The Jalapeño virtual machine

by B. Alpern
C. R. Attanasio
J. J. Barton
M. G. Burke
P. Cheng
J.-D. Choi
A. Cocchi
S. J. Fink
D. Grove
M. Hind
S. F. Hummel
D. Lieber
V. Litvinov
M. F. Mergen
T. Ngo
J. R. Russell
V. Sarkar
M. J. Serrano
J. C. Shepherd
S. E. Smith
V. C. Sreedhar
H. Srinivasan
J. Whaley

Jalapeño is a virtual machine for Java™ servers written in the Java language. To be able to address the requirements of servers (performance and scalability in particular), Jalapeño was designed "from scratch" to be as self-sufficient as possible. Jalapeño's unique object model and memory layout allows a hardware null-pointer check as well as fast access to array elements, fields, and methods. Run-time services conventionally provided in native code are implemented primarily in Java. Java threads are multiplexed by* virtual processors *(implemented as operating system threads). A family of concurrent object allocators and parallel type-accurate garbage collectors is supported. Jalapeño's interoperable compilers enable* quasi-preemptive *thread switching and precise location of object references. Jalapeño's dynamic optimizing compiler is designed to obtain high quality code for methods that are observed to be frequently executed or computationally intensive.

Jalapeño is a Java** virtual machine (Jvm) for servers. The memory constraints on a server are not as tight as they are on other platforms. On the other hand, a Jvm for servers must satisfy requirements such as the following that are not as stringent for client, personal, or embedded Jvms:

1. *Exploitation of high-performance processors*—Current just-in-time (JIT) compilers do not perform the extensive optimizations for exploiting modern hardware features (memory hierarchy, instruction-level parallelism, multiprocessor parallelism, etc.) that are necessary to obtain performance comparable with statically compiled languages.
2. *SMP scalability*—Shared-memory multiprocessor (SMP) configurations are very popular for server machines. Some Jvms map Java threads directly onto heavyweight operating system threads. This leads to poor scalability of multithreaded Java programs on an SMP as the numbers of Java threads increases.
3. *Thread limits*—Many server applications need to create new threads for each incoming request. However, due to operating system constraints, some Jvms are unable to create a large number of threads and hence can only deal with a limited number of simultaneous requests. These constraints are severely limiting for applications that

need to support thousands of users simultaneously.
4. *Continuous availability*—Server applications must be able to satisfy incoming requests while running continuously for long durations (e.g., several months). This does not appear to be a priority for current Jvms.
5. *Rapid response*—Most server applications have stringent response-time requirements (e.g., at least 90 percent of requests must be served in less than a second). However, many current Jvms perform nonincremental garbage collection leading to severe response-time failures.
6. *Library usage*—Server applications written in Java code are typically based on existing libraries (beans, frameworks, components, etc.) rather than being written "from scratch." However, since these libraries are written to handle generic cases, they often perform poorly on current Jvms.
7. *Graceful degradation*—As the requests made on a server oversaturate its capacity to fulfill them, it is acceptable for the performance of the server to degrade. It is *not* acceptable for the server to crash.

In Jalapeño, Requirement 1 is addressed by a dynamic optimizing compiler; lighter-weight compilers are provided for code that has not been shown to be a performance bottleneck. Requirements 2 and 3 are addressed by the implementation of lightweight threads in Jalapeño. Implementing Jalapeño in the Java language addressed Requirement 4: Java type safety aids in producing correct code, and the Java automatic storage management prevents "dangling pointers" and reduces "storage leaks." We expect Requirement 5 to be satisfied by the concurrent and incremental memory management algorithms currently being investigated. Requirement 6 will be satisfied by specialization transformations in the Jalapeño optimizing compiler that tailor the dynamically compiled code for a library (for example) to the calling context of the server application. Although we know of no programmatic way to guarantee satisfaction of Requirement 7, we try not to lose sight of it.

The paper is organized as follows. The next section considers implementation issues. The following section presents the Jalapeño Jvm, including its object model and memory layout and its run-time, thread and synchronization, memory management, and compilation subsystems. Following sections examine Jalapeño's optimizing compiler, describe Jalapeño's current functional status and give some preliminary performance results, and discuss related work. The final section presents our conclusions. Two appendices are included to explain how Jalapeño's run-time services evade some Java restrictions while preserving the integrity of the language for Jalapeño's users, and to detail the process of bootstrapping Jalapeño.

Design and implementation issues

The goal of the Jalapeño project is to produce a world-class server Jvm "from scratch." Our approach is to create a flexible test bed where novel virtual machine ideas can be explored, measured, and evaluated. Our development methodology avoids premature optimization: simple mechanisms are initially implemented and are refined only when they are observed to be performance bottlenecks.

