintain. This compounds the problem, as resolving performance issues will often require that some code be re-structured and sometimes even partially re-written.

Once an enterprise application is ready for deployment, it is critical to establish a performance test environment that mimics production. This environment is then used to identify and remove performance and scalability barriers, using an iterative, data-driven, and top-down methodology.

A key consideration in the performance analysis process is selecting the workload. A good workload exhibits three fundamental attributes.

- First, the workload must be representative. It must provide adequate functional coverage, realistic implementation and usage patterns, and it must be relevant to the goal. The best workload is a controlled baseline of the application under study, configured as close as possible to production. It is highly desirable to fully populate the back-end database with realistic data in order to uncover data access bottlenecks, a common source of performance problems.

- Second, the workload must be measurable. It must have well-defined metrics and must exhibit a stable measurement period over which to gather performance statistics. Performance metrics for enterprise applications are usually defined in terms of throughput (number of operations per unit of time) and response time (amount of time that it takes to process individual transactions). Another commonly used metric is the number of simultaneous requests applied to the workload. This is often referred to as the injection rate. In many cases the total number of concurrent users achieves the same purpose.

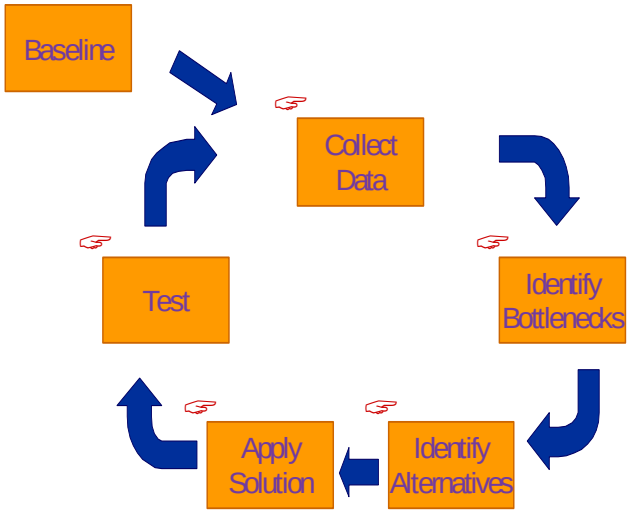


Figure 3: The iterative, data-driven, top-down process for performance tuning and optimizations

- Third, the workload must be repeatable. It needs to be consistent across tests, in order to be able to perform reliable data analyses and draw meaningful conclusions. In addition, the workload state needs to be the same at the start of each run. For example, if the workload adds data to the database, the database needs to be reinstated to the original state to avoid progressive performance degradation over successive runs. Variations in the primary metrics should not exceed a 5% margin across measurements.

Prior to engaging in performance tuning, it is important to establish a baseline to provide the basis for measuring performance improvements as the tuning process progresses. The baseline should be configured based on the estimated capacity needed to sustain the desired load, including network bandwidth and topology, processor memory sizes, disk capacity and physical database layout. In addition, the baseline configuration should incorporate basic initial configuration recommendations given by the application server, database server, JVM, and hardware platform vendors. These include: recommended tunable parameter settings, choice of database connectivity (JDBC) drivers, and the appropriate level of product versions, service packs, and patches.

Part of the baselining process also includes defining performance goals for the system. Performance goals are usually defined in terms of desired throughput within certain response time constraints: for example, the system needs to be able to process 500 operations per second with 90% or more of the operations taking less than one second.

The steps in the iterative process, as illustrated in Figure 3, are as follows:

- Collect data: Use stress tests and performance-monitoring tools to capture performance data as the system is exercised.
- Identify bottlenecks: Analyze the collected data to identify performance bottlenecks.
- Identify alternatives: Identify, explore, and select alternatives to address the bottlenecks.
- Apply solution: Apply the proposed solution.
- Test: Evaluate the performance effect of the corresponding action.

Once a given bottleneck is addressed, additional bottlenecks may appear, so the process starts over again: performance data is collected and the cycle is initiated again, until the desired level of performance is attained. Two very important points to keep in mind during this process are first, let the available data drive performance improvement actions, and second, make sure only one performance improvement action is applied at a time.

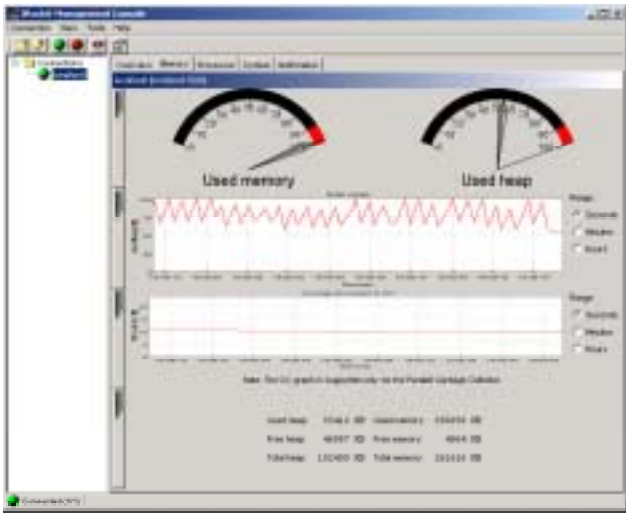


Figure 4: BEA WebLogic JRockit management console

As the quantity and variety of collected data can be overwhelming, and the bottlenecks can often come from

many interrelated sources, it is important to follow a top-down approach. At the top are system-level items such as disk subsystem configuration, network devices, and database configuration; in the middle are application-level items such as transaction configuration, persistence strategies, and JDBC drivers; and at the bottom are machine-level items such as JVM configuration, multi-processor configurations, and processor caches. The iterative process described above needs to be applied at each level of this hierarchy.

Having the right set of tools available is essential for supporting productive performance tuning activities. Performance tools fall under the following categories:

- Stress test tools. These provide the ability to script application scenarios and play them back, thereby simulating a large number of users stressing the application. Commercial examples of these types of tools are Mercury Interactive's LoadRunner* [12] and RADView's WebLoad* [13]; open-source examples include the Grinder* [14], Apache's JMeter* [15], and OpenSTA* [16].
- System monitoring tools. Use these to collect system-level resource utilization statistics such as CPU utilization (e.g., % processor time), disk I/O (e.g., % disk time, read/write queue lengths, I/O rates, latencies), network I/O (e.g., I/O rates, latencies). Examples of these tools are the Performance System Monitor from Microsoft's Management Console (known as perfmon*), and "sar/iostat" in the Linux environment.
- Application server monitoring tools. These tools gather and display key application server performance statistics such as queue depths, utilization of thread pools, and database connection pools. Examples of these tools include BEA's WebLogic* Console and IBM's WebSphere* Tivoli Performance Viewer.
- Database monitoring tools. These tools collect database performance metrics including cache hit ratio, disk operation characteristics (e.g., sorts rates, table scan rates), SQL response times, and database table activity. Examples of these tools include Oracle's 9i Performance Manager and the DB/2 Database System Monitor.
- Application profilers. These provide the ability to identify application-level hotspots and drill down to the code-level. Examples of these tools include the Intel® VTune™ Performance Analyzer [17], Borland's Optimizeit* Suite [18], and Sitraka's JProbe* [19]. A new class of application response time profilers is emerging that is based on relatively modest intrusion levels, by using bytecode instrumentation. Examples of these include the Intel VTune Enterprise Analyzer [20] and Precise Software Solutions Precise/Indepth* for J2EE [21].
- JVM monitoring tools. Some JVMs provide the ability to monitor and report on key JVM utilization statistics such as Garbage Collection (GC) cycles and compilation/code optimization events. Examples of these tools include the "verbosegc" option, available in most JVMs, and the BEA WebLogic JRockit* [22] Console, depicted in Figure 4.

An important issue to keep in mind when using the above tools is that the measurement techniques employed introduce a certain level of intrusion into the system. In some cases, the intrusion level is so great that the application characteristics are altered to the extent that they make the measurements meaningless (i.e., Heisenberg problem). For example, tools that capture and build dynamic call graphs can have an impact of one or more orders of magnitude on application performance (i.e., 10-100X). The recommended approach is to only activate the appropriate set of tools based on the level the data analysis is focused on at the time. For example, for system-level tuning, it only makes sense to engage system monitoring tools, whereas application-level tuning may require the use of an application profiler.

Additional application tuning methodology information can be reviewed in [23].

SYSTEM-LEVEL PERFORMANCE

It is possible to identify two broad classes into which software can be bucketed, batch processing and interactive processing. For the former, the raw throughput, the amount of work done in a period of time, is the only real metric of interest. The time spent on any specific unit of work is not a significant consideration. For the latter class of software, the response time, the time taken for each unit of work, is very important, and in some cases it may be of higher importance than the throughput.

Architecting the system and application for good performance goes a long way towards making the rest of the performance optimization methodology more efficient. It is important to understand the type of the application

* Other brands and names are the property of their respective owners.

Intel® VTune™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

(batch or interactive) and to identify system hardware and software components that meet that goal [24].

Any system can be visualized as a network of components with transactions passing through them. In enterprise Java applications, such components could be viewed as hardware, software, or a combination of the two. For example, the network or disk sub-system is a hardware component, the application software is a software component, and the database server is a combination of hardware and software. Viewing the whole system as a network of components is useful in understanding the capacity requirements of system components.

Multi-processing, the capability of a system component to work on more than one request at the same time, plays a big role in the performance of most components. Two methods of multi-processing are pipelining and parallelism.

Pipelining is the concept of breaking down the required work into many parts. While one section of the component is working on one part of one transaction, other sections of the component can be working on other parts of other transactions, thereby maximizing use of system components. Pipelining is extensively used to increase throughput, but its effect on the time taken for an individual transaction is not a primary consideration.

Parallelism throws multiple resources at a task so that the task completes faster. Its primary effect is to reduce response time. Multi-threaded code is a way to achieve this in software. Hardware examples would include mirrored disks and multiple network cards.

Block diagrams that show the work-flow and identify the network of queues and parallel entities are very useful here. They are especially valuable in ensuring that sufficient capacity is designed into the system to meet the desired throughput and response time goals. Most systems typically use both multi-processing approaches.

Theoretically, the only system with no performance bottleneck is designed such that every component of the system exhibits the same performance behavior and has identical capacity. In practical terms, every system has a performance bottleneck.

At the system level, the goal is to ensure that the bottleneck is in the application code over which the developer has direct control, so that changes can be made that directly improve performance. If the bottleneck was elsewhere in the system, then even large-scale performance improvements in the developer's code may have only a slight effect on measured system performance.

Drawing a throughput curve can be very valuable in understanding system-level bottlenecks and helping identify potential solutions. Figure 5 shows a conceptual diagram of a throughput curve, which plots system throughput, response time, and application server CPU utilization as functions of the injection rate, i.e., the rate of requests applied to the system.

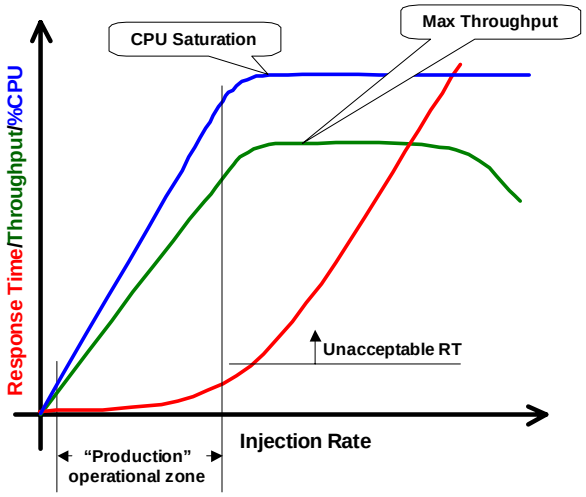


Figure 5: A conceptual throughput curve, which plots system throughput, response time and application server CPU utilization as functions of the injection rate, i.e., the rate of requests applied to the system

Through system-level tuning, the main goal should be to saturate the application server CPU (i.e., 90-100% utilization). Reaching maximum throughput without full saturation of the CPU is an indicator of a performance bottleneck such as I/O contention, over-synchronization, or incorrect thread pool configuration. Hitting a high response time metric with an injection rate well below CPU saturation indicates latency issues such as excessive disk I/O or improper database configuration.

Reaching application server CPU saturation indicates that there are no system-level bottlenecks outside of the application server. The throughput measured at this level would point out the maximum capacity the system has within the current application implementation and system configuration parameters. Further tuning may involve tweaking the application to address specific hotspots, adjusting garbage collection parameters, or adding application server nodes to a cluster.

Keep in mind that reaching CPU saturation is a goal for the performance tuning process, not an operational goal. An operational CPU utilization goal would be that there is sufficient capacity available to address usage surges.

Knowing the workload well is an important factor in the identification of the required capacity of many components. Some preliminary measurements and characterization can help. For instance, identifying the network bandwidth required for one unit of work will be

very useful in estimating the network capacity required when the desired system performance is N units of work.

Most components exhibit an exponential response time/throughput behavior. In other words, increasing the throughput will tend to increase the response time, and the higher the throughput, the faster the increase in response time. It is important to size these components such that the required throughput utilization for the component is relatively low to allow for the response time to be relatively small as well. This is especially important for network capacity, disk capacity, and the capacity of the data bus connecting processors to memory and I/O.

If the response time is an important aspect of the application, then low resource utilizations are particularly necessary; otherwise, higher utilizations will be acceptable. The precise thresholds that mark a utilization level as having hit a bottleneck depend on a variety of factors, and they are best identified through targeted experiments with test workloads.

System monitoring tools can be used to track system performance metrics, which can help find bottlenecks. In a multi-tiered system set-up where multiple computers are used, it is important to run these tools on all of the computers.

Key performance events that should be monitored include processor utilization, time spent in the kernel, interrupts, number of calls to the kernel (system calls), page faults, disk I/O, and network usage.

A key component of enterprise applications is a back-end relational database, as it provides essential persistence services, data retrieval capabilities for downstream systems, as well as support for querying and reporting applications. The back-end relational database is often a source of performance bottlenecks, because it manages large volumes of high latency disk I/O operations. It is, therefore, extremely important to pay special attention to the physical design and tuning of the database, to ensure acceptable levels of performance. Fundamental considerations include isolating log files to dedicated devices to reduce conflicts between the sequential nature of log operations and random access to data tables; adequately sizing the sort area memory size to minimize disk sort operations; allocating sufficient database cache memory (but avoiding swapping); carefully defining indexes such as indexing frequently used, highly selective keys, indexing foreign keys frequently used in joins, using full-text retrieval keys where appropriate; and using disk striping (e.g., RAID 1+0) to spread I/O operations and to avoid device contention.

Case Study 1: Database Tuning

The scenario described here was a performance issue related to database disk I/O, which is a common source of bottlenecks. In this case study, the system failed response-time requirements. Although throughput could be increased, response time increased as well. Also, the CPU was not fully utilized on the application server at maximum throughput and within response-time constraints. The supporting data included a high % disk time and long disk queues, high latencies (as seen in seconds per transfer), heavy log write activity, and excessive I/Os at some physical disks. The database used was Oracle, and Oracle statistics were helpful in pinpointing the source of the problem. In this case study, the data pointed to disk contention associated with the database log write operations. The solution was to isolate log files to dedicated devices to remove the perturbation of log write operations on other database activities, and to strip the log device to spread the I/O over multiple disks and reduce the associated latencies.

APPLICATION-LEVEL PERFORMANCE

Application design is one of the most important considerations for good performance. A well designed application will not only avoid many performance pitfalls from the start, but will be easier to maintain and modify during the performance testing phase of the development life-cycle.

Many J2EE application development best practices are well documented in design patterns [25] [26]. Design patterns provide a starting point for application design approaches that capture commonly encountered application functional requirements and usage scenarios. Several design patterns have positive performance implications, in addition to the associated maintainability and modularity benefits.

The following design patterns should be considered due to the clear performance advantages they provide:

- Composite Entity. This pattern provides mechanisms to implement coarse-grained entity beans that manage a set of subordinate persistent objects. It is used to limit the proliferation of entity beans and reduce the number of fine-grained remote calls.
- Value Object. This pattern assembles data requests into aggregated data objects to reduce remote calls to individual field get methods. It reduces the number of fine-grained remote calls and allows the transfer of more data with fewer remote calls.
- Session Façades. This pattern encapsulates business logic and data access using well-defined, coarse-grained, service-level interfaces to clients. It

eliminates the need for clients to access fine-grained business and data objects, thus reducing the number of remote calls. It is often used in combination with the Value Object pattern.

- Service Locator. This pattern encapsulates access to directory access through JNDI and provides caching of retrieved initial contexts and factory objects (e.g., EJB Homes). It reduces expensive accesses to JNDI by implementing caching strategies.
- Value List Handler. This pattern encapsulates access and traversal of database-generated lists of items. It improves performance by providing low-overhead list population mechanisms and implementing caching strategies.

In addition to design patterns, there are a number of programming practices that reduce performance bottlenecks, such as the following:

- Enterprise JavaBeans* (EJB*) homes and data sources should be cached to avoid repeated JNDI lookup of EJB objects and data source objects.
- Use of HTTP sessions should be minimized and used only for state that cannot realistically be kept on the client.
- Java Server Pages (JSP*) create HTTP sessions by default. This should be overridden (i.e., session="false") when not needed, to prevent inefficient use of session resources.
- Database connections should be released when not needed as unreleased connections result in resource leakage problems.
- Unused stateful session beans should be removed explicitly, and appropriate idle timeout seconds should be set to control stateful bean life cycle to conserve scarce resources.

A strategy frequently used to improve the responsiveness and scalability of enterprise applications involves the use of asynchronous messaging, either through the use of message-driven beans or via JMS directly. Asynchronous messaging can be used to implement high latency business operations that do not require instantaneous processing, such as order requests or document submissions, thus increasing the responsiveness of the system [27]. Messaging can also be used to break down complex business operations in message processing pipelines, which can be parallelized by instantiating multiple message queue consumers. This enhances the scalability of the system by enabling multi-threading with minimal data sharing requirements.

While good practices are a good starting point for a high-performance system, they alone are not sufficient. The workload itself still plays an important factor in performance and it may demand a specific optimal application server configuration. Many parameters can be tuned to optimize for both response times and throughput, as reducing response time can often help increase the capacity for a further increase in throughput. However, when the response times are broken into sub-components, it is necessary to further tune the system so that the response times of key sub-components are optimized too.

Many of these tunable parameters are easily accessible from common application servers such as the BEA WebLogic server. The lists of parameters presented here to help improve performance are not exhaustive. They are merely good starting points to tune the performance for your enterprise Java applications. The list includes tuning key application server parameters and tuning key container parameters.

Tuning Key Application Server Parameters

Many application server parameters can be tuned to enable better sharing and interaction with virtual machines and operating systems. The following parameters should be considered for most applications.

- Platform-optimized socket multiplexers should be used to improve server performance for I/O scalability, because they overcome performance limitations of the blocking nature of Java I/O APIs prior to version 1.4. For example, when a performance pack is available from a vendor, it should be used.
- A thread pool size should be gradually increased until performance peaks. However, one should not make the number too big as a higher number may degrade performance. The optimal number is likely dependent on the workload and the performance metrics.
- An application server may support the notion of multiple queues for transactions, e.g., by allowing the application developer to choose different values of thread pool sizes for those queues. One may find a specific distribution of execute threads to optimize for a specific workload. This is particularly important when certain transactions have tight response-time limits, and more threads for those transactions can be allocated accordingly. The support of multiple queues has an advantage over a single queue mechanism for shifting long response-time transactions to less critical areas.

- The database connection pool should be set equal to the number of available execute threads so that an execute thread does not need to wait for a connection.
- Experimenting with a JDBC prepared statement cache may yield a configuration that minimizes the need for parsing statements on the database. The value should be gradually increased until performance peaks.

Tuning Key Container Parameters

Many container parameters can be tuned to deploy an application more effectively to an application server. The following parameters should be considered for most applications.