Portability is *not* a design goal: where an obvious performance advantage can be achieved by exploiting the peculiarities of Jalapeño's target architecture—PowerPC* architecture[1] SMPs (symmetrical multiprocessors) running AIX* (Advanced Interactive Executive)[2]—we feel obliged to take it. Thus, Jalapeño's object layout and locking mechanisms are quite architecture-specific. On the other hand, we are aware that we may want to port Jalapeño to some other platform in the future. Thus, where performance is not an issue, we endeavor to make Jalapeño as portable as possible. For performance as well as portability, we strive to minimize Jalapeño's dependence on its host operating system.

The original impetus for building Jalapeño in the Java language was to see if it could be done.[3] The development payoffs of using a modern, object-oriented, type-safe programming language with automatic memory management have been considerable. (For instance, we have encountered no dangling pointer bugs, except for those introduced by early versions of our copying garbage collectors which, of necessity, circumvented the Java memory model.) We expect to achieve performance benefits from Java development as well: first, no code need be executed to bridge an interlinguistic gap between user code and run-time services; and second, because of this seamless operation, the optimizing compiler can simultaneously optimize user and run-time code, and even compile frequently executed run-time services in line within user code.

The Jalapeño *implementation* must sometimes evade the restrictions of the Java language. At the same

time, Jalapeño must enforce these restrictions on its users. Jalapeño's mechanism for performing such controlled evasions is presented in Appendix A.

Very little of Jalapeño is *not* written in the Java language. The Jalapeño virtual machine is designed to run as a user-level AIX process. As such, it must use the host operating system to access the underlying file system, network, and processor resources. To access these resources, we were faced with a choice: the AIX kernel could be called directly using low-level system calling conventions, or it could be accessed through the standard C library. We chose the latter path to isolate ourselves from release-specific operating system kernel dependencies. This required that a small portion of Jalapeño be written in C rather than Java code.

To date, the amount of C code required has been small (about 1000 lines). About half of this code consists of simple "glue" functions that relay calls between Java methods and the C library. The only purpose of this code is to convert parameters and return values between Java format and C format. The other half of the C code consists of a "boot" loader and two signal handlers. The boot loader allocates memory for the virtual machine image, reads the image from disk into memory, and branches to the image startup code (see Appendix B). The first signal handler captures hardware traps (generated by null pointer dereferences) and trap instructions (generated for array bounds and divide-by-zero checks), and relays these into the virtual machine, along with a snapshot of the register state. The other signal handler passes timer interrupts (generated every 100 milliseconds) to the running Jalapeño system.

Jvm organization

Following subsections describe Jalapeño's object model, run-time subsystem, thread and synchronization subsystem, memory management subsystem, and compiler subsystem.

Java objects are laid out to allow fast access to field and array elements, to achieve hardware null pointer checks, to provide a four-instruction virtual-method dispatch, and to enable less frequent operations such as synchronization, type-accurate garbage collection, and hashing. Fast access to static objects and methods is also supported.

In conventional Jvms, run-time services—exception handling, dynamic type checking, dynamic class loading, interface invocation, input and output, reflection, etc.—are implemented by *native* methods written in C, C++, or assembler. In Jalapeño these services are implemented primarily in Java code.

Rather than implement Java threads as operating system threads, Jalapeño multiplexes Java threads on *virtual processors*, implemented as AIX pthreads.[4] Jalapeño's locking mechanisms are implemented without operating system support.

Jalapeño supports a family of memory managers, each consisting of an object allocator and a garbage collector. All allocators are concurrent. Currently, all collectors are stop-the-world, parallel, and type-accurate collectors. Generational and nongenerational, copying and noncopying managers are supported. Incremental collectors are being investigated.

Jalapeño does not interpret bytecodes. Instead these are compiled to machine code before execution. Jalapeño supports three interoperable compilers that address different trade-offs between development time, compile time, and run time. These compilers are integral to Jalapeño's design: they enable thread scheduling, synchronization, type-accurate garbage collection, exception handling, and dynamic class loading.