- Setting appropriate session timeouts for the interval of time after which the HTTP session expires and for the idle timeout seconds to control stateful bean life cycle helps performance by making more efficient use of memory and other application server resources.
- Setting a good value for the initial bean pool size improves the initial response time for EJBs as they are pre-allocated upon application server startup.
- Setting optimal value for bean cache size will not let the server passivate beans too often, thus increasing performance by reducing file I/O activity.
- The least restrictive but valid transaction isolation level should be applied to specific EJB methods for good performance.
- Using call by reference when applicable increases the performance of method invocation.
- Configuring JSP fragment or full-page caching with appropriate, application-dependent timeout values to reduce dynamic page generation and database access requests.

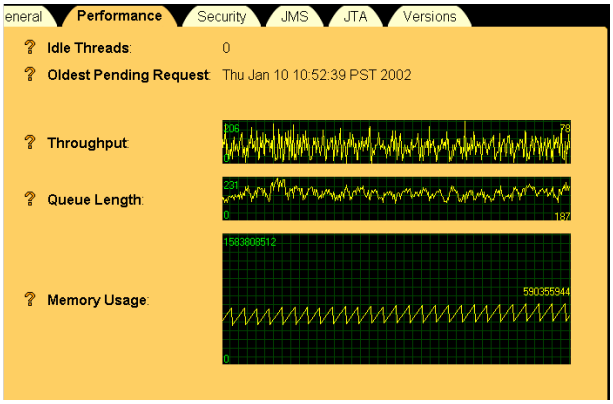


Figure 6, Case Study 2: Before application tuning. The middle chart for "Queue Length" contained values as high as 231. The queue build-up suggested that perhaps there were not enough threads to do work.

Case Study 2: Application Tuning

As an illustration of tuning application server parameters, a workload was studied using the BEA WebLogic server console. Before any tuning was applied, it was observed that while the maximum throughput was reached, the response time increased heavily with the load on the system but the processors were not fully utilized. The execute queue size was 15 while the average CPU utilization was about 70%. Using the console (see Figure 6) provided by the vendor, a high number of waiting requests was found. A considerable queue build-up was seen by the middle chart for "Queue Length." After the disk and network I/O bottlenecks were ruled out, it was decided to increase the execute queue size and see if the performance improves. At the execute queue size of 35, average CPU utilization reached 95% and the throughput could be increased by 40%. Figure 7 shows the performance of the system through the console. The queue build-up problem was much reduced as a result of the tuning effort.

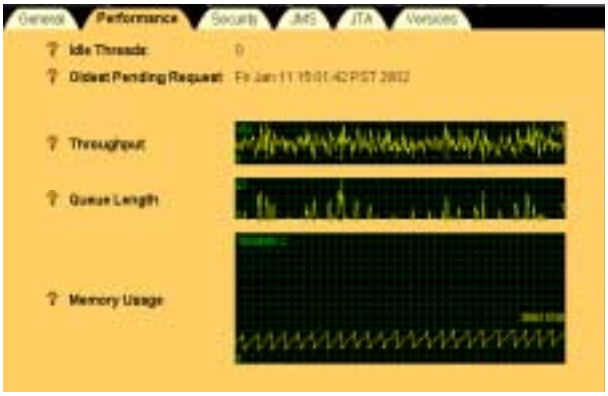


Figure 7, Case Study 2: After application tuning. The middle chart for "Queue Length" contained values mostly close to zero. The lack of queue build-up suggested that adequate threads were allocated to do work.

MACHINE-LEVEL PERFORMANCE

For applications developed using static environments such as C or C++, machine-level performance involves tuning the code for the hardware through recompilation, which complicates enterprise application deployment. With Java though, there is an additional layer – the virtual machine. This is significant for two reasons. The Java Virtual Machine (JVM) allows the application to take quick and effective advantage of new processor features since this involves only the deployment of a new JVM version and not an expensive rebuild of the entire application code.

Second, multiple versions of the application are not required to get best performance from differences in the platform such as available memory or cache. Such aspects

of the hardware are abstracted away by the JVM, and the very same version of application code can get optimal performance on the different platforms through JVM configuration tunables.

JVM-Level Performance

Selecting the correct JVM is critical. It is essential to use a JVM that has been optimized for the underlying hardware of choice. The best optimizations for various processor platforms are known [28] [29], and a Java application needs to rely on the JVM to harness these optimizations. It is also desirable that the JVM provide a rich set of configuration tunables that can be adjusted for peak performance. A JVM that can tune itself for good performance is an asset, since it simplifies deployment on a variety of platforms using the same architecture.

There are three JVM functions of interest to us: memory management, code generation, and thread management.

Memory Management

Memory management includes object allocation, heap management, and garbage collection. Modern JVMs use a variety of algorithms for these; some incorporate several algorithms for each and allow the user to select the desired one. The correct choice of algorithm is important since there are fairly significant performance differences between them. However, different techniques work better for different applications.

There are two aspects to object allocation: the management of the heap to identify the space where the object should be allocated and the preparation of the space for object allocation. The latter principally involves clearing the space by writing zeroes to it.

In a multi-processor system, the heap is shared across the processors, and obtaining space for allocation has to be a synchronized operation to ensure that no other processor is allocating an object to the same space. All synchronization is necessarily expensive, but it is especially so if the synchronized primitive is contended. A solution used by several JVMs is to allocate segments of the heap that are local to each thread, and these segments are called thread local areas (TLAs).

There are several choices for clearing space required for objects. One technique is to clear the whole TLA when it is allocated. This has the disadvantage of slowing down the object allocation that triggered the creation of a new TLA, but has the advantage that all subsequent objects created in that TLA will be able to allocate faster, since the space is already cleared. It also has the advantage of improving cache performance, since the clearing of the space serves as a prefetch from memory into the processor caches.

A second technique is to prepare the TLAs by clearing them during garbage collection (GC). It worsens the impact of GC, but all object allocation is now uniform. Finally, a third technique is to clear the space required for each object in the TLA just before allocation. This has the advantage that the object allocations are more uniform, and that neither GC nor TLA allocations take longer.

BEA JRockit, for instance, offers all three options as “cleartype:local,” “cleartype:global” and “cleartype:gc.”

The correct technique to use depends on the application. For instance, if the application puts a lot of pressure on the data bus to memory (FSB or front-side bus), then the improved cache performance of the first technique could be valuable. If uniform response time is a concern, then the third technique may be the correct choice.

There are similarly several approaches used in GC. Key aspects to consider are whether the heap should be arranged in generations and whether a significant part of GC should run concurrent with the application. It is imperative for performance that most if not all of the GC be multi-threaded. When the heap is arranged in generations, most of the GC cycles collect garbage only in the smaller nurseries, and this method is therefore not as intrusive to application performance. However, garbage is collected more frequently. Generations are effective when many objects die young, because then the GC in the small nurseries is particularly efficient. On the other hand, when GC is run concurrent with the application, the effect of each GC cycle is reduced, at the cost of a more frequent gradual impact.

Code Generation

There are two main approaches to code generation: interpreting and compiling (with a Just-in-Time (JIT) compiler). An interpreter translates each new bytecode to machine code just before execution; a compiler translates a whole segment of code (the whole application, a class, a set of classes, a set of methods, even a single method) into machine code before use.

Code compilation takes time and happens during application runtime, and the time taken to generate the code can have an impact on performance. However, the quality of code produced by the JIT is significantly superior to interpreted code, and the performance benefits of the better code should far outweigh the negative effect of compile time [30].

However, the time spent in the JIT is still an issue, and the JIT cannot therefore include all of the optimizations that a C/C++ compiler could include. This results in code that is inferior to what could have been produced if compile time was not an issue.

The solution to this is for the JVM to incorporate levels of optimizations in the JIT [31]. In other words, some portions of the application are compiled to very high quality code, whereas the remaining portions of the application are compiled quickly to lower performing code. If the portions of the application that are most used are compiled well, this will result in the whole application performing almost as well as if the whole application had been compiled thus, and it will not suffer the performance loss due to long compile times.

Many enterprise-class Java applications tend to have a large number of small methods. SPECjAppServer2002 [32], for example, has over 10,000 methods. Many compiler optimizations have a more significant impact on performance if they can operate on a larger block of code, which is impeded by the small methods. A good JIT should possess an excellent inliner to overcome this. Inlining of code during compilation is made more difficult by the large number of virtual calls in enterprise Java applications. The JIT must include approaches to de-virtualizing these calls [33].

Thread Management

A JVM either uses the threading package provided by the operating system (native threads) or it can use its own threading package and map several threads onto each kernel thread (thin threads). If the application suffers a lot from context switches, then the cost of that can be reduced by using thin threads. Similarly, if there is a pool of threads that operate on the same data, then cache performance can be enhanced by tying all the threads onto a single kernel thread. This will result in all of these threads tending to run on the same processor and benefiting from the shared data.

How a JVM handles synchronization plays an extremely important role in performance. The best way to handle a lock is to avoid locks all together, and it is good for the application developer to avoid unnecessary synchronized methods and blocks of code. In the event that such unnecessary synchronized code does exist, JVMs can possess techniques to detect them and eliminate the locks.

JVMs handle contended and uncontended locks differently, optimizing such that the uncontended lock operations go faster. The choices the JVM makes as to how it handles uncontended (thin) and contended (fat) locks, when it will promote a lock from thin to fat and when it will deflate a fat lock to a thin, and whether it will spin on a contended lock or switch over to another task can all impact performance [34].

JVM Configuration

Due to the range of choices that can be made by the JVM, a JVM can provide configuration parameters to the users to let them identify which techniques the JVM should use for optimal performance of their application.

The more important of these parameters are all in the area of heap management, ranging from the selection of the GC algorithm and the specification of heap sizes, to the specifics of TLA sizes and when the space for an object is cleared. It is usually preferable to set the minimum and maximum heap sizes to be the same. The selected heap size can have a profound effect on performance.

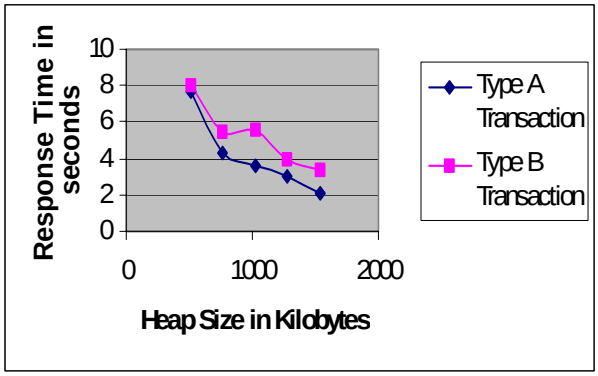


Figure 8: A variation in transaction response time for two types of transactions by changing the maximum virtual machine heap size in a typical application

The code generation can also be accessed through parameters that can decide the initial code quality and how frequently the JVM checks to see whether better code generation is warranted for a section of code, and so on.

While rules-of-thumb can be created and experience can be a guide, there is no real substitute for running a variety of experiments to identify the JVM parameters that work best for a given application.

An analogy can be made between the transmission systems in a car and JVM performance. In most cases, an automatic transmission performs adequately. For high-performance requirements such as auto-racing, however, a manual transmission works better, since, in the hands of a good driver, better performance can be had from a manual transmission. Similarly, while a good JVM will provide excellent performance as it is, appropriate selection of configuration parameters can result in even better performance.

Case Study 3–JVM Tuning

As stated earlier, heap size configuration can have a dramatic performance impact. Figure 8 shows the variation in transaction response time for two types of transactions in a typical enterprise Java application. By increasing the heap size in this experiment, we observed a two to four times improvement in response time, depending on the type of transaction.

Hardware-Level Performance

There are several hardware aspects that affect performance, including processor frequency, cache sizes, Front Side Bus (FSB) capacity, and memory speed [35]. In general we would like to get the best-possible performance on a single processor, then scale that performance across multiple processors in the box, and if more performance is desired, scale the performance out of the box by using clustering.

It is important to ensure that the settings for the BIOS and the populating of the memory sub-system follow prescribed norms. For example, a platform with 4 gigabytes of memory may perform better with four 1-gigabyte memory cards rather than with one 4-gigabyte memory card. Reading and following the system documentation can pay dividends.

Processor performance is affected most by the processor stalling, waiting on memory. The memory subsystem comprising the FSB, the server chipset and the memory cards, is typically an order of magnitude slower than the processors, and so the penalty of accessing memory for data and instruction is felt rather severely by the processors. Keeping more of the code and data near the processor, by using large caches, can alleviate this problem significantly [36].

As an experiment, we measured the performance of a typical enterprise Java application on a two-processor Intel® Xeon™ MP 2.0 GHz system with a 2 MB level-3 cache as well as on a two-processor Intel Xeon DP 2.2 GHz system without a level-3 cache. Both systems had the same amount of memory, the same operating system and application software, the same I/O connectivity, and they both had processors that included a 512 KB level-2 cache. The main difference between the systems was that one of them included an additional level of caching, a 2 MB level-3 cache. We found that having the large level-3 cache almost doubled the performance of the application.

Scaling to multiple processors in a box can be hurt by resource sharing. When the resource is a hardware resource such as the bus, larger caches can help. If larger caches are not available, or do not help, investigation into whether there is a significant impact on the bus due to data shared between threads is called for. If there is, then processor affinity could help.

If processor scaling is hurt due to software issues, it is typically a problem related to synchronization. All efforts must be made to identify the lock or locks that are the bottleneck, and to remove them or at least reduce their use.

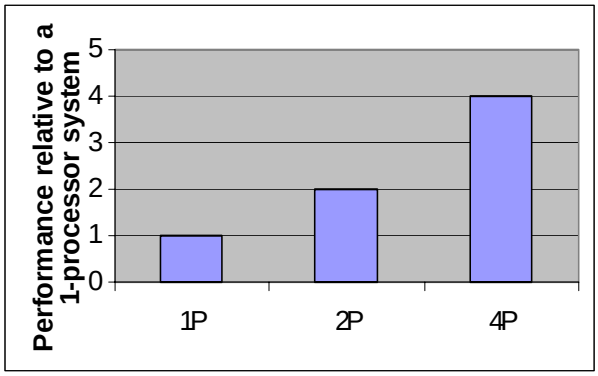


Figure 9: Performance scaling with the number of processors in a 1 GHz Itanium 2 system: performance of a 4-processor system is four times higher than a 1-processor system

One way to identify whether the processor scaling is affected by hardware or software is to measure the bus utilization. If it is high, then it most likely is a hardware resource bottleneck. Another experiment that is useful is to run the system at two different frequencies. The processor speed is now changed but not the bus or memory speed. However, the time taken to handle synchronization does scale with frequency. If now the performance scales with frequency, then it would point to a synchronization issue.

The Itanium® processor family can display excellent processor scaling. Figure 9 shows the performance scaling with the number of processors for a 1 GHz Itanium 2 system when running a typical enterprise Java application. The Itanium systems have large caches, a large capacity front-side bus, and out-of-order memory transfers, all of which enable high levels of processor scaling.

Clustering is an excellent way to increase performance when transactions share very little, frequently changed data. If there are substantial amounts of shared modified data, keeping the data coherent across the different components of the cluster will be a significant endeavor and have a big impact on performance.

It may also be possible to increase the performance within the box through clustering, by deploying more than one version of the code using multiple copies of the JVM. This has the advantage that each version of the application will run on its own copy of the JVM, with its own heap.

Intel® Xeon™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

® Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

In some platforms there is a limit placed by the operating system on heap size, and some applications do benefit from larger heap sizes. By clustering within a box, this performance benefit can be tapped.

CONCLUSION

This paper describes a top-down, data-driven, and closed-loop approach to boost enterprise Java performance. The opportunities to improve performance were examined from the perspective of the whole system, including the software/hardware stack at the system level, the software applications, and at the machine level for both the virtual machine and the physical hardware. The case studies presented in this paper suggested that all layers–not just one or two–of the system stack should be examined for performance bottleneck identification and removal.

Workloads evolved over time and no single solution works for all applications. While many good practices for enterprise Java performance were described in this paper, it is important to recognize that the performance characteristics of enterprise Java applications will change over time. The data-driven approach described in this paper should be able to adapt to the expected workload changes over time.

In the future, application servers and virtual machines may work hand-in-hand with the underlying hardware and self-tune for performance by using techniques such as dynamic feedback optimization and perhaps deriving data from some hardware performance counters. When the time comes, the approach described here may be suitable for integration into that infrastructure. Until then, our approach will help to enhance the performance of your current systems.

ACKNOWLEDGMENTS

The authors thank the members of Intel's MRTE group for performance data collection and analysis.

REFERENCES

[1] Ken Arnold, James Gosling, David Holmes, "The Java™ Programming Language Third Edition," 2000, Sun Microsystems, Inc.

[2] Java Community Process, "Java™ 2 Platform, Enterprise Edition 1.3 Specification," http://jcp.org/aboutJava/communityprocess/final/jsr058/

[3] Kevin McIsaac, "J2EE Paves the Way to Software Infrastructure," Meta Group, August 2002.

[4] BEA Systems Inc., "BEA WebLogic Server," http://www.bea.com/products/weblogic/server/index.shtml

[5] IBM Corporation, "WebSphere Application Server," http://www.ibm.com/software/webservers/appserv/was/

[6] Oracle Corporation, "Oracle9i Application Server," http://www.oracle.com/ip/deploy/ias/

[7] JBoss Group, "JBoss, " http://www.jboss.org/

[8] ObjectWeb Consortium, "Java Open Source J2EE Application Server (JOnAS)," http://www.objectweb.org/jonas/

[9] Jim Farley, William Crawford, David Flanagan, *Java Enterprise in a Nutshell*, April 2002, O'Reilly & Associates, Sebastopol, CA.

[10] Richard Monson-Haefel, *Enterprise JavaBeans*, September 2001, O'Reilly & Associates, Sebastopol, CA.

[11] Michael Girdley, Rob Woollen, Sandra L. Emerson, *J2EE Applications and BEA WebLogic Server*, August 2001, Prentice Hall PTR, Upper Saddle River, NJ.

[12] Mercury Interactive Corporation, "LoadRunner," http://www-heva.mercuryinteractive.com/products/loadrunner/

[13] RadView Software Ltd, "WebLoad," http://www.radview.com

[14] Paco Gómez, et al, "The Grinder," http://grinder.sourceforge.net/

[15] Apache Software Foundation, "Jmeter," http://jakarta.apache.org/jmeter/

[16] Various authors, "Open System Testing Architecture (OpenSTA)," http://www.opensta.org/

[17] Intel Corporation, "VTune™ Performance Analyzer," http://www.intel.com/software/products/vtune/vtune61/

[18] Borland Software Corporation, "Optimizeit Suite," http://www.borland.com/optimizeit/index.html

[19] Sitraka, Inc., "JProbe," http://www.sitraka.com/software/jprobe/

[20] Intel Corporation, "VTune™ Enterprise Analyzer, Java Edition," http://www.intel.com/software/products/vtune/vte_java10/

[21] Precise Software Solutions, Inc., "Indepth/J2EE," http://www.precise.com/Products/Indepth/J2EE/

[22] BEA Systems, Inc., "BEA WebLogic JRockit 8.0 (Beta) for Windows and Linux User Guide," http://e-docs.bea.com/wljrockit/docs80/index.html

[23] Intel Corporation, "Performance Methodology, Terminology and Concepts," http://cedar.intel.com/media/training/perf_meth/tutorial/

[24] Raj Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*, 1990, John Wiley & Sons, Hoboken, NJ.

[25] Deepak Alur, John Crupi, Dan Malks, "Core J2EE patterns: best practices and design strategies," 2001, Prentice Hall PTR, Upper Saddle River, NJ.

[26] Floyd Marinescu, *EJB Design Patterns*, 2002, John Wiley & Sons, Hoboken, NJ.

[27] Alejandro Buchmann, Samuel Kounev, "Improving Data Access of J2EE Applications by Exploiting Asynchronous Messaging and Caching Services," in proceedings of the *28th VLDB Conference*, Hong Kong, China, 2002.

[28] Intel Corporation, "Intel Itanium 2 Processor Reference Manual for Software Development and Optimization," http://developer.intel.com/design/itanium2/manuals/

[29] Intel Corporation, "Intel Pentium 4 Processor Optimization Reference Manual," http://developer.intel.com/design/pentium4/manuals/

[30] A. Adl-Tabatabai et al., "Fast Effective Code Generation in a Just-in-Time Java Compiler," in proceedings of the *ACM SIGPLAN'98 conference on Programming Language Design and Implementation*, 1998.

[31] M. Arnold, et. al., "Adaptive Optimization in the Jalapeno JVM," *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000)*, Minnesota, October 15-19, 2000.

[32] Standard Performance Evaluation Corporation, "SPECjAppServer2002," http://www.spec.org/jAppServer2002/index.html

[33] O. Waddell and R. K. Dybvig, "Fast and Effective Procedure Inlining," in proceedings of the *1997 Static Analysis Symposium (SAS '97)*, Sept. 1997, pp. 35-52. Springer-Verlag Lecture Notes in Computer Science vol. 1302.

[34] R. Dimpsey, et. al., "Java Server Performance: A case of building efficient, scalable JVMs," *IBM Systems Journal, vol. 39, no. 1*, pp. 151-174, 2000.

[35] David A. Patterson, John L. Hennessy, "Computer Organization and Design: The Hardware/Software interface," 1997, Morgan Kaufmann Publishers.

[36] C.A. Hsieh, M.T. Conte, T.L. Johnson, J.C. Gyllenhaal, and W.W. Hwu, "A Study of the Cache and Branch Performance Issues with Running Java on Current Hardware Platforms," in proceedings *IEEE Compcon '97*, pp. 211-216, 1997.

AUTHORS' BIOGRAPHIES

Kingsum Chow is a Senior Performance Engineer working with the Managed Runtime Environments group within the Software and Solutions Group (SSG). Kingsum has been involved in performance modeling and optimization of middleware application server stacks, with emphasis on J2EE and Java Virtual Machines. He has published 20 technical papers and presentations. He received his Ph.D. degree in Computer Science and Engineering from the University of Washington in 1996. His e-mail is kingsum.chow@intel.com.

Ricardo Morin is a Staff Architect working with the Managed Runtime Environments group within the SSG. Ricardo leads performance optimization activities for J2EE and JVM implementations. He has extensive experience architecting, developing, and deploying enterprise information systems. Ricardo is a Sun Certified Architect for J2EE. He received his Electronic Engineering (Cum Laude) degree from Simón Bolívar University, Caracas, Venezuela in 1980. His e-mail is ricardo.a.morin@intel.com.

Kumar Shiv is a Senior Performance Architect working with the Managed Runtime Environments group within the SSG. Kumar leads the team focusing on JVM and system performance optimization for the Itanium® Processor Family and earlier lead the team working on the Intel® Xeon™ technology. He received his Ph.D. degree in Computer Engineering from University of Missouri in 1991, and has been a performance architect on several hardware and software projects for more than a decade. His e-mail is kumar.shiv@intel.com.

Developing and Optimizing Web Applications on the ASP.NET Platform

George Vorobiov, Software and Solutions Group, Intel Corporation
Carl Dichter, Software and Solutions Group, Intel Corporation
John Benninghoff, Software and Solutions Group, Intel Corporation
Charlie Hewett, Software and Solutions Group, Intel Corporation

Index words: ASP.NET, CLR, performance, tuning

ABSTRACT

This paper discusses best practices in developing and tuning the performance on Intel architecture of Web applications based on the ASP.NET* platform. We provide an overview of the ASP.NET platform and discuss a number of optimizations that can be applied to this class of applications based on our findings from developing and characterizing an e-commerce workload. We also want to disseminate our knowledge to the industry on optimal software configuration and the use of the rich feature set provided by ASP.NET and Common Language Runtime (CLR).

Our major emphasis has been to characterize and improve the performance of these applications. To do this effectively the design of the application and configuration of the infrastructure to host this software application are discussed. The description of the application shows the components participating in request processing and response generation. We also present the analysis of performance problems and tradeoffs facing ASP.NET developers. Finally, we discuss the evolution of the workload to include different distributed computing scenarios using emerging technologies such as Web Services and .NET Remoting*.

INTRODUCTION

ASP.NET* is a new Microsoft Web development platform, which is built on top of the Common Language Runtime (CLR) platform, and it inherits many familiar features of the ASP platform. Despite the similarities in the naming and APIs exposed, these two implementations do not have much in common in terms of their underlying technology and performance characteristics. While the ASP platform used an interpreted scripting language and was limited in the way it can interact with other components of the operating system (OS), the new ASP.NET platform enables the use of a variety of different programming languages, all of which are compiled into Intermediate Language (IL) and all of which can take advantage of all the features provided by CLR and Windows .NET.

Some of the benefits of the new programming model are given below.

- There is a true separation of the presentation logic code and HTML scripting. Visual Studio .NET* facilitates this process by automatically creating a code-behind class in a language of the developer's choice that can process events and dynamically modify the page presentation, based on the current application state.
- Session state management now supports both efficient in-process session state handling and a more scalable Microsoft SQL server-based solution that allows sharing of the session state by a large number of servers in a server farm scale-out scenario.

* Other brands and names are the property of their respective owners.

* Other brands and names are the property of their respective owners.

- There is support for modular page design, using extensible server-side controls, that enables component-based programming models to easily share functionality between multiple ASP.NET pages.
- New and improved ADO .NET* data access APIs provide both fast forward-only data retrieval methods and a more advanced DataSet approach for creating an offline view of the data that can be accessed and modified independently and synchronized with the database when needed.
- There are more opportunities for efficient caching using both object cache and output caching features.

CLR is a new managed runtime platform by Microsoft that is designed to increase programmer productivity by providing a rich set of features such as automatic garbage collection, built in support for multiple remote procedure call (RPC) mechanisms, full compliance with multiple XML standards, and a state-of-the-art object-oriented programming framework. It also provides easier deployment and administration support with code verification and type safety checking, global assembly cache (GAC) for shared libraries, and code versioning support to eliminate the infamous "DLL Hell"[1] problem.

ASP.NET applications also take advantage of Microsoft's new Internet Information Server (IIS) processing model that was introduced in version 6. IIS streamlines the Web request execution by using a new 2-tier kernel mode listener/worker process model instead of the former 3-tier model.

To characterize and optimize the new programming platform, we developed an e-commerce workload modelled after a small Web-based bookstore. This workload is designed to exercise most of the major features of ASP.NET and CLR. In this paper we discuss the findings from our work on optimizing the performance of this application using the standard Intel methodology described in the optimization section.

We hope that the information in this paper will be useful to developers who want to understand system and software design issues, and their respective impact on performance. It should also be of interest to those with a current implementation in ASP or who want to weigh the cost and benefits of moving forward with the latest software technology from Microsoft.

[1] DLL Hell is often used to refer to the problem with deploying different versions of the same library that leads to incorrect execution of the programs that depend on it.

WORKLOAD DESCRIPTION

In order to successfully apply optimization techniques to a given application it is necessary to understand the design and performance characteristics of the system. This section concentrates on describing the architecture and configuration of the system as well as the hardware profile.

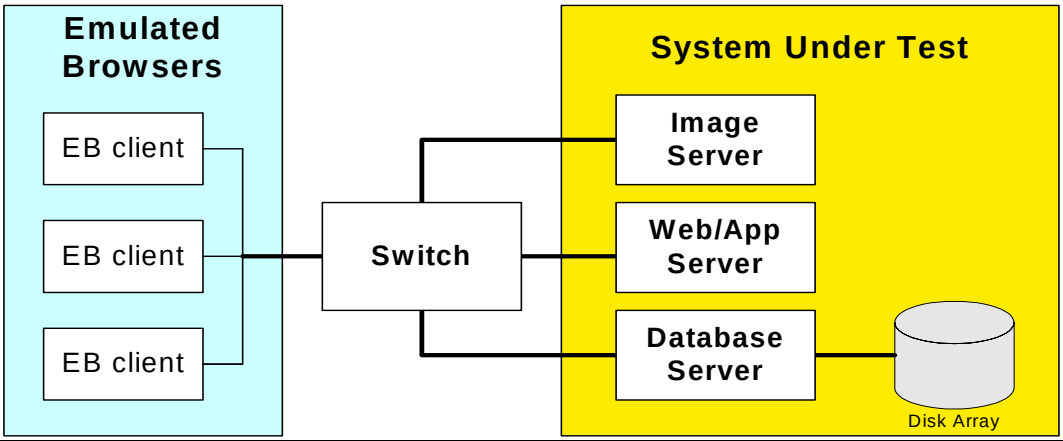
Hardware Description

Our e-commerce workload can be functionally divided into two distinct parts: emulated browsers, or EBs, and the system under test, or SUT. Every EB emulates a number of distinct users, each generating their own unique HTTP traffic to the SUT. The SUT is the collection of servers that makes up the e-commerce solution that accepts these requests. Figure 1 depicts the hardware layout.

Major components of the workload required to provide the full implementation of the application and assist in load generation include the following:

- Web/Application Server. This machine is running IIS with the actual ASP.NET application containing presentation, business logic, and data access components.
- Database Server. This contains an application database and a set of stored procedures to provide optimized data access support.
- Image Server. This Web server, running IIS, handles all the HTTP image requests from the clients.
- Emulated Browsers. These run our custom load generator tool to provide sufficient load on the system for workload characterization and performance problem identification purposes.

A typical Web request from an EB client is passed through the switch and accepted by the Web/app server, which parses the request and generates queries to the Database (DB) server if necessary. After receiving the response from the DB server, the Web/app server generates its response to the EB client. The EB client will then parse the response page and retrieve any images from the image server. Based on the response from the Web/app server and a few other conditions, the EB will randomly generate its next Web request, and the pattern repeats itself until the end of a run.



Machine	Hardware	Description
EB client	Pentium®III 2 x 1.26 GHz	Generates HTTP load to the SUT
Image Server	Pentium®III Xeon™ 4 x 900 MHz	Serves out images to the EB clients
Web/App Server	Intel® Xeon™ 4 x 2.0 GHz	Processes HTTP requests from the EB clients
Database Server	Pentium®III Xeon™ 8 x 900 MHz	Processes SQL queries from the web/app server
Switch	Extreme Networks Black Diamond	Gigabit Ethernet switch
Disk Array	EMC FC4700 Fiber Channel Storage	RAID 0 data storage for all tables

Figure 1: Hardware deployment diagram and descriptions for the e-commerce test suite

System Software Description

The workload runs as a Web application in the instance of the CLR platform (version 4322) hosted by the IIS worker process (W3WP.EXE). All machines in the SUT are running on Windows* Enterprise Server* version 3663. As seen in Figure 2 below, incoming request processing starts at the kernel mode listener (HTTP.SYS) and is then passed to the IIS worker process running our application. There can be more than one worker process if multiple Web applications are running concurrently. CLR is loaded inside the worker process, using the ISAPI extension mechanism common to all Web application types under the IIS.

The ASP.NET application is controlled by the web.config configuration file located in the application directory. This file specifies multiple aspects of the application runtime behavior, such as the type of session state handling to use (in-process is the default, but an SQL Server instance or a custom implementation can also be specified), or the custom error processing page, which provides a unified way of handling uncaught exceptions by displaying a more user-friendly message instead of a standard stack trace. You can also specify a different authentication mode, such as the use of the Windows authentication mechanism or the Microsoft Passport* service.

Since our workload is a managed application running on top of CLR, it is also controlled by the CLR machine.config file located in the config subdirectory of the framework installation. This XML file contains multiple settings for ASP.NET, such as the number of the worker threads available to process application requests and the size of the ASP.NET request queue. Optimal values for these settings are discussed later.

IIS settings are common to all types of Web applications and can be applied using the Web Administration Service, as shown in Figure 2. You can configure multiple application pools and assign different applications to a given pool. Application pool settings provide a set of configurable parameters to improve ASP.NET application robustness and performance.

* Other brands and names are the property of their respective owners.

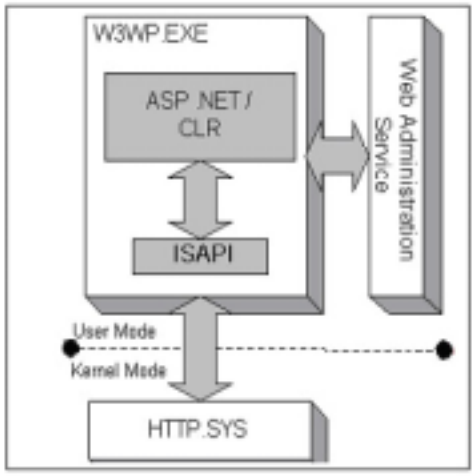


Figure 2: System software configuration

Application Description

Our workload is designed to provide a representative yet simple Web application, exercising most of the major features of ASP.NET and CLR, such as garbage collection (GC), ADO .NET data-access framework, server-side controls, and XML support. It is modeled after a small bookstore Website that provides catalog browsing, shopping-cart maintenance, ordering, and support for administrative functions. The workload logical architecture provides true separation of Web application layers by defining clear interfaces between these layers and providing implementations adhering to the defined interfaces. These logical application layers are the Presentation, Business Logic, and Data Access layer, respectively.

The design of the workload is based on the patterns and practices for developing distributed applications for .NET [5]. The main reasons to use such a multi-tiered logical structure were to make sure that the workload is representative of this class of applications and to facilitate future design changes to utilize distributed Web Services and Remoting components.

The Presentation layer of the workload consists of a set of ASP.NET Web pages (ASPXs) and custom server-side controls (ASCXs) with their code-behind classes. The code-behind classes dynamically modify the resulting document by requesting up-to-date information from the Business Logic layer and manipulating the properties of the controls on the page.

The Business Logic layer contains a set of stateless services, implementing the business interfaces that provide current application data, based on a set of business rules. This layer uses the Data Access layer to retrieve and update the data, based on parameters received from the Presentation layer.

The Data Access layer handles all data retrieval and update requirements defined by the application. It uses the ADO .NET framework to access an SQL Server database, using the .NET SQL Server driver. This layer can use either dynamically built SQL commands or a set of stored procedures residing in the database.

To better understand the system behavior, we present the sequence diagram of request processing execution in ASP.NET as shown in Figure 3. Inside the CLR, the request is handled by an instance of the HttpApplication object representing a single pipeline of execution. It maps the incoming request URL to a specific ASPX page, creates an instance of the code-behind class for this page, and executes any event handlers defined for the page or for the user-defined server-side controls defined on the page.

In most cases the application logic is executed by the Load event script. It starts by validating the input parameters and initializing data structures. Then the script requests an instance of the business logic class, specific to the request type. The application is structured so that an instance of a different stateless service class can be loaded at startup, depending on whether remote method invocation via Remoting or Web Services is used, or on whether the service runs locally. We discuss the latter in this paper, but part of the future direction of our workload is to run business and data access logic on a separate application server.

Once the request is passed to the service object, some additional business rules are applied to ensure the validity of the request, and an appropriate data access provider class is invoked. Our workload is configured to allow for different types of data access and retrieval, the two main alternatives being the use of SQL Server stored procedures and dynamically generated SQL statements. (Find out more about the comparison of these two alternatives in the analysis section.) The default for the workload is to use stored procedures for all data access. In this case, an object of the class implementing the stored procedure-based data access is instantiated dynamically at application startup.

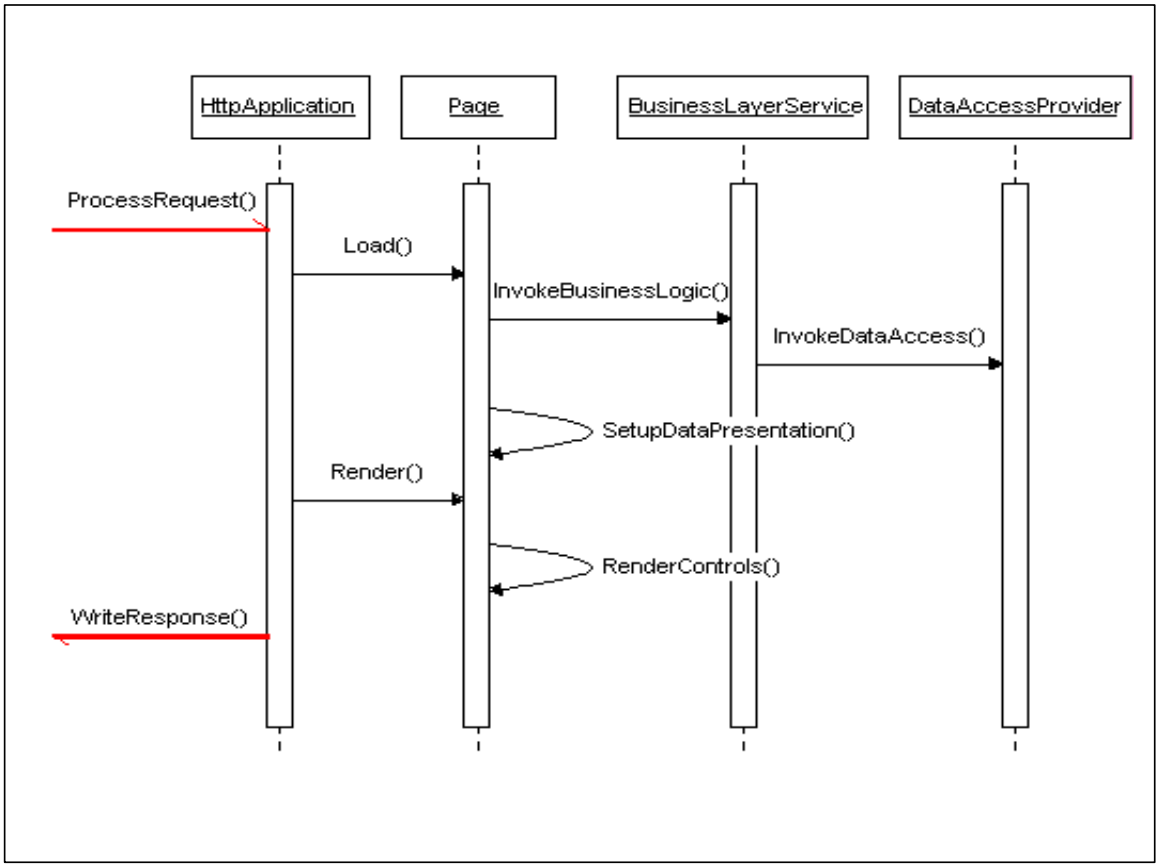


Figure 3: Workload sequence diagram

Once the stored procedure implementation is invoked, it constructs a new SQL command to call a stored procedure on the database server by using the .NET driver for the Microsoft SQL Server. The .NET driver has a built-in capability to provide connection pooling, so after the interaction with the database is completed, the current data connection utilized by the current request is returned to the pool of available connections. This helps eliminate the overhead of connection creation on every request, while freeing the developer from manually implementing connection pooling.

After the stored procedure results are received, the control passes back to the Load event script. The rest of the logic in it manipulates the controls on the page to show the results returned from the database. Once the script execution for the page is completed, the Load scripts for all custom controls on the page are invoked if necessary to execute control-level logic.

Finally, the engine proceeds with rendering the page, which involves recursively calling the *Render* method for every server-side control on the page and writing the result to the Response object output stream.

Note that if the output caching is defined for any of the controls on the page, or the page itself and the control or page output is currently in the ASP.NET output cache, then none of the event scripts will be executed for it, and the previously saved (cached) output from the page or control will be written to the output stream. Output caching is discussed in detail later in this paper.

Performance Data

Figure 4 shows the throughput scaling for one, two, and four Intel® Xeon™ 2.0 GHz processor configurations on the application server with Hyper-Threading enabled for our workload. As you can see, it scales fairly well with a two-processor scaling of 1.87, and a four-processor scaling of 3.27. A number of system- and application-level tunings were applied to enable good CPU utilization, which is critical to achieve such scaling. These optimizations are discussed in the following sections.