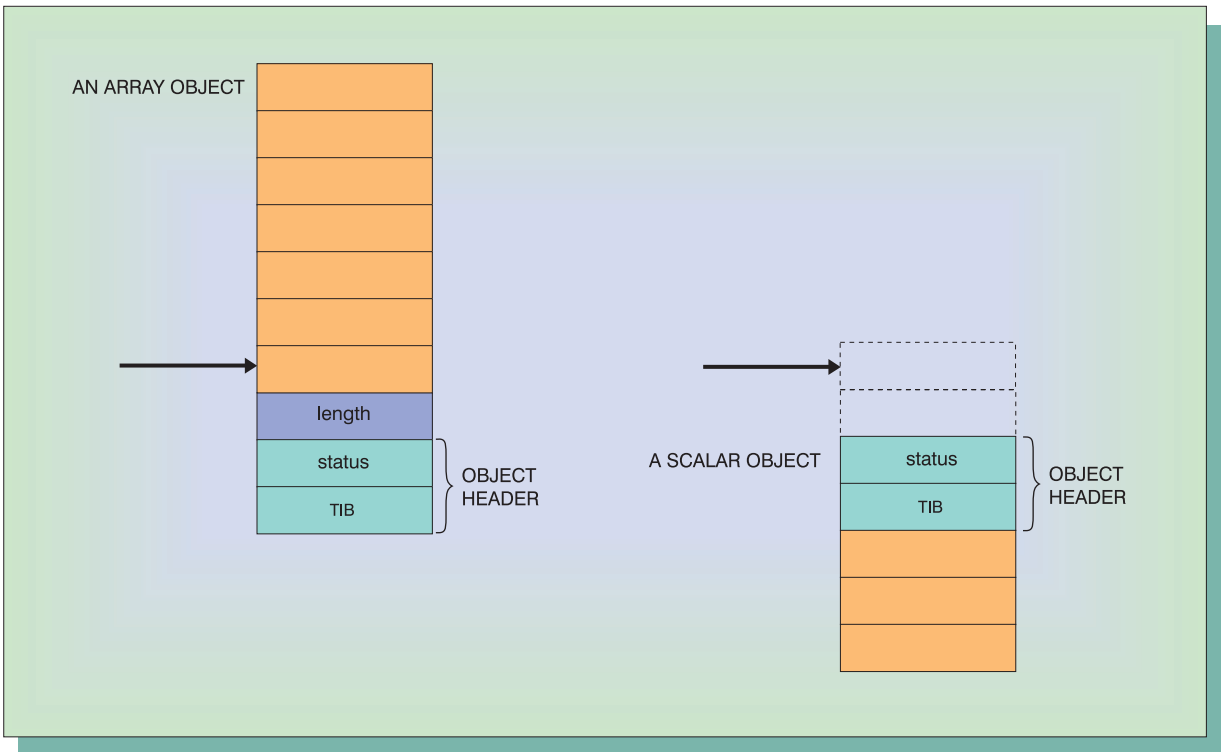
Object model and memory layout. Values in the Java language are either *primitive* (e.g., int, double, etc.) or they are *references* (that is, pointers) to objects. Objects are either *arrays* having components or *scalars* having fields. Jalapeño's object model is governed by four criteria:

- Field and array accesses should be fast.
- Virtual method dispatch should be fast.
- Null pointer checks should be performed by the hardware.
- Other (less frequent) Java operations should not be prohibitively slow.

Assuming the reference to an object is in a register, the object's fields can be accessed at a fixed displacement in a single instruction. To facilitate array access, the reference to an array points to the first (zeroth) component of an array and the remaining components are laid out in ascending order. The number of components in an array, its *length*, is kept just before its first component.

The Java language requires that an attempt to access an object through a null object reference generate a NullPointerException. In Jalapeño, refer-

Figure 1 Layout of an array object and a scalar object in Jalapeño



ences are machine addresses, and null is represented by Address 0. The AIX operating system permits loads from low memory, but accesses to very high memory, at small *negative* offsets from a null pointer, normally cause hardware interrupts.[5] Thus, attempts to index off a null Jalapeño array reference are trapped by the hardware, because array accesses require loading the array length, which is −4 bytes off the array reference. A hardware null-pointer check for field accesses is effected by locating fields at negative offsets from the object reference.