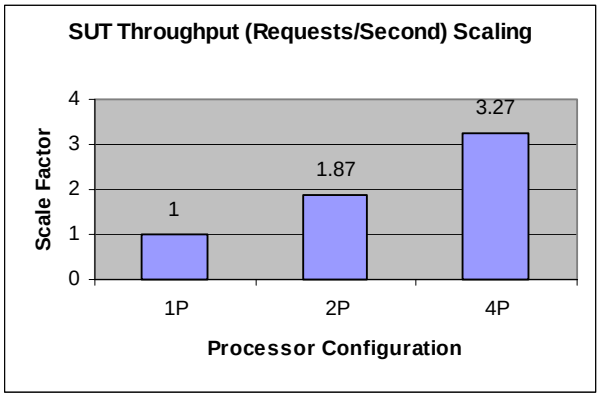


Figure 4. Throughput scaling diagram

OPTIMIZATION

The goal of applying optimizations is typically to achieve maximum throughput and/or to lower response time. Optimizing a complex system consisting of a number of hardware and software components can be difficult, but by applying a systematic approach, optimizations can be discovered and applied effectively.

The optimization process begins by collecting data on the system and analyzing the data. A performance issue is identified, and alternatives are proposed and explored. Next, a solution is applied to the system, and data is collected again to analyze the performance difference. The change is then accepted or rejected and the performance analysis cycle begins again.

In addition to this iterative approach, a top-down methodology is used. There are three levels of optimization: system, application, and micro-architectural. The top-down methodology means that tuning begins with higher impact changes such as I/O and database configuration at the system level. Only after the bottlenecks have been removed from the higher level of optimization, can the next level of optimization take place. This means that system-level optimizations have to come before application-level optimizations, which have to come before micro-architecture-level optimizations.

Furthermore, after applying a change at a lower level, you must go back again to the top level and begin the optimization again. For instance, an application-level change could help performance, but also expose a system-level bottleneck that then needs to be alleviated. Generally, when optimizing a complex system, a majority of the time is often spent identifying system-level issues.

Each different class of tuning requires a different set of tools and performance tests. For instance, system-monitoring tools such as Microsoft's Perfmon* are excellent at finding system-level issues, but will not find micro-architectural-level issues. The Intel® VTune™ Performance Analyzer is an excellent performance analysis tool, providing Call Graph, Counter Monitor, and other valuable data. It is also flexible enough to be applied in system-, application-, or micro-architecture-level tuning, but it may not provide the necessary specific data. For instance, if you know you have a network issue, Microsoft's Netmon* tool can be used specifically for that purpose.

In tuning an application, you will need to apply a systematic, top-down approach, and use the correct tool to analyze the performance of your system. The following sections will explain system- and application-level optimization. Although it is possible to achieve performance benefits with micro-architectural tuning, they are typically smaller and not within the scope of this article.

A great deal of this paper is dedicated to caching. This is because it is the single biggest source code or configuration file optimization that you can perform in most Web applications.

System-Level Optimizations

Our methodology is to look at the system-level issues first, making sure that the performance bottleneck is not in the physical capacity of I/O devices, disks, and network, etc., but in the application- and system-level code. Once the system-level optimizations are applied, we can proceed with our top-down methodology, analyzing performance and applying changes to the application, configuration, and tunable parameters.

To apply system-level tuning we utilized Perfmon*, Microsoft's performance-monitoring tool. When using Perfmon, a number of counters are selected to monitor such metrics as Processor, Network, and Disk utilization, as well as more application-specific counters such as "ASP.NET\Requests Queued" and ".NET CLR Memory\Number of Bytes in all Heaps." Perfmon, along with some knowledge about the physical limits of the components you are studying, can help diagnose many system-level issues.

---

* Other brands and names are the property of their respective owners.

Intel® VTune™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

For our workload, we found network bottlenecks in both our application and image servers. By looking at the Perfmon logs, we were able to ascertain that we were packet limited on the application server, due to the large number of requests entering from the emulated browsers (EBs), and that we were throughput limited on the image server, due to the large size of some of the images being accessed. In each case, the problem prevented us from saturating our application server processors. Installing Gigabit (1000Mbps) Ethernet cards in both the application and image servers alleviated the network bottlenecks.

Database optimizations were also necessary, including the addition of indices and tuning of specific database parameters. In the early stages of our workload development, we found that the disk I/O subsystem on the database server was overutilized. Moving the database and log files to physically separate disks helped, but did not completely alleviate the problem. In order to reduce latency and alleviate the disk bottleneck, it was finally necessary to move the database and log files each to their own RAID 0 disk array.

It is obvious that without applying system-level optimizations such as those described above, application-level tuning will not provide any benefit. For example, if the processor is busy only 10% of the time, making the code twice as efficient would only result in a modest performance improvement. However, if the processor is busy 100% of the time, code efficiency improvements result in significant performance improvement. The goal of system-level tuning is to ensure that the bottleneck lies in code that the developer controls, such that application-level optimizations can then be applied.

Application-Level Optimizations

After system-level bottlenecks have been alleviated, developers can further increase performance through changes in the application, including caching, data access optimizations, and application tunable parameters. The following is a discussion of some caching and tuning options provided by ASP.NET and ADO.NET.

In our efforts to optimize the workload, we concentrated on achieving better use of the platform features and not improving the efficiency of our application code, since our application code only consumes less than 1% of the CPU, as measured by the Intel VTune Performance Analyzer. The main reason for this is the fact that the platform automates most of the common programming tasks that used to be handled by the application code, such as memory management or session state handling.

Caching

Popular Web sites receive millions of hits per day. Normally, that means many millions of disk accesses hit their databases, large amounts of processing are used to serve their application, and possibly other systems are used for Web services or other remote objects.

What would happen if every request for a Web page could be met by sending the image of a page already created? In this ideal situation, the Web server would only have to serve up existing pages from memory, along with statically included information such as pictures. This integrated cache is part of the power of Microsoft's .NET. It has the ability to cache entire Web pages, fragments of a Web page, or any object.

Overall, there are two types of caching: output caching and data caching. Output caching can be either caching of entire Web pages or fragments of Web pages. This caching in .NET is unlike caching of static Web pages because it works with dynamic content (pages built with ASP.NET). Each time a page (or a fragment of a page) is requested with the same dependencies as another recent request, the information is delivered from cache. (Dependencies and expiration periods are more fully described later.)

Data caching, which is also called Object Caching, is the ability to cache the output of any method. Typically it is used for methods with a high-performance cost, such as methods that access a database. Until .NET, many serious applications would build their own data cache, but few did it well.

If properly used, .NET's caching options can provide huge performance improvements. Proper use of the cache results in less object creations, less processing, and most important of all, lower dependency on slow system resources (such as disk I/O) or external facilities such as databases, remote objects, or Web services [1].

We now provide a detailed analysis of how different types of caching can be applied to a Web application to improve its performance.

How to Use .NET's Object Cache

Using the object cache is easy; using it properly is difficult.

We cover the basic syntax for using the object cache. More detailed information about how to use the cache can be obtained from the Microsoft Developer Network (MSDN) and *GotDotNet* (www.gotdotnet.com) Web sites.

The object cache can be used wherever a method returns an object. There are also several forms for inserting data into the cache, including versions that allow you to set

expiration times for the item. In our example, the cache has a simple key as the parameter to the *fetchMyData* method.

```
DataView Source;
// See if the object is in cache
Source = (DataView)Cache["MyDataKey"];
// If object isn't in cache, get
if(Source == null )
{
        Source = fetchMyData("MyDataKey");
        // Put data in cache
        Cache["MyDataKey"] = Source;
}
// use the data
MyDataGrid.DataSource = Source;
MyDataGrid.DataBind();
```

The cache key must contain all the dependencies that would affect the output of the method to be cached. The key is always derived from some or all of the parameters to the method being called; however, it may also use other variables that can affect the result. For example, fetchMyData might contain internal state information – such as my user information – such that it would return a different value for another user. In this case, it is prudent to include the user ID as part of the key to the cache.

In addition to simple variables, dependencies can also include files, directories, or the keys for other objects in the cache.

Incorrectly identifying the dependencies for caching can result in incorrect program output or low cache hit rates. Determining these dependencies requires a programmer's skill, i.e., don't expect to see tools to automate the process anytime soon.

Proper Use of the Object Cache

Using the cache properly to optimize performance is not a simple matter. There are two performance problems that are prevalent in many .NET applications:

- Under-caching. To under-cache is to fail to take advantage of caching where it would be a benefit.
- Over-caching: Using the cache where you shouldn't is known as over-caching.

Under-caching is caused by the fact that unlike a processor cache, .NET caching is not something that happens automatically. If you haven't implemented caching, you are missing out on performance.

The second problem, over-caching, is equally serious. On applications that abuse the object cache, the object cache itself becomes a major bottleneck that limits the performance of the application.

Part of the problem with over-caching is that putting something in cache might imply "kicking-out" something else that is needed. With .NET, a bigger part of the problem is that the cache object is a synchronized (thread safe) collection; therefore, it can become a major source of contention when multiple threads are trying to access it at the same time.

Proper use of the .NET cache includes putting all things in cache if they will be used again, and only if they will be used again.

Finding Under-Caching and Over-Caching

The best way to find these problems would be to have a tool do all the work; unfortunately, that is not possible today. These tools are probably coming in the near future, but they will require additional instrumentation in the .NET framework.

In the meantime, here are some strategies to consider and some tips on how each might be applied in various situations.

- Start from the ground up: implement caching only where appropriate.
- Observe what is removed from cache.
- Manually instrument every use of cache to determine hit/miss rates.

What follows is a description of each approach and an analysis of each from the perspective of accuracy, ease-of-use, and ease-of-implementation.

Start From the Ground Up

In this approach, start without any caching, and then use strategies to implement caching only where it is strongly indicated.

This approach is applicable if you have not implemented any caching or when you have caching that may be improperly implemented. It is easier to find under-caching than it is to find over-caching.

The implementation effort is very high with this approach, especially for programs that already have caching. First, you must remove (comment out) all caching, then you must use a methodology to decide where to cache, and finally you must re-implement caching where appropriate.

One way to find out where to cache is to start with the objects that have the highest creation rates, implement caching there, and observe the hit rate.

When applying this approach, keep the following in mind:

- String and other objects will show up as high usage in many places. You must determine whether the same data are ever likely to be returned, or time will be wasted implementing caching where the hit rate will be low.
- Incorrectly implementing caching can result in incorrect program output, or low cache hit rates. You must correctly identify dependencies to implement proper caching; unfortunately, determining dependencies is not a trivial task. Dependencies may be more than just parameters to the method; they may include files or time information.

Ideally, you should only cache where it will yield the most benefits. In reality, your results will depend a lot on the skill and effort of the implementer. Intel® Solution Services has helped many customers determine appropriate caching schemes for their applications, often improving application performance in the process.

Observe What is Removed From Cache

If the code already makes extensive use of caching, use this technique to determine if you are caching when you shouldn't or have implemented caching incorrectly.

For this technique, you can use the *CacheItemRemovedCallBack* to tell when an item is removed and to track the type of objects being evicted. The most evicted objects might represent objects that were placed into the cache but never used because unused items in cache will get evicted due to the Least Recently Used algorithm the cache employs.

There are three problems with this technique. One is the complexity of implementation. You'll have to implement cache (with the *CacheItemRemovedCallback*) in all the places where it is likely to be helpful. You will also need to implement the actual method that will be called by the callback mechanism. You will then have to figure out where in the program these callbacks are happening. Unless you use a different callback, you will only have the name and value of the key to guide you.

Another problem is there is no way of knowing how long these items were used in cache before they were evicted. They might have been worth caching. They might have been used for a while then evicted when they expired or a dependency changed.

Manually Instrument Use of Cache

In this approach, you modify the application's source code to instrument every use of cache to determine hit/miss rates, hit/miss counts, and the ratio for that use. For example:

```
DataView Source;
Source = (DataView)
Cache["MyDataKey"]; // Cache lookup.
if(Source != null )
   // Add instrumentation
   // to count cache hits
            XXXXXXXXX();
else {
            // Add instrumentation to
  // count cache misses
            YYYYYYYY();
 // Get the data
            Source = fetchMyData("MyDataKey");
// Put the data in cache
            Cache["MyDataKey"] = Source;
}
// use the data
```

This is very accurate: it simulates processor cache counters to show where cache is effective and where it is not. Unfortunately, extensive code changes are required to implement this instrumentation, and they can be more complicated in certain code situations.

First, caching has to be implemented on every likely candidate. Implementing caching is difficult because although it is only a couple lines of code, you must correctly identify dependencies.

Second, since C# does not provide macros like *__FILE__* and *__LINE__* or the ability to build macros which would combine these built-in counters with instrumentation, implementing the cache manually is labor intensive.

Using ASP.NET Output Caching

Output caching represents another way of reusing the previously generated data instead of performing the processing-intensive operation again.

Output caching is used when the whole page, or part of it, can remain static for some period of time. Obviously, the biggest benefits can be reaped if the whole page is stored in cache for future use, but this may not always be the case. In this case, the page can be restructured to make the cacheable part of it a separate user-defined server-side control. Then caching can be applied to the output from this control. The easiest way to enable output caching for both user control and the whole page is to put the OutputCache directive at the top of the ASPX page containing its definition:

```
<%@ OutputCache Duration="#ofseconds"
    VaryByParam="parametername" %>
```

This approach is the best for the pages and controls that can always be cached for a specified period of time. You can also use the VayByParam attribute to specify caching by parameter, which caches the output for every unique value of the input parameter. If the decision of whether to cache the output depends on more complex criteria, the framework provides a programmatic API to dynamically enable the output caching:

```
Response.Cache.SetCacheability(HttpCacheability.Public);

Response.Cache.SetExpires(DateTime.Now.AddSeconds(30));
```

Based on the output from our workload, we experienced a 23% performance degradation in the case where output caching is disabled. Based on our experience, significant performance gains can be attained if you examine your application to see which output pages do not need to always show the current snapshot of your data, so that output caching can be applied.

Better yet, the new IIS 6 architecture allows for an output of dynamic ASP.NET pages to go into kernel mode cache. This increases the benefit from caching, since the processing of requests does not need to transition from kernel to user mode, a transition that involves a context switch and is, therefore, much more expensive.

Use of output caching can also help in scale-out scenarios, where all the cache requests can be serviced by a separate system running the Microsoft ISA Server*.

How to Choose Which Type of Caching to Use

When choosing a type of caching you need to examine your application and determine whether the data you can cache are used by a single page or by a number of pages. If only one page (or control) is affected, it is likely that output caching will work better. The reason for this is that in the case of output caching, you eliminate both data retrieval and presentation logic, so the amount of work to satisfy the request is minimized. In the case of object caching, only the data retrieval part is eliminated, but the application still needs to execute the programming logic that produces the resulting HTML document that is based on the data retrieved. One potential problem with the output caching API, as compared to object caching, is that it currently lacks the ability to dynamically invalidate cached pages that are based on criteria other than expiration time.

The object cache approach works better when multiple pages are using the data object cached, or the source of the data is a file on the application server.

Session State Usage Tuning

Session state is a service provided to your Web application by default; the assumption is that all pages in your application require session state data to process incoming requests. If any of your pages do not rely on session state, you can get a limited performance benefit by disabling it for a given page.

To disable session state for a page, set the EnableSessionState attribute in the @Page directive to false as follows:

```
<%@ Page EnableSessionState="false" %>
```

Server-Side Control Guidelines

The general advice is not to use any server-side controls unless you have specific reasons to do so. Some of the reasons to use server-side controls over plain HTML might be to programmatically modify/access control properties, or to implement the same functionality on multiple pages, or use fragment caching on them.

If you don't have these reasons to use server-side controls, use plain HTML tags (located under the HTML tab in the WebForm Designer Toolbox) as a better performing alternative, since HTML tags do not require additional objects to be instantiated and manipulated during page processing on the Web server.

Software Configuration Issues

There are a number of ASP.NET settings that reside in the machine.config file provided with the .NET framework that can be tweaked to improve system performance.

The following is an example of some of the important components of the machine.config file: httpRuntime, sessionState, and processModel.

```
<httpRuntime
        executionTimeout=""
        minFreeThreads=""
        minLocalRequestFreeThreads=""
        appRequestQueueLimit= ""
    />

<sessionState
        stateNetworkTimeout=""
        timeout=""/ >

<processModel
        requestLimit=""
        requestQueueLimit=""
```

* Other brands and names are the property of their respective owners.

```
    memoryLimit=""
    maxWorkerThreads=""
    maxIoThreads=""
/>
```

The two parameters that will likely need to be tuned on an ASP.NET server are the request queue limit and the maximum number of worker threads.

Request Queue Limit

When system resources such as the CPU, or available worker threads, become saturated, ASP.NET will direct the request to the ASP.NET request queue. The requests in the queue are then serviced in the order in which they arrived (FIFO) as resources become available. The request queue is very effective in handling heavy or non-steady-state loads, but a limit must be placed on this queue, because throughput and response times will degrade as the queue grows larger.

The request queue limit can be found at \System\Web\httpRuntime\@appRequestQueueLimit in the machine.config file. This parameter is the maximum number of requests that will be queued before 503 "Server too busy" HTML errors are returned. For debugging, performance tuning, or any situation where error pages should not be returned, assigning a high value to the appRequestQueueLimit will prevent the HTTP 503 errors. In our configuration, we are testing performance and cannot tolerate errors being returned, so we set the queue size to be large enough such that requests will never be rejected. However, in a production environment, where the server can become overloaded with requests, a balance must be made between performance and not rejecting requests.

Maximum Thread Count

Many Web pages rely on an external resource such as a database or Web service, and waiting on these resources will often halt the processing of that request. If more CPU bandwidth is available, it is advantageous to process another request while the other is halted. Thus it is important to have the maximum thread count set high enough such that the CPU is saturated fully. However, having an excessively high thread count can cause too many requests to be processed concurrently, which can degrade performance.

The maximum worker thread limit can be found at \System\Web\processModel\@maxWorkerThreads in the machine.config file. This parameter is the maximum number of worker threads that will process requests. It is beneficial to run with as few threads as possible. However, without sufficient worker threads, the CPU will not be fully utilized, so the optimum value must be experimentally determined.

Unfortunately, ASP.NET does not provide a performance counter to measure the number of worker threads in use. The only way to estimate this number is to use the "ASP.NET Applications\Pipeline instance count" counter. Pipeline in this context is an instance of your HttpApplication-derived class named Global, which resides in the Global.asax.cs file in your Web application directory. This class defines the logic of HTTP request handling inside ASP.NET, common to all types of request processing, whether HTML (ASPX pages) or XML based (Web Services). At any given moment, a pipeline instance can process only one request, so this counter indicates how many requests can be processed concurrently, which has some correlation to the number of worker threads.

One example of potentially needing a higher thread count is when the ASP.NET server has long-standing requests out to a database (or other external resource). In the early stages of our workload development, the database implementation was slow and inefficient and response times were poor. Because of this, more threads were needed on the ASP.NET server so that it could continue processing requests while waiting for responses from the database.

However, in our current, more optimized workload, after varying max threads with 1, 5, 10 and 15, the results revealed that 10 threads (per processor) provided the maximum throughput for this application. This setting was found to be quite different if the SQL server did not use stored procedures. It is therefore likely that you will need to run a series of experiments to determine the optimal configuration. This procedure may need to be repeated if the design of your application changes.

ADO .NET Related Optimizations

ADO .NET provides a number of alternatives for accessing and modifying the data residing in the SQL Server database. The DataSet API provides a unified way to manipulate the data in the offline mode and synchronize it with its source. The DataSet API supports different types of data sources: an SQL Server database, an XML file, or some custom data provider for a proprietary format.

A different set of APIs supports fast data retrieval of ResultSet objects and execution of either dynamically generated SQL stat elements or stored procedures. The main difference between the last two methods is the fact that the SQL Server needs to parse and recompile the SQL statement submitted in the case of dynamic SQL statements, whereas stored procedure execution uses a precompiled version of the SQL statement, so these two steps are no longer required. We have also discovered that the previous assumption about the high overhead of

executing the stored procedure compared to a dynamic SQL statement is not valid in the case of the Microsoft SQL Server. This is because in the case of dynamic SQL statements, the .NET client implicitly invokes a stored procedure called sp_executesql.