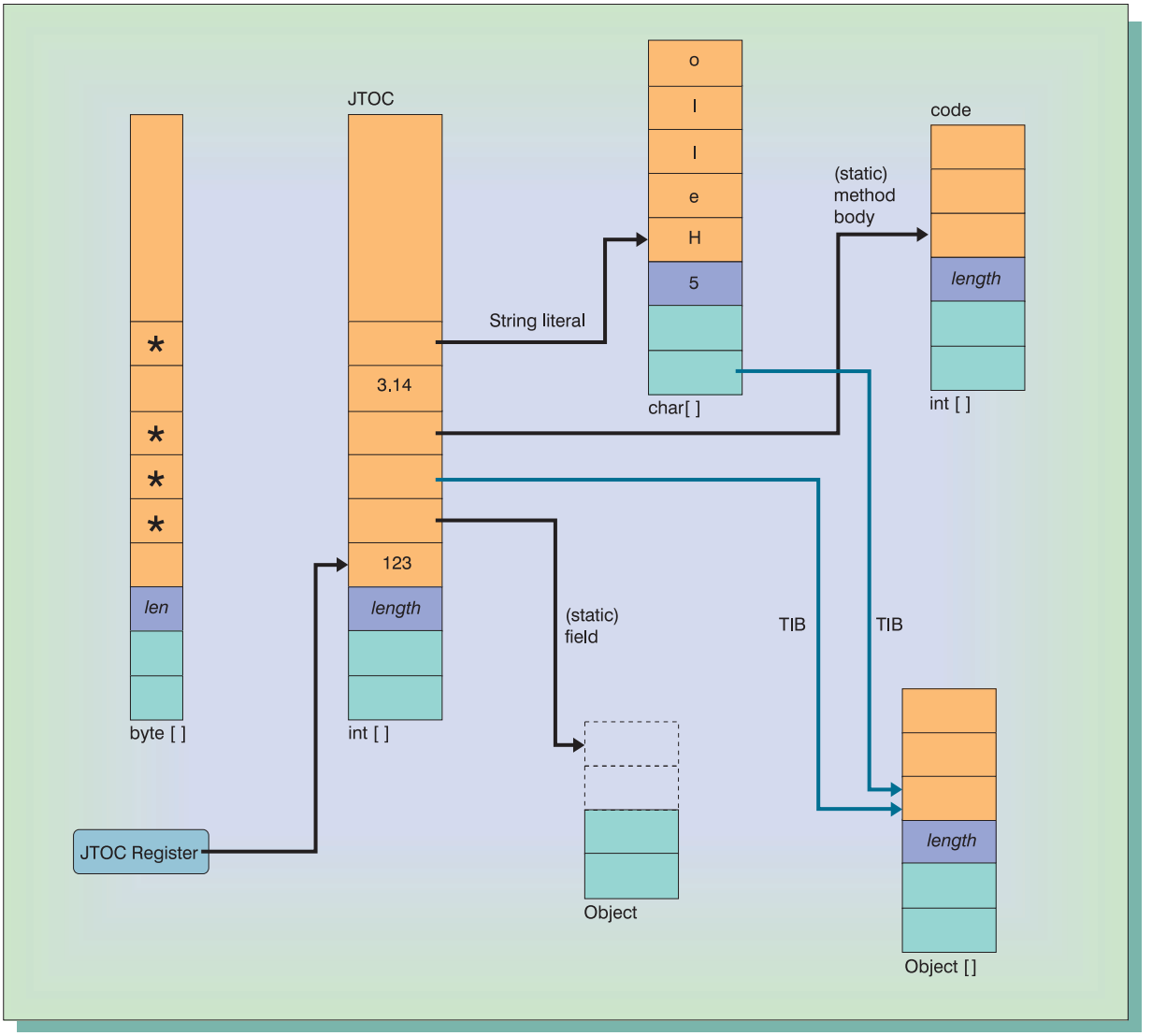
In summary, in Jalapeño, arrays grow *up* from the object reference (with the array length at a fixed negative offset), while scalar objects grow *down* from the object reference with all fields at a negative offset (see Figure 1). A field access is accomplished with a single instruction using base-displacement addressing. Most array accesses require three instructions. A single trap instruction verifies that the index is within the bounds of the array. Except for byte (and boolean) arrays, the component index must then be shifted to get a byte index. The access itself is accomplished using base-index addressing.

Object headers. A two-word object header is associated with each object. This header supports virtual method dispatch, dynamic type checking, memory management, synchronization, and hashing. It is located 12 bytes below the value of a reference to the object. (This leaves room for the length field in case the object is an array, see Figure 1.)

One word of the header is a *status* word. The status word is divided into three *bit fields*. The first bit field is used for locking (described later). The second bit field holds the default hash value of hashed objects. The third bit field is used by the memory management subsystem. (The size of these bit fields is determined by build-time constants.)

The other word of an object header is a reference to the *Type Information Block* (TIB) for the object's class. A TIB is an array of Java object references. Its first component describes the object's class (including its superclass, the interfaces it implements, offsets of any object reference fields, etc.). The remaining components are compiled method bodies (execut-

Figure 2 The Jalepeño Table of Contents and other objects

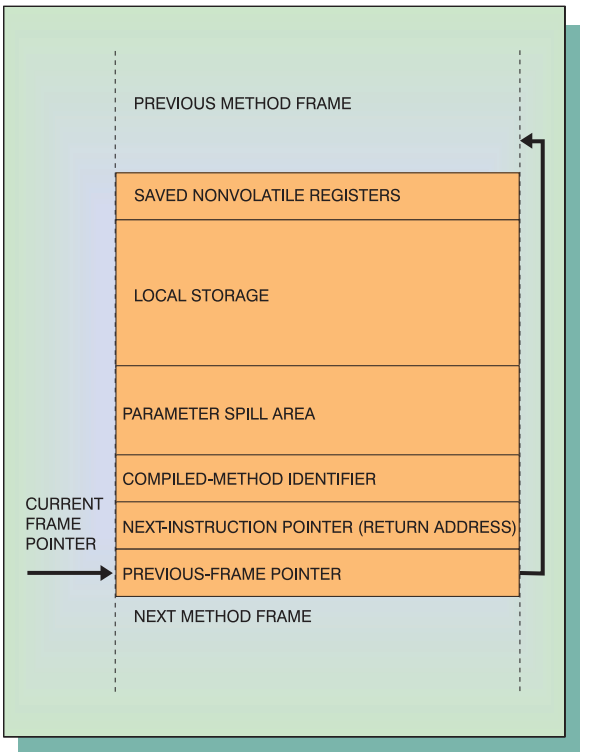


able code) for the virtual methods of the class. Thus, the TIB serves as Jalapeño's virtual method table.