To show the benefits of the stored procedure approach over dynamic SQL generation, we ran a series of tests on the current workload implementation. In the run where we replaced the stored procedure implementation with the dynamically generated SQL alternative, the throughput of the system as a whole dropped by over 18%. Moreover, in order to compensate for the worse response times of the database (DB) server, we needed to increase the number of ASP.NET worker threads to 20 from the original setting of 10.

WORKLOAD EVOLUTION

The current version of the workload is geared towards improving the performance of ASP.NET* Web applications by exercising most of the common features of the new engine. However, the current computing trends point toward more scalable scenarios, where the Web server integrates the data provided by the remote application services. The .NET platform provides a number of choices for implementing distributed computing.

Distributed Computing Scenarios

As mentioned earlier, our workload has a built-in flexibility that allows it to dynamically load different implementations of stateless business service instances. While the current version of the workload is configured to create instances locally to study the behavior of a single Web server, we currently have two more versions of the business service classes: one implemented as a Web Service and another one implemented as a Remoting server application. In the case of the latter, only remote object proxies are instantiated locally, not the actual implementations itself. The only thing that is required to enable this change is to change the application configuration file to specify a different class to be loaded. This process is totally transparent to the presentation layer: no change to the source code is required.

Web Services

Web Services are the new emerging standard in the distributed computing arena. Their most appealing feature is that they utilize existing standards, such as HTTP and XML, which are the most widely deployed and used. The downside of this protocol is the relatively high overhead that is associated with it: it is based on XML and rides on top of HTTP. However, Web Services are clearly the best choice for heterogeneous computing environments where ease of integration and support are the key factors.

.NET provides the highest level of support for the Web Services standard, so the integration of components running on remote machines is almost transparent.

.NET Remoting API

Remoting API is an alternative to Web Services, which provides an extensible framework that allows you to easily configure communication channels by using different protocols and encoding standards. This set of protocols allows for binary encoding and use of the TCP protocol instead of HTTP.

Our preliminary results indicate that this option provides significant performance benefits over Web Services, and that it is also better integrated with the .NET framework. The downside, however, is that it can only be used when both client and server applications are CLR-based, which means that it is not the right solution if cross-platform compatibility is required.

CONCLUSION

ASP.NET* is a great new development platform for Web applications. It provides a rich feature set and automates a number of common tasks. However, because it hides the complexity of handling Web requests from the developer, it is easy to incorrectly estimate the performance impact of different implementation alternatives and system-level configuration options.

The object and output cache are powerful capabilities provided by .NET and using them properly is a key to high performance and scalable applications. As the object cache capability is fairly new, tools are not yet available to automate the process, but a skilled programmer can use these techniques and get great results. Output caching brings substantial performance benefits while requiring minimal coding efforts.

By applying systematic methodology and appropriate tools it is possible to identify and alleviate the performance bottlenecks in the system. Also, it is extremely important to tune the application software stack to achieve optimal performance. This is a complicated task that requires some experimentation in

* Other brands and names are the property of their respective owners.

* Other brands and names are the property of their respective owners.

order to find optimal values for your specific application. When the proper tuning has been applied, ASP.NET applications can scale well on Intel-based servers.

Web Services and .NET Remoting technologies enable the next generation of Web applications that consist of a number of distributed services, seamlessly integrated, using these protocols. .NET Remoting offers higher performance than Web Services, but it is not platform independent; it requires that both the client and server applications are CLR-based.

Ongoing research occurs at Intel to ensure that the current and emerging technologies such as ASP.NET, Web Services, and .NET Remoting perform well on Intel Architecture. We hope that this article has provided valuable information to assist in developing and optimizing ASP.NET applications on Intel-based servers.

ACKNOWLEDGMENTS

Sam Warner provided a lot of advice and hands-on experience with tuning the HTTP and Microsoft's Internet Information Server (IIS) layers, and he also helped with identifying performance issues.

Paul Delvecchio was a great source of information on hardware setup and configuration as well as system-level tuning.

REFERENCES

[1] F. Yeon, "ASP.NET Performance Tips and Best Practices," http://gotdotnet.com/team/asp/ASP.NET%20Performance%20Tips%20and%20Tricks.aspx, October 22, 2001.

[2] J. Richter, "Applied Microsoft .NET Framework Programming," Microsoft Press, January 2002, ISBN: 0735614229.

[3] Microsoft ASP.NET, http://asp.net/

[4] Microsoft Patterns and Practices, http://msdn.microsoft.com/practices/

[5] "Application Architecture for .NET: Designing Applications and Services," http://msdn.microsoft.com/library/?url=/library/en-us/dnbda/html/distapp.asp, December 2002.

[6] "Developing High-Performance ASP.NET Applications," http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpcondevelopinghigh-performanceaspnetapplications.asp, December 2002.

AUTHORS' BIOGRAPHIES

George Vorobiov is a senior software engineer in the Software and Solutions Group in Bellevue, Washington. He is working on performance optimization of managed runtime technologies in the Web server application space. George holds an M.S. degree in Computer Systems from Kursk State Technical University, Russia; His e-mail is George.Voroibov@intel.com

Carl Dichter has been a systems and software engineer for over twenty years and has been at Intel since 1995. He has developed methodologies for optimizing server applications, especially managed code (C# and other .NET languages, as well as Java) and currently works with the processor architects to make sure our processors run today's applications best. Carl has filed 11 patents and written over 60 articles and one book (on software engineering and related subjects). His e-mail is cdichter@yahoo.com

John Benninghoff is a lead software engineer working on performance analysis of enterprise workloads on the .NET CLR Framework and ASP.NET. He has been at Intel for three years and has worked in the software industry for 20 years. Prior to Intel he worked at a major network software company doing performance analysis on Web and LDAP servers for their Internet portal, serving millions of registered users. Prior to that he worked for a major computer system vendor on graphics and network software. His e-mail is john.benninghoff@intel.com

Charlie Hewett is a software engineer in the Software and Solutions Group in Bellevue, Washington. He works on performance analysis and optimization of managed runtime technologies such as Microsoft's Common Language Runtime (CLR) and .NET Framework for Intel Architecture. Charlie holds a B.S. degree in Electrical Engineering from the University of Washington. His e-mail is charlie.j.hewett@intel.com

Runtime Environment Security Models

Selim Aissi, Intel R&D, Intel Corporation

Index words: security models, runtime security, access control, sandbox, CLR security, ASP.NET security, JRE security, Java security, runtime access control models

ABSTRACT

The tremendous new potential offered by distributed computing, inside and outside the home and business, also carries with it the necessity to exercise certain security safeguards. As distributed, mobile, and executable content moves among devices, the opportunity for security breaches increases dramatically. Also, as device-to-device e-Commerce services become more automated [11], new types of security threats are emerging. With these drastic changes in computing models comes a greater need for robust application security.

For example, "executable content" is the idea of sending code to a remote compute engine to be executed. In addition to flexibility and expressiveness, executable content brings new potential problems. A program received from a remote source must be regarded as non-trusted to some degree, and its access to certain resources must be restricted. However, this new execution model is not bound by the limitations of the operating system because the runtime environment enforces the security policies based on the code's origin. Both the Java* Runtime Environment (JRE) and .NET* Framework Common Language Runtime (CLR) security models have the following common security features: language type-safety, bytecode verification, runtime type checking, name space separation via class loading, and fine-grained access control.

This paper compares the JRE and the CLR evolutionary security mechanisms. The paper also compares the two models to the Clark-Wilson security model, a formal, application-level model used to ensure the integrity of commercial data. The Clark-Wilson model is a formal presentation of the security policy enforced by a system, and it is useful for testing a policy for completeness and consistency. It also helps describe what specific mechanisms are necessary to implement a security policy.

Besides exploring the nature and scope of the sandbox-based JRE and CLR security models and comparing them to the Clark-Wilson integrity model, this paper also provides some insight into the future of runtime security.

INTRODUCTION

The idea of using a sandbox to secure the threads running inside of that box is very similar to the idea of building a wall around a town to protect its inhabitants.

The concept of building a thick wall for protection is as old as history itself. Greek legend provides an interesting case of a thick wall that caused more destruction than protection: the Trojan Wall. Let's explore that security legend a bit further. During the Trojan War, the Greeks asked Epeius, an excellent craftsman, to build a wooden horse, which he did with the aid of the goddess Athene. Inside the horse were placed a handpicked group of warriors. Then the Greek fleet sailed away, leaving behind a warrior named Sinon, who pretended he had been left behind by accident. He also pretended that the huge wooden horse was an offering to Athene and that, if taken into the city, would make the city invincible. Despite warnings from some quarters, the Trojans pulled down part of their battlements and hauled the wooden horse inside the city. Lulled into a false sense of security, little watch was kept. The Greek fleet returned furtively and Sinon released the warriors from inside the wooden horse. Troy fell because the Trojans' confidence in an impenetrable wall led them to overlook the security risk in their midst [1]. When programmers (or users) fail to check inside the Horse (a metaphor for malicious code) before they roll it within the computing device, like the Greek legend, the result is unpleasant.

A Trojan Horse is an easily written security hack that has been used for years to breach traditional computer and network security barriers. The first Trojan Horses were disguised as demos, freeware, and shareware. The

* Other brands and names are the property of their respective owners.

unsuspecting victim would run the software from inside traditional security walls where the program could effectively attack. Sometimes the program just displayed a witty joke, performed some harmless mischief, installed viruses or broke into password files. Sometimes the Trojan Horse used security holes to break deeper into the computer. In all of those cases, the Trojan Horse causes some damage, including loss of productivity and confidence in the security of our systems.

Contrary to early claims, Trojan Horses and viruses are no strangers to runtime environments either [6]. In August of 1998, a proof-of-concept virus called Strange Brew appeared. While it did not carry a damaging payload, it did prove the concept that cross-platform Java* viruses and Trojan Horses could be written. Strange Brew, however, affects only Java applications, not Java applets that typically run inside a Web browser.

In January of 1999, the second known Java virus, called Java.BeanHive, was discovered. This virus was designed to infect both Java applets as well as Java applications.

The Java.BeanHive virus was, however, the first to exploit JRE's access control mechanisms by asking the user to grant the virus permission for full file access. Because the virus was a seemingly innocuous Java applet, some users inadvertently granted it full permission, not knowing it was malicious code.

In Java and .NET, the runtime environments provide security models that deal with access control to system resources. The following sections describe the capabilities offered by those mechanisms.

The .NET Framework also has had its share of security holes. In June 2002, session highjacking, information-leakage, and buffer overflow vulnerabilities were identified [12].

MOBILE CODE SECURITY: JAVA* AND THE .NET* ENVIRONMENTS

"Mobile code" denotes program code that traverses a network and executes at a remote site. The process of traversing can either be active as in the case of mobile agents which move around in a network at their own volition, or it can be passive, as in the case of user-downloaded code such as applets.

Both Java* and .NET* environments can be used as platforms for both types of code mobility, and in conjunction with the Internet, they open new possibilities for software development, software deployment, and computing architectures. The downside is that they also open new security threats. Downloaded code can include a virus or be a Trojan Horse and thus pervert the concept of code mobility over the Internet in a possibly dangerous way. Any mobile code platform, including both Java and .NET, suffers from four basic categories of potential security threats [5]:

- *Leakage.* This occurs when there are unauthorized attempts to obtain information belonging to or intended for someone else.
- *Tampering.* Tampering is unauthorized changing or deleting of information.
- *Resource stealing.* This occurs when there is unauthorized use of resources or facilities such as memory or disk space.
- *Antagonism.* These are interactions that don't result in a gain for the intruder but are, nonetheless, annoying for the attacked party.

To deal with these threats, Java and. NET environments provide special runtimes that try to protect users from erroneous or malicious mobile code and try to ensure the security and privacy of the user's system.

They both provide fairly good levels of protection against leakage and tampering but resource stealing and antagonism cannot be fully prevented since it is still hard to automatically distinguish between legitimate and malicious actions.

THE EVOLUTION OF THE JAVA* RUNTIME ENVIRONMENT'S SECURITY MODEL

In runtime environments, the security model is based on policy construction and enforcement. A security policy consists of the rules that must be obeyed by a program, the mechanisms to enforce these rules and to detect when they are violated, and the actions that are taken when a security violation is detected.

In Java*, a security policy is implemented by writing a subclass of the *SecurityManager* class and installing it as the system's security manager.

While the bottom three layers of Java's security model are fixed and defined by the Java language specification [2], the Java Virtual Machine (JVM) specification [3], and the Java API specification [4], the runtime environment is implementation-dependent. Although it is the only configurable part of the security model, this is,

* Other brands and names are the property of their respective owners.

nevertheless, sufficient for a wide range of different security policies to be implemented.

The Java Sandbox Model

Java security has undergone considerable evolution. In the JDK 1.0 security model, any code run locally had full access to system resources while dynamically loaded code had access to system resources controlled by a security manager. The default security manager sandbox provided minimal access to resources such as disk drives. In order to support a different security model, a new security manager would have to be implemented. The concept of trusted, dynamically loaded code was introduced in JDK 1.1. Any dynamically loaded code that was digitally signed by a trusted code provider could execute with the same permission as local code. JDK 1.2 introduced an extensible access control model that applies to both local code and dynamically loaded code. Fine-grained access to system resources can be specified in a policy file on the basis of the source of the code, the code provider, and the user of the code. Unlike earlier versions of the JDK, this policy file allows the security model to be adjusted without writing a new security manager. The security manager has standard access control checkpoints embedded in its code whose behavior is determined by the selection of permissions enabled in the policy file. New permissions can be defined, but explicit checks must be added to the security manager or application code if the permissions apply to application resources rather than system resources.

JVM Security

Four practical techniques for securing mobile code exist: the sandbox model, code signing, firewalls, and proof-carrying code. In order to secure mobile code, Java uses a hybrid approach, which combines sandboxes and code signatures. The Java core classes act as a security shield and enforce the sandbox model by granting or forbidding access to resources, based on a security policy. The rules specified in the security policy define the actions a piece of code is allowed to perform depending on the origin of the code and an optional signature. Not all of Java's powerful security mechanisms are in place by default when launching the JVM. While some basic checks are performed automatically, the more sophisticated concepts, including the sandbox model, have to be put into action explicitly.

Before a class is loaded, the following steps occur. First, the Verifier performs a set of security checks to guarantee properties such as the correct class file format, the correct parameter types, and binary compatibility. Doing these checks before loading enhances both security and runtime performance. They ensure the integrity of the Java runtime environment since no malformed class can be loaded that could cause a general system fault. Having passed the Verifier, the class loader loads the bytecode representation of the class and checks optional signatures. Furthermore, the source (i.e., origin) of the class's code is constructed, which consists of the location from which the class was obtained and a set of certificates representing the signature.

The source of the class code is the key input for the security policy construction for a given class. In Java 2, the security policy is defined in terms of protection domains, which define what a piece of code with a given source is allowed to do. Hence, a protection domain contains a code source with a set of associated permissions. Given the code source of a class, the security policy is searched to determine the permissions of the class.

Finally, the class is “defined,” meaning it is made publicly available and added to the class loader’s cache of classes. This is important to ensure class uniqueness. Java considers two classes equal if, and only if, they have the same name and were loaded by the same class loader.

After these initial steps, the class can be used in the Java runtime environment. However, every time the class tries to access a system resource, its permissions are checked by the security manager. If the call to the security manager returns silently, the requesting caller has sufficient permissions to access the resource, and the execution continues. If not, a security exception is raised and has to be handled by the caller or otherwise the JVM terminates.

A key question is how the security manager decides whether access to a resource is granted. Since Java 2, the security manager is mainly included for compatibility reasons and delegates nearly all of its tasks to the access controller. The access controller uses a stack inspection algorithm and the security policy to decide how to proceed.

The stack inspection algorithm is based on the call stack of the current method. Since every class is assigned an appropriate set of permissions when it is loaded, the stack inspection algorithm can use this information to make its decision.

THE .NET* FRAMEWORK COMMON LANGUAGE RUNTIME SECURITY MODEL

The Microsoft .NET* Framework offers code access security and role-based security to help address security concerns about mobile code, and to help determine what users are authorized to do. Both code access security and role-based security are implemented using a common infrastructure supplied by the Common Language Runtime (CLR).

Because they use the same model and infrastructure, code access security and role-based security share several underlying concepts described in the following sections.

The CLR Code Access Security Model

Any application that targets the CLR must interact with the runtime's security system. When an application executes, it is automatically evaluated and given a set of permissions by the runtime. Depending on the permissions that the application receives, it can either run properly or it will generate a security exception. The local security settings on a particular computer ultimately decide which permissions the code receives. Because these settings can change from computer to computer, one can never be sure that code will receive sufficient permissions to run. This is in contrast to the world of unmanaged development, in which one may not have to worry about the code's permission to run. CLR code access security is based on the following four concepts: writing type-safe code, using imperative and declarative syntax, requesting permissions for the code, and using secure class libraries.

In order to write type-safe code and to enable code to benefit from code access security, a compiler that generates verifiably type-safe code must be used.

Interaction with the runtime security system is performed using imperative and declarative security calls. Declarative calls are performed using attributes; imperative calls are performed using new instances of classes within your code. Some calls can only be performed imperatively, while others can be performed only declaratively. Some calls can be performed in either manner.

Requests for permissions in the code are applied to the assembly scope, where the code informs the runtime about permissions that it either needs to run, or specifically does not want. Security requests are evaluated by the runtime when the code is loaded into memory. The purpose of requests is only to inform the runtime about the permissions it requires in order to run. Requests do not influence the runtime to give the code more permissions than it "deserves."

The secure class libraries use code access security to specify the permissions they require in order to be accessed. The developer must be aware of the permissions required to access any library that the code uses and make appropriate requests in the code [7].

The CLR Role-Based Access Model

Roles are often used in an application to enforce some policies. Role-based security can be used when an application requires multiple approvals to complete an action.

The .NET Framework's role-based security supports authorization by making information about the principal, which is constructed from an associated identity, available to the current thread.

A principal represents the identity and role of a user and acts on the user's behalf. Role-based security in the .NET Framework supports three kinds of principals: Generic Principals which represent users and roles that exist independent of Windows NT* and Windows 2000* users and roles, Windows Principals which represent Windows* users and their roles (or groups), and Custom Principals which can be defined by an application in any way that is needed for that particular application.

The identity, as well as the principal it helps to define, can be either based on a Windows account or be a custom identity unrelated to a Windows account. The .NET Framework applications can make authorization decisions based on the principal's identity or role membership, or both. A role is a named set of principals that have the same privileges with respect to security. A principal can be a member of one or more roles. Hence, applications can use role membership to determine whether a principal is authorized to perform a requested action.

To provide ease of use and consistency with code access security, .NET Framework role-based security provides *PrincipalPermission* objects that enable the common language runtime to perform authorization in a way that is similar to code access security checks. The *PrincipalPermission* class represents the identity or role that the principal must match and is compatible with both declarative and imperative security checks. You can also access a principal's identity information directly and

* Other brands and names are the property of their respective owners.

* Other brands and names are the property of their respective owners.

perform role and identity checks in your code when needed [7].

Comparison Between the JRE and CLR Security Models

Figure 1 compares the Java* and .NET Framework architectures. The JRE and CLR are viewed as middle layers between intermediate languages and the underlying operating systems.

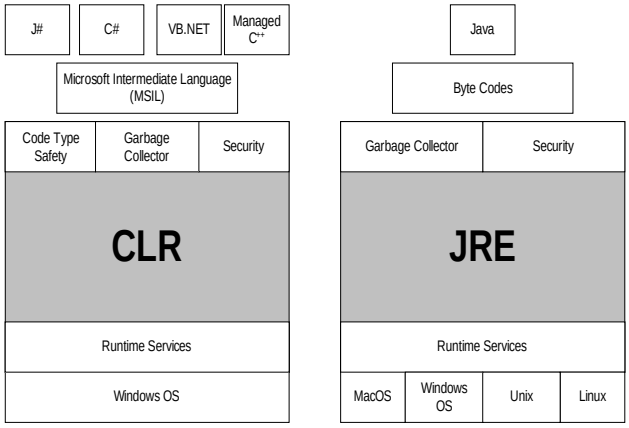


Figure 1: CLR versus JRE

In order to compare the two security approaches, the Clark-Wilson Security Model is used.

The Clark-Wilson Security Model

Integrity models [8] are used to describe what needs to be done to enforce information integrity policies. There are three goals of integrity: to prevent unauthorized modifications, to maintain internal and external consistency, and to prevent authorized but improper modifications.

To accomplish these goals, a collection of security services that embodies the properties needed for integrity as well as a framework for composing them is needed. The needed security properties for integrity include access control, auditing, and accountability.

The Clark-Wilson [9] model is an integrity, application-level model that attempts to ensure the integrity properties of commercial data, and it provides a framework for evaluating security in commercial application systems. It was published in 1987 and updated in 1989 by David D. Clark and David R. Wilson [13].

The Clark-Wilson model is based on analyses of security models actually applied within businesses. These security models aim at ensuring the integrity of resources rather than simply controlling access to them. They depend on controlling state transformations, and upon maintaining separation of duties between users of the system.

Clark and Wilson partitioned all data in a system into two types of data items for which integrity must be ensured: Constrained Data Items (CDIs) and Unconstrained Data Items (UDIs). The CDIs are objects that the integrity model is applied to, and the UDIs are objects that are not covered by the integrity policy (e.g., information typed by the user on the keyboard). Two procedures are then applied to these data items for protection. The first procedure, namely the Integrity Verification Procedure (IVP), verifies that the data items are in a valid state (i.e., they are what the users or owners believe them to be because they have not been changed). The second procedure is the Transformation Procedure (TP), which changes the data items from one valid state to another. If only a transformation procedure is able to change data items, the integrity of the data is maintained. Integrity enforcement systems usually require that all transformation procedures be logged, to provide an audit trail of data item changes.

In runtime environments, CDIs and UDIs can be mapped to fields of components (e.g., assemblies). TPs can be mapped to Java methods or .NET assemblies. An assembly is a collection of types and resources that is built to work together and form a logical unit of functionality.

A principal in this context is an authenticated Java or .NET principal where the authentication has been achieved using either the Java Cryptographic Extension (JCE) or Microsoft's Cryptographic API (CAPI*).

DISCUSSION

Resource Integrity

The basic concepts of access control in the JRE and CLR security models do not meet Clark-Wilson's requirement of resource integrity. Both environments require each controlled operation to be re-coded to include a permission check. This is not appropriate for a component that is delivered in binary form. Both the CLR and JRE also require determination of which operations update the state of an object so that only those operations that maintain the integrity of the system are allowed. This approach is error prone. A better approach would be to intercept all state accesses and allow only those made from operations that maintain integrity while blocking all others.

Execution-Time Checking

The .NET* Framework holds an advantage in the area of execution-time checking. .NET's application domains are

* Other brands and names are the property of their respective owners.

less permeable than Java's, i.e., NET code verification is stronger by default for local applications. Because a JVM verifies only remotely loaded code by default, one can run a Java* program locally without any security manager at all [10].

Data Protection

Neither environment offers a significant advantage in source code and data protection. Each platform has its strengths and weaknesses in this area. .NET's cryptography relies on the developer properly configuring the CAPI because it's so closely tied to Windows*. However, the variety of plug-ins and components available for Java makes it a more flexible environment.

The Java Cryptographic Extension (JCE) is a mechanism that allows suppliers of cryptography to integrate their libraries in a standard way with Java applications. The API is fairly flexible, allowing detailed control of the cryptographic process. However, that flexibility can lead to excessive complexity and can make the API difficult to use.

Communication Security

Developers using the .NET Framework may need to use Microsoft's Internet Information Server (IIS*) for communication protection. This strong dependency on a Web Server, such as IIS, to provide runtime security services, could restrict CLR's communication security.

Code-Access Security

The code-based security mechanisms for Java and .NET are very similar. The Windows connection gives the new platform a richer set of permissions and evidences than Java does. Java is more stripped down due to its platform independence; however, Java's code-based access control is very mature and offers several configurable policy levels. .NET provides hierarchical code groups and allows for targeted code checks.

User Authentication

The .NET Framework offers good authentication out of the box. It implements authentication through authentication "providers," such as forms, Passport, and IIS. .NET's close ties to IIS can hinder its flexibility.

Java code is easier to modify and the Java Authentication and Authorization Service (JAAS) is available for developers to modify and then plug in. JAAS also provides several levels of customization, making Java's authentication and role-based access control stronger than .NET's.

Both platforms, however, lack a mechanism for advanced user-based access control, such as permission delegation. For more complex role-based access control projects, users have to build their own layer of security for determining user-based access control.

Auditing and Tracking

Neither platform offers much support for secure authentication and tracking. Although JDK 1.4 introduces a logging package, it offers no secure facilities. .NET offers a managed wrapper around Windows *EventLog*, but that's as far as its auditing features go. Consequently, .NET applications are restricted to the functionality and limitations of *EventLog*.

Neither .NET nor Java provides acceptable support for auditing and tracking transactions. With .NET, developers can use the Windows mechanisms, but they need to go outside the .NET Framework to get them. Even when .NET and Java add logging packages, it is not clear how secure that mechanism would be.

Managed and Unmanaged Code

While the .NET Framework provides a solid security model through managed code in the CLR, the ability to run unmanaged code confers the ability to bypass CLR security through direct calls to the underlying operating system APIs. Also, in Java, signed and trusted code has unrestricted access to system resources. Java's calls to native code through the Java Native Interfaces (JNIs) confer the ability to bypass JRE's security. Similarly, running unmanaged code in the CLR can be used to bypass the .NET Framework security.

FUTURE OF RUNTIME SECURITY

The real test for runtime security facilities will be their deployment in large distributed systems. In their current models, the overall system security depends on perfect functioning of the application, the language, the virtual machines, and the underlying operating systems. It also depends on the interaction of those elements. Therefore, this kind of system security becomes very complicated and unstable if the system is very large. The experience of Java security has shown that most of the security problems reported come from defects in the implementation of the security mechanism and from malicious applets that use vulnerabilities in the applications that use the virtual machine.

The ability to encrypt communication and provide digital signatures is only part of enabling secure applications, trusted communication, and proof-of-identity. There is

* Other brands and names are the property of their respective owners.

still the issue of where and how the keys are generated and stored. Not only do the keys have to be exchanged over secure links, they have to be generated and then managed in a secure way. Hardware-based secure storage can play a major role in securing the generation and safekeeping of keys.

Security hardware may also provide more reliable random-number-generation, time-stamping, and auditing capabilities, which are crucial for cryptographic and signature functions.

Both Java* and the .NET* Framework include cryptographic capabilities based on software libraries. However, performing the encryption on hardware is inherently more secure than leaving it to the software. Furthermore, hardware-based cryptography, signatures, and key storage capabilities can provide a common security infrastructure for both Java and .NET, which can make the development of secure runtime applications much easier than having to develop code to invoke the Java or .NET cryptographic extensions.

CONCLUSION

Both Java's* JRE and .NET's* CLR do not meet Clark-Wilson's requirements of resource integrity. However, they both provide quite comprehensive security services, though each has a different focus.

Java's authentication and authorization services are fairly flexible. Although its use is not mandated, authentication and authorization functionality can be provided by the JAAS. .NET's authentication and authorization services, however, are provided through the Windows* operating system or identification stores (e.g., Passport*).

Both environments use similar concepts for handling user and code access to resources, with permissions being critical to both. The concept of roles is used to associate permissions with principals in both environments.

Common hardware-based cryptographic and key-management capabilities can drastically enhance the security of the runtime environment. However, getting the industry to agree on a common hardware architecture for mobile and non-mobile security will be a challenge for the next few years.

ACKNOWLEDGMENTS

The author extends his thanks to the reviewers who provided invaluable feedback. Special thanks to Gene Forte, Srinivasan Krishnamurthy, Joel Munter, Gururaj Nagendra, Murthi Nanja, and Carlos Rozas for their careful review of this paper. My gratitude is also extended to Norbert Mikula for generating many of the thoughts in the paper.

REFERENCES

[1] Seton-Williams, M.V., *Greek Legends and Stories*,. Barnes & Noble, Inc., New York, New York, pp. 103-111.

[2] Joy, B., Steele, G., Gosling, J., and Brasha, G., *Java Language Specification*,. Book News, Inc., Portland, Oregon.

[3] Lindholm, T. and Yellin, F., *The Java Virtual Machine Specification*, Addison-Wesley, New York, New York.

[4] Gong, L., *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*,. Sun Microsystems Press, Santa Clara, California.

[5] Goulouris, G., Dollimore, J., and Kindberg, T,. *Distributed Systems – concepts and design*, International Computer Science Series, Addison-Wesley, Massachusetts and London, pp. 477-516.

[6] Schweitzer, D., *Securing the Network from Malicious Code: A Complete Guide to Defending Against Viruses, Worms, and Trojans*, John Wiley & Sons, New York, New York.

[7] LaMacchia, B.A., Lange, S., Lyons, M., Martin, R., and Price, K., *.NET Framework Security*, Addison-Wesley, Massachusetts and London, New York, New York, pp. 43-79.

[8] Summers, C. R., *Computer Security: Threats and Safeguards*, McGraw Hill, New York, page 142.

[9] Anderson, R., *Security Engineering: A Guide to Building Dependable Distribution Systems*, Wiley Computer Publishing, New York, pp. 188.

[10] Kunene, G., *Software Engineers Put .NET and Enterprise Java Security to the Test*,. http://archive.devx.com/enterprise/articles/dotnetvsjava/GK0202-1.asp.

[11] Aissi, S., Pallavi, M, and Krishnamurthy, S., "Ebusiness Process Modeling: the Next Big Step," *IEEE Computer*, May 2002.

[12] Adams, L., *ASP.NET security holes*, ZDNet, Australia, June 2002, http://www.zdnet.com.au/builder/architect/sdi/story/0,2000035062,20266124,00.htm

[13] Clark, D. D. and Wilson, D.R.,"A comparison of commercial and military computer security policies,"

* Other brands and names are the property of their respective owners.

IEEE Symposium on Security and Privacy, pp. 184-194, Oakland, CA, 1987.

AUTHOR'S BIOGRAPHY

Selim Aissi has been involved in the development of secure, safety-critical systems in the R&D sector, and in military, automotive, and wireless appliances for over twelve years. Before joining Intel in 1999, he worked at the University of Michigan, General Dynamics' M1A2 Abrams Battlefield Tank Division, General Motors' Embedded Controller Excellence Center, and Applied Dynamics International. At Intel, he played several management and senior architecture roles, and he is currently a Senior Security Architect at Intel's Research & Development group. Selim served on the review board of several publications and conferences. He currently serves on ACM's CCS'03, SAM'03, NCISSE'03, and IC'03 conference boards. He holds a Ph.D. degree in Aerospace Engineering from the University of Michigan and is a member of the IEEE, ACM, and ISSA. His e-mail is selim.aissi@intel.com.

Runtime Abstractions in the Wireless and Handheld Space

Lynn Comp, Wireless Computing and Communications Group, Intel Corporation
Tim Dobbing, Wireless Computing and Communications Group, Intel Corporation

Index words: Wireless Handheld Devices, Mobile Information Devices, Java, J2ME, .NET, Managed Runtime Environments

ABSTRACT

The wireless handheld industry faces a number of challenges that managed runtime environments are uniquely positioned to solve. Unlike the Information Technology (IT) and personal computing industries in recent history, the handheld and wireless markets have experienced significant fragmentation in the associated application development environments. Developer support is referred to in the tens and hundreds of thousands of individual developers, rather than the millions referred to in the IT and personal computing industries. Another challenge that managed runtime environments help solve is providing the ability to connect to back-end infrastructure, whether in the carrier network or the enterprise. This connectivity is critical to device functionality in wireless. While runtime abstractions are intended to simplify the lives of the developers by providing the added protection and capabilities of Java* and .NET*, the abstraction can also present a challenge for users desiring optimal device performance, and component suppliers wishing to quickly enable advanced functionality on their latest products. This paper examines the tradeoffs of implementing an abstracted runtime environment in the wireless and handheld spaces, focusing on two of the most common managed runtime architectures, Java and .NET.

A BRIEF HISTORY OF THE CELLULAR INDUSTRY

The cellular industry (traditional wireless) is a relatively new industry having evolved over just the last two to three decades. In that time, the technology that came out of AT&T research labs, was first used by the military, then moved to the business community, and finally made available to the consumer at large. An important point regarding the cellular telephone, in regards to its relatively rapid adoption when compared to other ground-breaking technologies, is that the user experience with a cellular phone is nearly identical to the user experience with a business or home phone: the cellular phone looks and dials just like a landline phone, and it has, by and large, been limited to voice communication. Adding functionality to a traditional cellphone is accomplished by replacing the cellphone altogether – there is no provision for adding software or applications to this type of device. To support dynamic software additions/updates, the cellphone needed to integrate voice communication services with the system management and data communications services of the traditional computing industry while maintaining the integrity of both.

The portable version of the data computing industry, handheld computing, is also relatively new; it attempts to mimic must-have computing functionality in small, more portable devices capable of achieving better battery life. Entering the market in 1993 [1] the Apple Newton* was one of the first mainstream handheld computers, or as they were called at the time, Personal Digital Assistants (PDAs). Apple's hope was that the loyal developer community of Apple enthusiasts would also support the handheld version. Unfortunately, the Newton was unwieldy in functionality, size, and cost; it was too large for a shirt pocket or a pocketbook, too expensive to be an impulse purchase and unable to perform a few key basic tasks well. Due to these factors the Newton achieved a faithful, yet small following of users.

The Palm Pilot*, introduced in 1996 [2] was an invention that benefited a great deal from observing the market introduction and acceptance of the Newton. The tools to program the device were very inexpensive; the synchronization with the desktop PC was reliable and fast; the device was small enough to be carried in pockets or pocketbooks; and the price of $300 was just slightly higher than an impulse purchase for the more technically

* Other brands and names are the property of their respective owners.

savvy consumer. The Palm Pilot ultimately attracted as many as 60,000 developers to its platform and unique API set. In the UK, a company called Psion made headway with a keyboard-based device with similar functionality to the Palm and Newton. This success resulted in a third set of unique APIs for a software vendor to write to, test, and support. By the time Microsoft entered the market in 1997 with the WindowsCE* platform, the independent software vendors had to write, test, and support no fewer than four different platforms to support the portable market at large. Given the low-cost nature of the handheld market in general, it was tough to support all the platforms and still have a positive return on development investment as an Independent Software Vendor (ISV). The highest volume market, the cellular telephone, allowed no extensibility once the platform was on the market, and the portable but extensible platforms were fragmented to the point where an ISV faced a potentially negative return on investment on his or her development efforts.

THE GREAT UNIFIER

When Sun Microsystems introduced Java* in 1995 with the tagline "write once, run anywhere," it was unclear exactly what Java was intended to do. Editors and analysts who studied and researched the computing and software industries wrestled with how to categorize Java: Was it intended to replace the operating system, the CPU instruction set, or create network computers[1], or was it meant to provide an alternative to Microsoft desktop operating systems? Whatever category that Java was placed in at that time, it was too large and unwieldy, too proprietary and too slow for small, memory-constrained devices (although that didn't stop early visionaries such as the Nortel Orbiter [3] phone development team from attempting to create an advanced cellular phone running Java). Java gained its early strength and acceptance in the enterprise infrastructure, simplifying and unifying multiple types of Information Technology (IT) and Internet systems through a higher layer of abstraction.

Java in Wireless

A number of developments spurred the acceptance of Java into wireless client devices ("client" refers to a device that must attach to a network or other infrastructure to accomplish specific mission-critical tasks). The first key development was Sun's introduction of a small-Java solution and virtual machine for memory-constrained devices in 1999 [4]. The inherent protection from memory overruns provided in the Java architecture itself already made Java attractive, since wireless operators are highly sensitive to handsets suspending operations without warning while a customer is operating the handset, or being capable of proliferating network-damaging viruses. Non-functional handsets increase operator costs, and a healthy on-line network represents the only revenue source for the wireless operator.

A second reason for the wireless operators having a favorable response to Java is the inherent memory efficiency of bytecodes versus native code [3, 6]. Operators have extremely high investment costs in preparing to provide wireless services. The first cost to an operator is paying national governments extremely high license fees for rights to a limited amount of wireless spectrum in a specific geographical area. The operators are limited to servicing wireless data traffic within that allocated spectrum. The more customers that an operator can support within the limited amount of spectrum purchased without customers experiencing dropped calls or being unable to connect to the network, the faster the operator is able to recoup cost and earn positive cash flow.

The final factor in Java's favor is that operators are familiar with consortium-based standardization processes through years of participation in the International Telecommunication Union (ITU) and are more comfortable with standardization than accepting whatever an external vendor develops in binary form, with no ability to influence or adapt it. The establishment of the Java Community Process further enabled the acceptance of Java by wireless operators. Not only was Java a relatively open standard, it attracted a great number of software developers in the enterprise space due to its ability to provide a cohesive layer over whatever system existed underneath it. A tenth of the three million developers supporting Java in 2002 represents quadruple the number of developers supporting the most popular handheld device of the 90s, the Palm Pilot*.

Java in the Wireless Client Device–Devices Now vs. Devices of the Future

Java technology for the wireless client device is in a state of transition. Currently, Java wireless client devices are based on the Java 2 Platform, Micro Edition (J2ME) and specifically on the Connected Limited Device Configuration (CLDC) version 1.0 and the Mobile Information Device Profile (MIDP) version 1.0. The specifications for CLDC 1.0 and MIDP 1.0, JSR-030 and JSR-037 respectively, were released in 2000 at a time

* Other brands and names are the property of their respective owners.

[1] Network computers are thin clients in the network architecture where the "network is the computer." In this model server computers provide the intelligence.

when memory capacity and processor power were limited, and as such, compromises were made in terms of the level of support for the Java Language Specification and the Java Virtual Machine Specification. In addition, new libraries (e.g., user interface, networking, and persistence libraries) were developed to support these resource-constrained devices, reducing the consistency between standard and wireless Java.

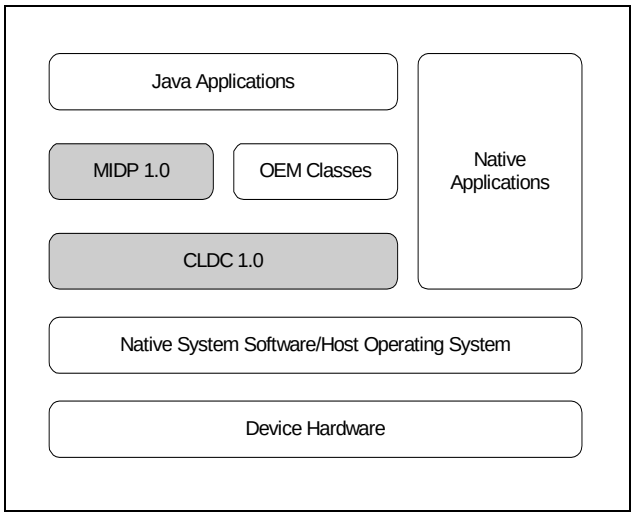


Figure 1: CLDC 1.0/MIDP 1.0 architecture

The high-level architecture of these CLDC 1.0/MIDP 1.0 devices is shown in Figure 1 above. One of the goals of the J2ME architecture was to provide a highly portable, secure, small footprint application development environment for resource-constrained connected devices [8], and it has been largely successful.

The CLDC 1.0 provides the platform that is intended to serve as the lowest common denominator for all such devices (e.g., mobile phones, pager, and point-of-sale terminals) while the MIDP 1.0 addresses the needs of a specific vertical market (e.g., mobile phones).

CLDC 1.0 addresses the following areas: support for the Java language and virtual machine features, core libraries (e.g., java.lang, java.io, java.util), input/output, networking, security, and internationalization. MIDP 1.0 addresses the following areas: application models, user interfaces, persistent storage, networking, and timers.