Virtual methods. Method bodies are arrays of machine instructions (ints). A virtual method dispatch entails loading the TIB pointer at a fixed offset off the object reference, loading the address of the method body at a given offset off the TIB pointer, moving this address to the PowerPC "link-register," and executing a branch-and-link instruction—four instructions.

Static fields and methods (and others). All static fields and references to all static method bodies are stored in a single array called the *Jalapeño Table of Contents* (JTOC). A reference to this array is maintained in a dedicated machine register (the JTOC register). All of Jalapeño's global data structures are accessible through the JTOC. Literals, numeric constants, and references to string constants are also stored in the JTOC. To enable fast common-case dynamic type checking, the JTOC also contains references to the TIB for each class in the system. The JTOC is depicted

Figure 3 A thread's method invocation stack



in Figure 2. Although declared to be an array of ints, the JTOC contains values of all types. A JTOC descriptor array, coindexed with the JTOC, identifies the entries containing references.

Method invocation stacks. Jalapeño's stack layout is depicted in Figure 3. There is a stack frame for each method invocation. (The optimizing compiler may omit stack frames for leaf and inline methods.) A stack frame contains space to save nonvolatile registers, a local data area where usage is compiler-dependent, and an area for parameters that are to be passed to called methods and that will not fit in Jalapeño's volatile registers. The last three words in a stack frame are: a *compiled-method identifier* (identifying information about the method for the stack frame), a *next-instruction pointer* (the returned address of any *called* method), and a *previous-frame pointer*.

Method invocation stacks and the JTOC are the only two Jalapeño structures that violate the Java language requirement that arrays not contain both primitives and references. Since neither is directly accessible to users, this is not a lapse of security. However, in order to facilitate type-accurate garbage collection, Jalapeño's compilers must maintain reference maps (described later) that allow location of the object references in a stack.

Run-time subsystem. Through judicious exploitation of the MAGIC class (see Appendix A), Jalapeño's runtime subsystem provides, (mostly) in Java code, services—exception handling, dynamic type checking, dynamic class loading, interface invocation, I/O, reflection, etc.—that are conventionally implemented with native code.

Exceptions. A hardware interrupt is generated if a null pointer is dereferenced, an array index is out of bounds, an integer is divided by zero, or a thread's method-invocation stack overflows. These interrupts are caught by a small C interrupt handler that causes a Java method to be run. This method builds the appropriate exception and passes it to the deliverException method. The deliverException method is called with software-generated exceptions as well. It has two responsibilities. First, it must save in the exception object information that would allow a stack trace to be printed if one is needed. It does this by "walking" up the stack and recording the compiled method identifiers and next-instruction pointers for each stack frame. Second, it must transfer control to the appropriate "catch" block. This also involves walking the stack. For each stack frame, it locates the compiled-method object for the method body that produced the stack frame. It calls a method of this object to determine if the exception happened within an appropriate "try" block. If so, control is transferred to the corresponding catch block. If not, any locks held by the stack frame are released and it is deallocated. The next stack frame is then considered. If no catch block is found, the thread is killed.

Dynamic class loading. One of the innovative features of the Java language is its provision for loading classes during the execution of an application. When a Jalapeño compiler encounters a bytecode (putstatic or invokevirtual, for example) that refers to a class that has not been loaded, it does not load the class immediately. Rather, the compiler emits code that when *executed* first ensures that the referenced class is loaded (and resolved and instantiated) and then performs the operation. Note that when this code is generated the compiler cannot know the actual offset values (because they are not assigned until the class is loaded) of fields and methods of the class (e.g., the JTOC index assigned to a static field). The

baseline compiler (Jalapeño's compilers are described later) handles this uncertainty by emitting code that calls a run-time routine that performs any necessary class loading and then overwrites the call site with the code the baseline compiler would have produced had the class been resolved at initial code generation time. This is particularly tricky on an SMP with processors that adhere to a relaxed memory consistency model.[6]

The optimizing compiler uses an alternative approach based on an added level of indirection; when executed, the code it emits loads a value from an offset array. If the offset value is valid (nonzero), it is used to complete the operation. If the offset is invalid, the required class is loaded and, in the process, the offset array is updated to contain valid values for all of the class's methods and fields. For each dynamically linked site, the baseline compiler's approach incurs substantial overhead the first time the site is executed, but every subsequent execution incurs no overhead, while the optimizing compiler's approach incurs a small overhead on every execution of the site. However, the optimizing compiler is not normally called on a method until the method has executed several times; any dynamically linked site this compiler sees can be assumed to be very rarely executed. It is not yet clear which approach would be most appropriate for the quick compiler.