In the case of CLDC 1.0, a number of sacrifices were made in terms of the support for the Java Language Specification and the Java Virtual Machine Specification. Specifically, there is no floating-point support, there is no support for finalization, and there is limited exception handling support. In addition, there is no support for the Java Native Interface[2] (JNI), no support for user-defined class loaders, no reflection, no support for thread groups or daemon threads, and no support for weak references. The lack of support for reflection also means that there is no support for language or virtual machine features that rely on reflection (e.g., RMI, object serialization, JVM debugging, and profiling). From a programming perspective and also in terms of being able to port Java 2 Standard Edition (J2SE) applications to the J2ME platform, these are serious issues that have led to fragmentation of the J2ME architecture.

In terms of library support, the CLDC adopted subsets of most of the corresponding J2SE libraries. The exception to this was the specification of a new networking library, the Generic Connection Framework.

While CLDC 1.0 addressed the needs of the horizontal market, MIDP 1.0 addressed the needs of a specific vertical market (e.g., mobile phones). Specifically it covered the following areas: user interface support, networking support (i.e., HTTP), persistent storage, application models, and timers. In doing so, MIDP 1.0 diverged significantly from the J2SE. New user interface and persistent storage libraries were specified, and the networking libraries were based on extensions of the CLDC 1.0 Generic Connection Framework.

In terms of achieving the goals that the J2ME expert groups set for themselves in specifying the J2ME architecture, the results have been mixed. Certainly the goal of a small footprint has been achieved. However, there has been a cost in terms of application portability and security. The approach consistently taken by the expert groups has been one of specifying the lowest common denominator in terms of functionality for both CLDC 1.0 and MIDP 1.0, which has led to a large degree of fragmentation of the architecture. Specifically, proprietary Original Equipment Manufacturer (OEM) classes have been used by individual phone and PDA OEMs to make up for the lack of functionality in the CLDC and MIDP specifications. A number of companies, including Sprint PCS, Motorola, and Nokia, have added their own unique device-specific functionality (e.g., sound, additional network protocols, additional user interface functionality) in the OEM classes, and in doing so sacrificed application portability for enhanced functionality.

Portability has not only been sacrificed between J2ME CLDC/MIDP1.0 devices but also between the J2ME (i.e.,

[2] Java methods are able to invoke native methods through proprietary means.

wireless) and J2SE* (i.e., desktop) clients. In addition to the inconsistencies between user interfaces and networking libraries, the CLDC/MIDP 1.0 client device does not align with the J2SE security model.

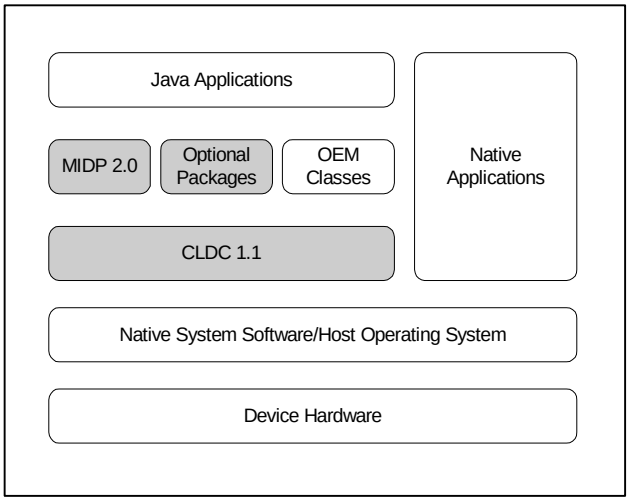


Figure 2: CLDC 1.1/MIDP 2.0 architecture

MIDP 2.0 and CLDC 1.1 (JSR-118 and JSR-139, respectively) address many of the concerns of MIDP 1.0 and CLDC 1.0. CLDC 1.1 has added back support for floating point as well as support for weak references. MIDP 2.0 provides significant improvements over MIDP 1.0. It provides enhanced networking support, enhanced user interface support, support for gaming, support for sound, and a security model that better aligns with the J2SE model of security.

The architecture model for a CLDC 1.1/MIDP 2.0 device is shown in Figure 2 above. Aside from the inclusion of CLDC 1.1 and MIDP 2.0, the major enhancement is the support for optional packages. Optional packages are intended to supplement the functionality provided by CLDC 1.1 and MIDP 2.0 and eliminate the need for the use of OEM classes. Examples of optional packages include JSR-120 (Wireless Messaging API), JSR-135 (Mobile Media API), JSR-82 (Bluetooth), JSR-80 (USB), JSR-177 (Security and Trust APIs), JSR-172 (Web Services) and more. The Java Specification that ties all of these together is JSR-185–Java Technology for the Wireless Industry. This Java Specification Review (JSR), due out in 2003, will specify the mandatory JSRs that must be supported by wireless client devices along with a list of optional JSRs that may be supported.

Java is predicted to be the dominant technology for wireless client devices through to 2007 [9]. It is expected that, on average, 50% or more of the applications running on wireless client devices will be Java applications. A breakdown of the predicted amount of data traffic generated by Java applications (as a percentage of all technologies) by type of application is shown in the following table.

	2002	2003	2004	2005	2006	2007
Messaging	2%	6%	13%	26%	44%	67%
m-Commerce	7%	16%	29%	50%	65%	77%
Content	<1%	12%	26%	38%	51%	63%
Information	<1%	<1%	2%	4%	8%	15%
Location-based Services	3%	8%	16%	29%	45%	64%
Industry Applications	6%	10%	18%	36%	56%	85%
Intranet	2%	5%	14%	30%	50%	73%
Total	3%	9%	18%	31%	46%	62%

Table 1: Data traffic generated by Java applications

In terms of numbers, it is predicted that there will be 691.6 million Java-enabled phones in the marketplace by 2007 out of a total 727.3 million handsets (95% of the market). This is a significant number and could increase as alignment improves between the J2ME platform and the J2SE platform. This improvement in alignment is due to the potential for improved application portability across a wider range of devices, including the desktop, mobile phones, and PDAs.

OTHER MANAGED RUNTIME ENVIRONMENTS

In 2000, Microsoft introduced .NET*, an architecture intended to accomplish many of the same tasks as Java*. According to Microsoft, .NET intends to provide more coherence and less fragmentation across device types for application developers, providing a true "write once, run anywhere" experience on Microsoft-based platforms. What makes .NET attractive is the fact that it does allow a developer to write an application once, in the most commonly used Microsoft Visual Studio tools, and deploy it across a variety of Microsoft-based platforms. Similar to the deployment of Java, the larger, enterprise-focused .NET was available prior to the smaller footprint Compact Framework (CF) .NET, which began to roll out gradually in the 2002 releases of Microsoft Windows CE*.

* Other brands and names are the property of their respective owners.

* Other brands and names are the property of their respective owners.

.NET has a number of differences when compared to other runtime environments including Java. First, .NET supports a number of different languages including C, C++, C#, Visual Basic, and JavaScript. Using Microsoft's Visual Studio .NET, programs written in these languages are compiled into a common intermediate language representation that executes within the Common Language Runtime Environment (CLR). Some of the benefits to this approach are that existing developer skills (such as programming language experience) and existing software collateral (such as applications) may be reused with minimal effort. In other words, it's not necessary to learn C# nor to completely rewrite applications (in C#) in order to support .NET architectures.

Microsoft also claims that their driver model is more flexible in that it avoids the need for proprietary extensions and improves application portability provided the same operating system is present across all devices. Drivers in the application expose underlying hardware differences, and the device only makes use of the underlying features if it can. Applications do not need to be recoded for each device [9].

In 2001 and 2002, Microsoft signed a number of agreements with a variety of wireless network operators such as Deutsche-Telecom, indicating that the wireless industry views .NET as a possible alternative to Java. However, the main question with .NET is not if, but when, the average user will be able to utilize .NET applications and services on wireless handheld devices.

Technical Tradeoffs in a Difficult Software Environment

Even when using the smaller footprint Java or CF .NET in the wireless and handheld spaces, successfully running these managed runtime environments is a tightrope act between good enough performance and the portability of the device. Unlike in the desktop and server space, it is an ineffective use of memory to compile every bytecode in an application to native code and optimize the code during subsequent runtime iterations to maximize performance. The majority of phones on the market currently have 8MB of ROM/RAM, feature phones have up to 40MB, and only the most feature-rich segment of the market reaches 64MB. Stored bytecodes use memory far more efficiently than compiled code. Relying solely upon a Just in Time (JIT) compiler that assumes it has unlimited free space available to it, negates one of the reasons the operators selected Java for an applications framework in the first place.

What might seem to be an obvious solution to the memory and performance constraints, adding an execution unit dedicated to executing only bytecodes, turns out to have its own shortcomings. Bytecode translators are actually less efficient overall, because they cannot use compilation to improve execution efficiency for a CPU pipeline architecture or a given software program code flow. Bytecode translators also cannot use the general-purpose register set available to native execution, and because the protocol stacks for a phone are written in compiled C, the processor must attempt to execute both the native processor instruction set as well as the bytecodes. Even after adding the bytecode translation units to RISC processors, there are still a number of bytecode instructions that must be emulated vs. executed directly. This is due to the higher level of abstraction inherent in Java: a single bytecode operation often cannot map directly to a single RISC CPU instruction since the operations required by some bytecode operations are, in reality, a combination of RISC CPU instructions. To have a direct 1:1 mapping between bytecode operations and CPU instructions requires significant enough changes and additions that the benefits of adopting a RISC architecture because of lower power consumption and reduced complexity could be completely lost.

What enabled Java to become small enough to run on a phone is also a tradeoff between integration and functionality with standard systems running Java in carrier or enterprise infrastructures. In order to downsize Java to fit into an unfriendly climate like the wireless handset, Sun made a number of tradeoffs, essentially cutting Java to the bare essence. So Java developers from the J2SE or J2EE world find themselves without any of the standardized graphics (AWT/Swing) and without the security libraries in the J2ME CLDC/MIDP 1.0 world. It is only slightly better in the J2ME CDC world: the graphics libraries are added, but they are not the most recent version adopted into J2SE (e.g., AWT vs. Swing).

Currently the CF.NET situation has improved coherence, but is limited to the high-end segment of the phone market. While Java can be fit into all but the least expensive of the handsets, CF.NET awaits the inevitable increase in capability and lowering of cost for memory and processing components (sometimes referred to as "Moore's Law") to catch up to the memory and performance requirements necessary to adequately support it.

THE INTEL APPROACH TO MANAGED RUNTIME ENVIRONMENTS

The Intel approach to the challenges presented in the wireless runtime world is to achieve the most optimal balance between speed, memory use, and power efficiency and to focus on decreasing the fragmentation of functionality in the wireless space. While Intel believes

Java* and CF.NET* are key technologies in wireless for many of the reasons mentioned above, the power of these managed runtime environments is in using them as a unifying framework to simplify application development experience. "Write once, run anywhere" should not be applied without accounting for the challenges specific to running in a handheld wireless device (e.g., intermittent, low-bandwidth connections).

The first underlying principle in Intel's wireless managed runtime approach is to focus on advanced, low-power, high-performance and scalable processors such as the Intel® XScale™ microarchitecture core. Rather than adding bytecode translation, which has its own problems, the focus of Intel's efforts is on providing a processor capable of scaling a range of devices running Java, CF.NET, and native code efficiently. For the simplest phones running basic Java, components utilizing the Intel XScale microarchitecture running directly out of execution-capable flash memory is expected to be an appropriate balance of performance and functionality for a phone that is essentially free with the purchase of a network contract. By running directly from the flash memory, execution speed is increased and memory is saved. In this situation, the interpretation process does not copy into RAM in order to run. The program executes directly and immediately from its original place in the device memory.

The mid-range of functionality adds a self-limiting Just in Time (JIT) compiler. Other descriptions include "dynamic, adaptive" or "small footprint" JIT. The primary difference between the small footprint JIT and the larger JIT compilers seen in the PC world is the fact that the system integrator is able to determine how much memory the JIT may use within that specific piece of equipment. The smaller footprint JIT compilers limit themselves to a specific area in memory for "scratch pad" space during the process that compiles Java bytecodes into native code. The power of a JIT compiler is that by and large, compilation is more efficient at executing code over the lifetime of the program execution as a whole. The goal is to determine on an iterative basis how to best optimize the code that executes the most often: where the branches are taken, what branches are not taken, which variables to store and which are transient, and how code blocks can be rearranged to execute most efficiently on a given CPU architecture. The difference is that the JIT compiler process happens on the fly, and in the case of the wireless handheld devices, it happens within a limited space in memory. In this scenario, the easiest way to maximize the available memory to the JIT compiler is to execute the JIT from flash memory, leaving the available RAM space for the results of the compilation process. Eliminating the need to copy the compiler into shadow RAM saves both time and memory space for the operation of the JIT compiler.

Another related process to the JIT compiler that increases performance is to pre-compile the libraries required by the Java profile in use to native code. The goal is to selectively compile the most performance-critical libraries used by applications popular in the wireless space to balance the speed of native code against the memory compactness of Java bytecodes. In wireless, the most popular Java applications currently are entertainment, which are game and graphics focused. Hence, the libraries supporting the user interface and the multimedia operations (optional in some profiles but highly recommended by Intel for an improved user experience) should be the first order of business for system implementers. The primary difference between a feature phone and the higher end smartphones and PDAs is the memory dedicated to improved JIT compiler performance as well as the ability to pre-compile all the existing libraries to native code vs. selecting a few key areas to focus on for improving performance.

FRAGMENTATION AND RE-UNIFICATION

The specification of CLDC 1.0 and MIDP 1.0 were constrained by the limited amount of processor power and memory. However, recent advances in hardware technology have all but eliminated these constraints and it is not uncommon to find devices with processor speeds of 200-400MHz and total memory budgets of 40MB. This is a far cry from the 328KB of memory allotted for CLDC 1.0/MIDP 1.0.

The removal of the constraints that drove the specification of CLDC 1.0 and MIDP 1.0 means that these devices may easily support the Connected Device Configuration (CDC). CDC is the big brother of CLDC, and together with the Foundation Profile (FP), provides the same platform functionality as the J2SE version 1.3.

The J2ME architecture of a CDC-based device is shown in Figure 3 below. Similar to CLDC, CDC provides the common platform services while the FP and Personal Profile (PP) address the needs of a particular vertical market.

* Other brands and names are the property of their respective owners.

Intel® XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

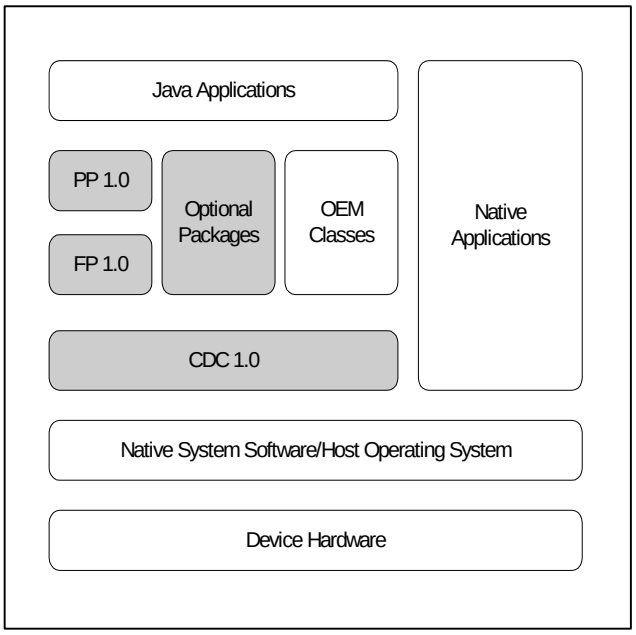


Figure 3: CDC/FP/PP architecture

The PP is similar in scope to MIDP in that it provides user interface libraries, networking libraries, and the application model. Unlike MIDP, the PP is compatible with the J2SE* (e.g., the PP supports AWT).

Many of the optional packages that have been developed to date are reusable across both CLDC and CDC platforms (e.g., JSR-120 [Wireless Messaging API], JSR-135 [Mobile Media API], etc.).

By moving towards a CDC-based J2ME architecture, many of the fragmentation and portability issues associated with CLDC/MIDP can be addressed and a closer alignment with the J2SE achieved. This will facilitate the development of common client devices that may be run on both the desktop and wireless client devices.

Figure 4, below, shows how the transition from today's CLDC/MIDP-based J2ME architecture to tomorrow's CDC/FP/PP-based J2ME architecture might be accomplished. On the left-hand side is the situation today: a CDC-based software stack and a CLDC-based software stack. Moving to the right, CDC is adopted as the configuration platform for MIDP with the benefit that MIDP and MIDP applications now enjoy full Java Language and Java Virtual Machine support. The next phase in the migration supports a transition period between MIDP and PP. It is during this time that MIDP applications are ported to the PP. The final stage on the right is the result of the migration: a single application development and runtime environment based on CDC/FP/PP. The CLDC/MIDP stack is no longer required nor should it be used.

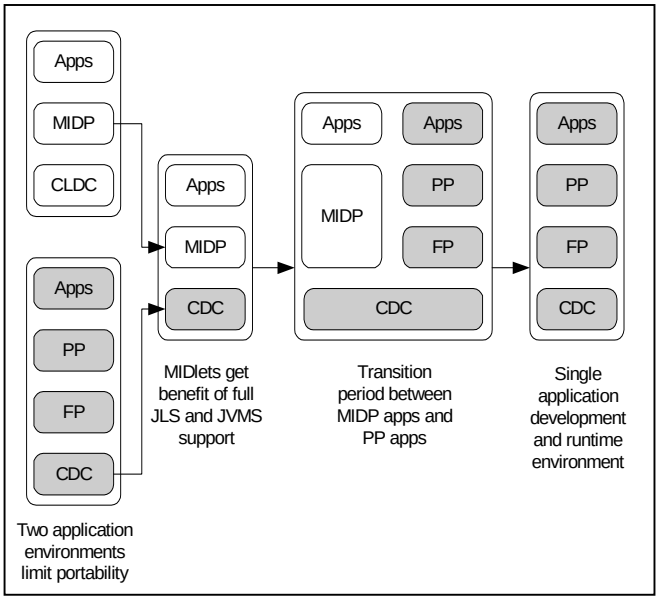


Figure 4: CLDC/MIDP evolution strategy

The proposed CDC-based platform that forms the basis of the Intel PCA Java architecture is shown in Figure 5 below. Along with CDC, FP, and the PP, it also shows the optional packages, organized by application type, that are required to support the predicted growth of Java* applications for wireless client devices[3].

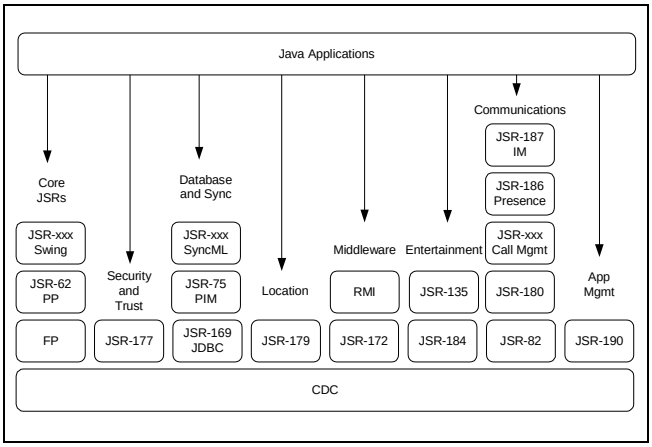


Figure 5: Proposed CDC/FP/PP platform

* Other brands and names are the property of their respective owners.

* Other brands and names are the property of their respective owners.

[3] Those JSRs that do not have numbers associated with them do not yet exist.

CONCLUSION

The handheld and wireless markets have their own unique challenges and opportunities. An emphasis on the ability to connect to back-end infrastructure, the need for highly compressed and efficient data transmission and highly constrained screen and battery all shift the dynamics of the traditional development environment.