Input and output. I/O requires operating system support. To read a block from a file, an AIX stack frame is constructed and an operating system routine is called (through the C library) with an address to which to copy its result. This address is a Java array. Care is taken to prevent a copying garbage collector from moving this object until the call is complete (see Appendix A for details). So far, we have not observed a performance degradation from delaying garbage collection until the read completes. Other I/O services are handled similarly.

Reflection. Java's reflection mechanism allows run-time access to fields (given their names and types) and run-time invocation of methods (given their signatures). It is easy for Jalapeño to support reflective field access: the name is turned into an offset and the access is performed at the appropriate raw memory address. Reflective method invocation is a little harder. The address of the method body is obtained by finding the signature in a table. An artificial stack frame is constructed. (Since this stack frame does not contain any object references, it is not necessary to build a reference map for it.) The method's parameters are carefully unwrapped and loaded into registers. The method is then called. When it returns, the artificial stack frame must be disposed of, and the result wrapped and returned to the reflective call.

Thread and synchronization subsystem. Rather than mapping Java threads to operating system threads directly, Jalapeño multiplexes Java threads on *virtual processors* that are implemented as AIX pthreads. This decision was motivated by three concerns. We needed to be able to effect a rapid transition between mutation (by normal threads) and garbage collection. We wanted to implement locking without using AIX services. We want to support rapid thread switching.

Currently, Jalapeño establishes one virtual processor for each physical processor. Additional virtual processors may eventually be used to mask I/O latency. The only AIX service required by the subsystem is a periodic timer interrupt provided by the incinterval system call. Jalapeño's locking mechanisms make no system calls.

Quasi-preemption. Jalapeño's threads are neither "run-until-blocked" nor fully preemptive. Reliance on voluntary yields would not have allowed Jalapeño to make the progress guarantees required for a server environment. We felt that arbitrary preemption would have radically complicated the transition to garbage collection and the identification of object references on thread stacks. In Jalapeño, a thread can be preempted, but only at predefined *yield points*.

The compilers provide location information for object references on a thread's stack at yield points. Every method on the stack is at a *safe point* (described later). This allows compilers to optimize code (by maintaining an internal pointer, for example) between safe points that would frustrate type-accurate garbage collection if arbitrary preemption were allowed.

Locks. Concurrent execution on an SMP requires synchronization. Thread scheduling and load balancing (in particular) require atomic access to global data structures. User threads also need to synchronize access to their global data. To support both system and user synchronization, Jalapeño has three kinds of locks.

A *processor* lock is a low-level primitive used for thread scheduling (and load balancing) and to implement Jalapeño's other locking mechanisms. Pro-

cessor locks are Java objects with a single field that identifies the virtual processor that owns the lock. If the field is null, the lock is not owned. The identity of a virtual processor is maintained in a dedicated *processor* (PR) register. To acquire a processor lock for the thread it is running, a virtual processor:

- Loads the lock owner field making a CPU reservation (PowerPC lwarx)
- Checks that this field is null
- Conditionally stores the PR register into it (PowerPC stwcx)

If the stwcx succeeds, the virtual processor owns the lock. If the owner field is not null (another virtual processor owns the lock), or if the stwcx instruction fails, the virtual processor will try again (i.e., spin). A processor lock is unlocked by storing null into the owner field. Processor locks cannot be acquired recursively. Because processor locks "busy wait," they must only be held for very short intervals. A thread may not be switched while it owns a processor lock for two reasons: because it could not release the lock until it resumes execution, and because our implementation would improperly transfer ownership of the lock to the other threads that execute on the virtual processor.

Jalapeño's other locking mechanisms are based on *thin locks*:[7,8] bits in an object header are used for locking in the absence of contention; these bits identify a heavyweight lock when there is contention. Jalapeño's approach differs in two ways from the previous work. In the previous work, the heavyweight locking mechanism was an operating system service; here it is a Java object. Here, if Thread A has a thin lock on an object, Thread B can promote the lock to a *thick lock*. In the previous work, only the thread that owned a thin lock could promote it.

A bit field in the status word of an object header (see earlier discussion) is devoted to locking. One bit tells whether or not a thick lock is associated with the object. If so, the rest of this bit field is the index of this lock in a global array. This array is partitioned into virtual-processor regions to allow unsynchronized allocation of thick locks. If the thick bit is not set, the rest of the bit field is subdivided into two: the *thin-lock owner* subfield identifies the thread (if any) holding a thin lock on the object. (The sizes of the bit fields can be adjusted to support up to half a million threads.) The *recursion count* subfield encodes the number of times the owner holds the lock: unlike a processor lock, a thin lock can be recursively acquired. If the object is not locked, the entire locking bit field is zero.

To acquire a thin lock, a thread sets the thin-lock owner bit field to its identifier. This is only allowed if the locking bit field is zero. The identifier of the thread currently running on a virtual processor is kept in a dedicated *thread identifier* (TI) register. Again, lwarx and stwcx instructions are used to ensure that the thin lock is acquired atomically.

A thick lock is a Java object with six fields. The mutex field is a processor lock that synchronizes access to the thick lock. The associatedObject is a reference to the object that the thick lock currently governs. The ownerId field contains the identifier of the thread that owns the thick lock, if any. The recursionCount field records the number of times the owner has locked the lock. The enteringQueue field is a queue of threads that are contending for the lock. And, the waitingQueue field is a queue of threads awaiting notification on the associatedObject.

Conversion of a thin lock to a thick one entails:

1. Creating a thick lock
2. Acquiring its mutex
3. Loading the object's status word setting a reservation (lwarx)
4. Conditionally storing (stwcx) the appropriate value—thick-lock bit set and the index for this thick lock—into the locking bit field of the object header
5. Repeating Steps 3 and 4 until the conditional store succeeds
6. Filling in the fields of the thick lock to reflect the object's status
7. Releasing the thick lock's mutex

There are two details to be considered about locking on the PowerPC. First, the reservation (of a lwarx) could be lost for a variety of reasons other than contention, including a store to the same cache line as the word with the reservation or an operating system context switch of the virtual processor. Second, before a lock (of any kind) is released, a sync instruction must be executed to ensure that the caches in other CPUs see any changes made while the lock was held. Similarly, after a lock is acquired, an isync instruction must be executed so that no subsequent instruction executes in a stale context.

Thread scheduling. Jalapeño implements a lean thread scheduling algorithm that is designed to have

a short path length, to minimize synchronization, and to provide some support for load balancing. Thread switching (and locks) are used to implement the yield and sleep methods of java.lang.Object and the wait, notify, and notifyAll methods of java.lang.Thread as well as quasi-preemption. A thread switch consists of the following operations:

- Saving the state of the old thread
- Removing a new thread from a queue of threads waiting to execute
- Placing the old thread on some thread queue (locking the queue if necessary)
- Releasing the process lock (if any) guarding access to this queue
- Restoring the new thread's state and resuming its execution

In the processor object associated with a virtual processor there are three queues of executable threads. An idleQueue holds an idle thread that will execute whenever there is nothing else to do. A readyQueue holds other ready-to-execute threads. Only the virtual processor associated with them can access these two queues, so they need not be locked to be updated. This virtual processor is the only one that removes threads from a transferQueue. However, other virtual processors can put threads on this queue, so access to it is synchronized by a processor lock. The transferQueue is used for load balancing.

Monitors. The Java language supports the *monitor* abstraction[9] to allow user-level synchronization. Conceptually, there is a monitor associated with every Java object. However, few monitors are ever used. Typically, a thread acquires the monitor on an object by executing one of the object's synchronized methods. Only a handful of monitors are held at any one time. A thread can (recursively) acquire the same monitor multiple times, but no thread can acquire a monitor held by another. Jalapeño uses its locking mechanisms to implement this functionality.

When a thread attempts to acquire the monitor for an object, there are six cases, depending on who owns the monitor for the object, and whether the object has a thick lock associated with it:

1. Object not owned—no thick lock. The thread acquires a thin lock on the object as described previously. (This is by far the most prevalent case.)
2. Object owned by this thread—no thick lock. The thread increments the recursion-count bit field of the status word using lwarx and stwcx instructions. This synchronization is necessary since another virtual processor might simultaneously convert the thin lock to a thick one. If this bit field overflows, the thin lock is converted to a thick lock.
3. Object owned by another thread—no thick lock. This is the interesting case. Three options are available. The thread could: try again (busy wait), yield and try again (giving other threads a chance to execute), or convert the thin lock to a thick one (Case 6). We are investigating various combinations of these three options.
4. Object not owned—thick lock in place. We acquire the mutex for the thick lock, check that the lock is still associated with the object, store the thread index (TI) register in the ownerId field, and release the mutex. By the time the mutex has been acquired, it is possible that the thick lock has been unlocked and even disassociated from the object. In this extremely rare case, the thread starts over trying to acquire the monitor for the object.
5. Object owned by this thread—thick lock in place. We bump the recursionCount. Synchronization is not needed since only the thread that owns a thick lock can access its recursionCount field (or release the lock).
6. Object owned by another thread—thick lock in place. We acquire the mutex, check that the lock is still associated with the appropriate object, and yield to the enteringQueue, releasing the mutex at the same time.

We are exploring two issues associated with releasing a monitor: what to do with threads on the enteringQueue when a thick lock is unlocked, and when to disassociate a thick lock from an object.