Runtime abstraction solves many problems presented to the average application developer, and it presents many challenges to the hardware implementation teams in adapting to the unique execution environment. The Intel philosophy on balancing application developers' desires for simplicity and coherence against the hardware implementers' desire for low memory and high performance is to appropriately select and tune key components in the system. A high-performance yet low-power general-purpose processor combined with execution-capable flash memory and selectively tuned native components provides the basis for a wide range of wireless client devices.

JSR-185 (Java* Technology for the Wireless Industry) is slated to be completed in 2003, and it will address some of the fragmentation concerns with respect to J2ME. It will provide an architectural overview of the essential client components of an end-to-end wireless solution including recommended combinations of J2ME components (i.e., configurations, profiles, and optional packages). The availability of this architectural specification will also be used to trigger compatibility requirements that, in turn, will be reflected in the associated Technology Compatibility Kit (TCK) that is used to determine conformance.

At the same time, .NET* will also be evolving and spreading throughout the wireless ecosystem as Microsoft-based platforms are deployed more widely. While .NET-based platforms do not face as many integration challenges due to the fact that the number of variables decrease, the rollout of .NET devices is just beginning to ramp.

Regardless of the managed runtime environment an application developer selects, Intel technologies create a balanced and flexible platform upon which the hardware implementer and application developer have freedom to innovate and differentiate their solutions and products. As the platform costs decrease and the available performance and memory increase, the user experience improves. It is expected that the wireless client device will become a significant force in the computing industry at large.

REFERENCES

[1] http://www.wired.com/news/mac/0,2125,54580,00.html, Apple's Newton Just Won't Drop, Leander Kahney, Aug. 29, 2002 PT.

[2] http://www.fortune.com/fortune/fsb/specials/innovators/dubinsky.html , "HOW WE GOT STARTED," Donna Dubinsky.

[3] http://www.microjava.com/articles/perspective/shosteck?content_id=2179, "The Battle For BREW, J2ME And Related Technologies," The Shosteck Group, 10/29/2001.

[4] http://java.sun.com/features/2000/06/time-line.html, "The Java Platform: Five Years in Review."

[5] http://more.abcnews.go.com/sections/business/dailynews/silicon_insights_seybold_010716.html#1, "Spectral Efficiency, Which technologies work best?," Andy Seybold, July 2002.

[6] http://www.byte.com/documents/s=693/byt19990811s0006/index.htm, August 1999.

[7] http://www.pctel.com/cellular_problem.php, other references available upon request.

[8] Roger Riggs et. al., "Programming Wireless Devices with the Java 2 Platform," Micro Edition.

[9] ARC Group, "Wireless Java 2002 Handset and Application Revenue Streams."

AUTHORS' BIOGRAPHIES

Lynn Comp is a strategic marketing engineer in the Wireless Computing and Communications Group at Intel Corporation and has been active in the portable and wireless computing market for the last six years, setting strategy for WCCG in runtime environments and software platforms. Prior to her involvement in the wireless and portables' market, Lynn was an applications engineer on data communications silicon supporting customers developing routers, cellular basestations, and WAN/LAN bridges. Lynn has a B.S.E.E. degree from Virginia Tech and an MBA degree in Technology Management from the University of Phoenix. Her e-mail is lynn.a.comp@intel.com

Tim Dobbing has more than 15 years of software architecture, design, and implementation experience in the telecommunications industry in the areas of network management, high availability, traffic management, and call processing for a variety of technologies including CDMA, ATM and ISDN. In his current position as a J2ME Architect at Intel Corporation, his focus is on the specification of a J2ME reference architecture for Intel's Personal Internet Client Architecture (PCA). Other Java

experience includes the architecture, design, and implementation of a Web-based Management solution for a CDMA Base Station Subsystem using J2SE and PersonalJava. Tim received a B.E. and an M.E. degree in Electrical Engineering in 1984 and 1990, respectively from Carleton University in Ottawa, Canada, and he is a member of the Association of Professional Engineers, Geophysicists, and Geologists of Alberta (APEGGA). His e-mail is timothy.c.dobbing@intel.com

Managed Runtime Environments for Next-Generation Mobile Devices

Paul Drews, Intel R&D, Intel Corporation
Doug Sommer, Intel R&D, Intel Corporation
Roger Chandler, Intel R&D, Intel Corporation
Terry Smith, Intel R&D, Intel Corporation

Index words: Mobile, Handset, Wireless, Services, Mobile Data, Operator, Mobile Applications

ABSTRACT

The adoption of cellular communications has been one of the fastest growing technology trends in history. Many analysts predict that the demands of our increasingly wireless world will result in the rapid convergence of cellular communications and powerful, handheld computing devices, enabling a wide array of exciting new user experiences. By 2006, the analyst firm Instat/MDR predicts there will be over 760M Internet-enabled mobile devices in use (1), and the ARC Group predicts that by 2007, over 1.7 billion users will utilize wireless data services (2). Intel is a leading building block supplier to this new converged device industry with our Flash memory products, our high-performance Intel® XScale™ mobile application processor family, and our Personal Internet Client Architecture (PCA) for mobile computing devices and handsets. Intel is also developing key technologies that will accelerate the adoption of managed runtime environments (MRTEs) for mobile devices. The industry predicts that the majority of converged devices will include MRTEs (primarily in the form of J2ME and .NET*) and because of this, MRTEs are a key component of Intel's overall mobile industry strategy.

This paper describes how MRTEs are important enabling technologies for the future of wireless computing and how they are contributing to the fast delivery of wireless data services. The term "managed runtime environment" as used in this paper refers to new functionality and technologies that extend the capabilities of the first generation of runtime environments, notably, the Java* Virtual Machine and the Microsoft .NET Framework. These first-generation MRTEs need to evolve further to serve the demands of the mobile data market. We also illustrate how PCA is the ideal platform to take full advantage of MRTEs, and we conclude with descriptions of Intel's R&D to enable the next generation of MRTEs for mobile devices

INTRODUCTION

The launch of cellular communications several years ago had one purpose in mind: mobile voice communication. Analog-based voice communication was the sole purpose of cellular handsets for many years until the transition to digitized voice data came about in the 1990s. Regardless of the type, cellular handsets and networks were architected and deployed to accommodate voice traffic. It was not until the introduction and ensuing popularity of Short Messaging Services (SMS) that cellular operators and handset manufacturers began to explore the possibilities and ramifications of building systems capable of handling a wide range of digital datatypes. The promise of 2.5G and 3G networks, widely deployed wireless LANs, in combination with the Internet explosion, has fueled a global demand for cellular handsets that deliver both voice communications and general-purpose computing capabilities.

Most cellular network operators and handset manufacturers have either announced or begun deployment of first-generation data applications and services for cellular handset users. These initial applications include 2D games, electronic mail, multi-media messaging, personal information management, and a host of other applications once found only on personal computers. As more data-intensive mobile applications are deployed, developers and service providers are

encountering troublesome issues stemming from the heterogeneity of today's mobile device platforms. In order to successfully deploy a wide range of mobile data applications and services, the device platform itself needs a consistent software interface layer that developers can rely upon when developing and deploying applications. This will serve to not only insulate the mobile application developer from the underlying hardware and software variables of the device, but also create a "common platform" necessary to jump-start a new wave of mobile applications. This interface layer, so to speak, must also protect the integrity of the device's core capabilities and enable the service provider to more effectively install, manage, and maintain the applications and services on the device. This layer is a managed runtime environment (MRTE).

BENEFITS OF MANAGED RUNTIME ENVIRONMENTS

The term "managed runtime environment" as used here refers to new functionality and technologies that extend the capabilities of the first generation of runtime environments, notably the Java Virtual Machine and the Microsoft .NET* Framework. MRTEs provide an exciting array of benefits to mobile application developers, operators, and end users, as follows:

- A platform-independent programming environment that makes it far easier (than native code) to move applications between platforms.
- A sandbox runtime environment that prevents rogue programs from disrupting the platform.
- Garbage-collection-style memory management and incorrect-reference (pointer) protection that together nearly eliminate a major source of programming errors.
- A dynamic code-loading mechanism that makes it easier to extend platform capabilities with new applications and class libraries.

These benefits are so substantial that MRTEs should be considered an essential design element of all new mobile device designs. Figure 1 below represents the architectural framework for an MRTE.

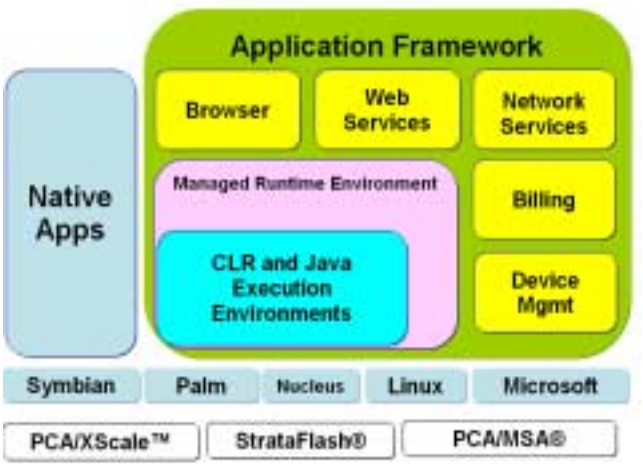


Figure 1: Architectural framework for MRTEs

Further Development of MRTEs

While MRTEs provide many benefits, today's managed execution environments need to evolve further to serve the growing demands of the mobile communications and computing industry. Increasing the intelligence and capabilities of mobile clients introduces many new applications and services to mobile devices, such as Web access, e-mail, and multimedia application processing. With these new capabilities, it is critical that the communication capabilities and integrity of the device be protected, even in the presence of unauthorized applications. Several kinds of protection need to be built into the platform to ensure highly reliable communications, including the following:

- Communications need to be isolated from applications.
- Applications need to be isolated from one another.
- Resource management and recovery need to be built into the platform.

Wireless communications also need to guard against incorrect or malicious devices in the environment.

ESSENTIAL BUILDING BLOCKS FOR THE NEXT GENERATION OF MANAGED RUNTIME ENVIRONMENTS

Through internal technology development and partnering with key technology providers in the industry, Intel is committed to ensuring future MRTEs meet the requirements of next-generation mobile applications and services. To deliver on this vision, Intel is creating building blocks that meet the following criteria:

* Other brands and names are the property of their respective owners.

- *Standardized.* The mobile software development environment demands this attribute in order for manufacturers and operators to ensure interoperability. Communities and standards groups such as the Open Mobile Alliance (OMA) and the Java Community Process (JCP) are therefore essential.
- *Open.* The best specifications are those that are developed by a wide array of contributors, thereby meeting the needs of the entire mobile industry. Intel sees tremendous value in creating building blocks whose functionality was specified by many contributors.
- *Optimized*. Intel is working diligently to ensure that MRTE building blocks run efficiently on our architectures and deliver value to application developers, original equipment manufacturers (OEMs), and carriers.
- *Scalable.* Application developers are confronted by a myriad of platform choices in targeting applications. Requiring a unique application image for each platform increases development cost and complexities for independent software vendors (ISVs) and carriers. Enabling seamless scalability across the range of mobile platforms, from low-end cellular terminals to high-end "Smart Phones," is a key design requirement.
- *Adaptable.* MRTEs and the applications that take advantage of them are evolving rapidly. In anticipation of the needs of newly emerging MRTE standards, the low-level building blocks provided by mobile platforms need to be more general-purpose and more powerful than what is offered today. This enables rapid time-to-market for system designers by allowing them to adapt the platform's foundation capabilities quickly and efficiently to the new standards.

With these criteria in mind, Intel and its partners are working hard on the next generation of MRTE building blocks, which are described next.

MRTE Building Blocks

Described below are some of the attributes of the next generation of MRTE building blocks.

- *Advanced dynamic compilers*. The initial versions of MRTEs for cellular terminals were generally reliant on interpreted execution. This was fine for simplistic data applications, but it fails to meet the performance requirements of the latest mobile applications and services. Combined with the latest advances in mobile application processor technology found in the Intel® XScale™ technology family, advanced dynamic compilers deliver superior performance within a memory-constrained environment.
- *Platform management.* One of the most resource-intensive aspects of cellular operations is the customer care requirement for deployed handsets. Diagnosing and resolving customer problems in an efficient manner can enhance the operator's bottom line. New platform management technologies will enable carrier operations to more quickly spot and fix software problems and identify hardware issues for replacement or repair.
- *Extended battery life*. A key criterion in the end-user selection process is battery life, both standby and talk-time. The addition of data services will only increase the demand for battery-saving technologies. Intel is hard at work, both at the platform level and in the MRTE environment, to deliver power management capabilities and more power-efficient components and building blocks.
- *Flash management*. The Intel Personal Internet Client Architecture (PCA) supports scaling of flash memory over a wide range of densities and mid-level building blocks for a flash file system.
- *Secure provisioning.* The industry's "best known methods" for checking the integrity and permissions of software and commands centers on verifying digital signatures against configured public keys that specify their authorized source. A secure provisioning building block provides capabilities for configuring authorization keys; verifying digital signatures of software, commands, or other data; defining sets of permissions and associating them with authorities; and checking permissions. This foundation building block can easily be adapted to implement current and emerging provisioning standards.
- *Resource monitor and recovery*. These functions provide a generic mechanism for the managed runtime environment to track the allocation of system resources, such as peripherals, and to recover the resources in the event of an unexpected application termination or failure to return resources.

- *User-data management*. These interfaces provide a framework for applications to separate data created by the user from application installation data, giving the ability to backup, remove, and restore user data and application data independent of one another.

- *Data location management*. This building block supports an evolving industry trend toward flexibility and transparency regarding the actual location (local or remote) of a persistent data object to promote "go-anywhere, access-anytime" availability of data.

DELIVERING A MOBILE DEVICE PLATFORM OPTIMIZED FOR MANAGED RUNTIME ENVIRONMENTS

Intel has a long history of delivering building block components to the computing industry. Delivering high-performance platform components for the converged communications and computing industry is a top strategic priority for Intel. Achieving the best-possible performance often involves a particular challenge: on the one hand, a performance improvement frequently takes advantage of a particular hardware feature; while on the other hand, it is important to keep applications free from particular hardware dependencies so that they can scale to a wide variety of devices.

A highly effective technique for meeting both of these criteria is to identify areas where performance is critical, optimize them with native features at a low level, and adapt these low-level optimizations to widely used, standardized industry interfaces. In this way, applications only "see" standardized interfaces, so they may run anywhere, while still getting higher performance on optimized platforms.

Some current and future fruitful areas that Intel has found for optimization are briefly described below:

- *Intel® Integrated Performance Primitives*. These primitives provide high-speed implementations of functions used in algorithms such as multimedia codecs (encoder/decoder engines). They significantly reduce the time and effort spent on algorithm development. The Intel Performance Primitives Library is a low-level building block that may be used as the foundation for other mid-level building blocks such as media playback and graphics.

- *Real-time media playback.* A rapidly emerging use for mobile devices is in the area of playback of real-time media types such as audio and video, either from streaming or stored sources. This tends to demand high computation rates to achieve good playback quality and high compression or decompression ratios. A current R&D effort is focused on producing a high-performance optimized JSR 135 Java Mobile Media framework, including a set of audio and video players for popular formats.

- *High-performance graphics.* Mobile devices are experiencing rapidly increasing use for graphics-intensive gaming. Today's mobile games emphasize 2D animation, scrolling, and sprites. As this trend continues, we can expect such games to expand to 3D graphics. Future R&D efforts will focus on producing optimized implementations of the mobile information device profile (MIDP) 2.0 2D gaming additions and the JSR 184 3D graphics libraries.

- *Execute in place*. This library provides functions for optimizing application modules for high-speed start-up directly out of flash memory. It does not have a distinct interface directly available to applications but would instead be integrated into the loader or the optimizing compiler of a managed runtime environment (MRTE) virtual machine.

Delivering high-performance, memory-efficient, and energy-friendly MRTE solutions is a tricky balance. MRTE solutions must also conform to industry standards and scale across a range of platform capabilities. The need for optimized libraries grows along with the rapid emergence of new MRTE libraries. One company alone is not up to the optimization task. Intel has developed a number of initiatives and established a broad range of industry partnerships to enable "best-of-class" MRTE support. Some of these efforts are described below:

- *Tools.* Intel has developed or supported the development of several different optimizing tool-chains for the Intel® XScale™ software creation. Details can be found at http://www.intel.com/software/products/. In addition, Intel has released a version of its award-winning VTune optimization tool to better support software developers creating high-performance software for our architecture.

- *Guides.* The Intel XScale architecture is a tremendous advance over our previous StrongARM implementation, both in terms of processor clock frequency and in the detail of its microarchitecture. To assist the software developer in building high-

® Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Intel® XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

performance Intel XScale applications and MRTEs, developer documentation can be found at http://www.intel.com/design/intelxscale.

- *Partnerships.* Intel has established a number of critical relationships for "best-of-class" MRTE solutions. Many of these have yet to be publicly announced. Combined with superior platform and architecture technology, these efforts will enable new classes of mobile applications and services that will drive new business opportunities for operators and manufacturers alike.

CONCLUSION

Mobile data applications represent the next big revenue opportunity for the wireless industry. New hardware and software technologies will enable mobile handsets to run Internet-aware applications and Web services, and managed runtime environments like Java* and Microsoft .NET* Framework are making it easier to quickly create, deploy, and manage mobile data applications. Managed runtime environments effectively insulate applications from the variables of the systems they run on, reducing development time and easing deployment. They also help provide secure, manageable, and connected applications to users, increasing the demand for data services on wireless networks.

Still, the technological and business challenges facing today's mobile application developer are complex, daunting, and rapidly changing. Managed runtime environments provide the device independence, software portability, speed of development, and security that today's Internet applications demand, while the Intel Personal Internet Client Architecture, coupled with Intel's software building blocks allow hardware and software developers to more easily implement these capabilities in new handheld designs. These systems can take advantage of low-level hardware features for efficiency while supporting high-level standards, thus allowing application interoperability across a wide range of devices.

REFERENCES

[1] Instat/MDR, *Mobile Internet Access Devices, 2002-2007*, April 2002.

[2] ARC Group, *Future Mobile Handsets: Worldwide Technology & Market Developments, 2002-2007*, April 2002.

* Other brands and names are the property of their respective owners.

AUTHORS' BIOGRAPHIES

Paul Drews is a senior software engineer in the Emerging Platforms Lab, currently working on performance-optimized Java* libraries for Intel® XScale™ technology. Mr. Drews has been with Intel for over 20 years, and he has been awarded eight patents in the areas of operating system performance, graphics, distributed hypertext, and security. He also received an Intel Achievement Award for his work on the WinSock 2 project. Drews received his B.A. degree from Luther College with triple majors in physics, mathematics, and computer science. His e-mail is paul.drews@intel.com.

Doug Sommer is a senior engineering manager in Intel's Emerging Platforms Lab. Doug and his group are focused on high-performance virtual machine technologies and new manageability and security capabilities for mobile platforms. Doug has worked for Intel for the last eleven years on projects such as development tools, digital video, new media, Java technology, and mobile platform building blocks. He holds a B.S. degree in Computer Science from Oregon State University. His e-mail is doug.sommer@intel.com.

Roger Chandler is a Market Development Manager in Intel's Emerging Platform Lab. While at Intel, his areas of focus have included manufacturing process analysis, high-performance 3D technologies, and home networking. He was a 2001 recipient of an Intel Achievement Award for his work in the field of Web 3D. He is currently focused on market development strategies for Intel's many wireless technology initiatives. He holds an MBA degree from the University of Georgia and a B.A. degree from the University of Tennessee. His e-mail is roger.d.chandler@intel.com.

Terry Smith is a Business Development Manager in Intel's Emerging Platform Labs, focusing on Intel's work in Managed Runtime Environments and Mobile Architectures. Previously he was responsible for Intel's Common Data Security Architecture strategy. He holds an MBA from the University of Texas-Austin and a B.S. degree in Math/CS from the University of Illinois. His e-mail is terry.a.smith@intel.com.

* Other brands and names are the property of their respective owners.

For further information visit:

developer.intel.com/technology/itj/index.htm