Memory management subsystem. Of all of the Java language features, automatic garbage collection is perhaps the most useful and the most challenging to implement efficiently. There are many approaches to automatic memory management,[10,11] no one of which is clearly superior in a server environment. Jalapeño is designed to support a family of interchangeable memory managers. Currently, each manager consists of a concurrent object allocator and a stop-the-world, parallel, type-accurate garbage collector. The four major types of managers supported are: copying, noncopying, generational copying, and generational noncopying.

Concurrent object allocation. Jalapeño's memory managers partition heap memory into a *large-object space* and a *small-object space*. Each manager uses a noncopying large-object space, managed as a se-

quence of pages. Requests for large objects are satisfied on a first-fit basis. After a garbage collection event, adjacent free pages are coalesced.

To support concurrent allocation of small objects by copying managers, each virtual processor maintains a large *chunk* of local space from which objects can be allocated without requiring synchronization. (These local chunks are not logically separate from the global heap: an object allocated by one virtual processor is accessible from any virtual processor that gets a reference to it.) Allocation is performed by incrementing a space pointer by the required size and comparing the result to the limit of the local chunk. If the comparison fails (not the normal case), the allocator atomically obtains a new local chunk from the shared global space. This technique works without locking unless a new chunk is required. The cost of maintaining local chunks is that memory fragmentation is increased slightly, since each chunk may not be filled completely.

Jalapeño supports a family of memory managers for object allocation and garbage collection.

Noncopying managers divide the small-object heap into fixed-size blocks (currently 16 kilobytes). Each block is dynamically subdivided into fixed-size slots. The number of these sizes (currently 12), and their values, are build-time constants that can be tuned to fit an observed distribution of small-object sizes. When an allocator receives a request for space, it determines the size of the smallest slot that will satisfy the request, and obtains the current block for that size. To avoid locking overhead, each virtual processor maintains a local current block of each size. If the current block is full (not the normal case), it makes the next block for that size with space available the current block. If all such blocks are full (even more rare), it obtains a block from the shared pool and makes the newly obtained block current. Since the block sizes and the number of slot sizes are relatively small, the space impact of replicating the current blocks for each virtual processor is insignificant.

From mutation to collection. Each virtual processor has a collector thread associated with it. Jalapeño operates in one of two modes: either the mutators (normal threads) are running and the collection threads are idle, or the mutators are idle and the collection threads are running. Garbage collection is triggered when a mutator explicitly requests it, when a mutator makes a request for space that the allocator cannot satisfy, or when the amount of available memory drops below a predefined threshold.

Scalability requires that the transition between modes be accomplished as expeditiously as possible. During mutation, all collector threads are in a waiting state. When a collection is requested, the collector threads are notified and scheduled (normally, as the next thread to execute) on their virtual processors. When a collector thread starts executing, it disables thread switching on its virtual processor, lets the other collector threads know it has control of its virtual processor, performs some initialization, and synchronizes with the other collectors (at the first rendezvous point, described later). When each collector knows that the others are executing, the transition is complete.

Note that when all the collector threads are running, all the mutators must be at yield points. It is not necessary to redispatch any previously pending mutator thread to reach this point. When the number of mutator threads is large, this could be an important performance consideration. Since all yield points in Jalapeño are safe points, the collector threads may now proceed with collection.

After the collection has completed, the collector threads re-enable thread switching on their virtual processors and then wait for the next collection. Mutator threads start up automatically as the collector threads release their virtual processors.

Parallel garbage collection. Jalapeño's garbage collectors are designed to execute in parallel. Collector threads synchronize among themselves at the end of each of three phases. For this purpose Jalapeño provides a rendezvous mechanism whereby no thread proceeds past the rendezvous point until all have reached it.

In the *initialization* phase, a copying collector thread copies its own thread object and its virtual-processor object. This ensures that updates to these objects are made to the new copy and not to the old copy, which will be discarded after the collection.

The noncopying managers associate with each block of memory a mark array and an allocation array, with one entry in each array for each slot. During initialization, all mark array entries are set to zero. All collector threads participate in this initialization.

In the *root identification and scan* phase, all collectors behave similarly. Collector threads contend for the JTOC and for each mutator thread stack, scanning them in parallel for roots (that is, object references conceptually outside the heap), which are marked and placed on a work queue. Then the objects accessible from the work queue are marked. The marking operation is synchronized so exactly one collector marks each live object. As part of marking an object, a copying collector will copy its bits into the new space and overwrite the status word of the old copy with a *forwarding pointer* to the new copy. (One of the low-order bits of this pointer is set to indicate that the object has been forwarded.)

Roots in the JTOC are identified by examining the coindexed descriptor array that identifies the type of each entry. Roots in a thread stack are identified by analyzing the method associated with each stack frame. Specifically, the local data area will have any of the stack frame's ordinary roots; the parameter spill area may have roots for the next (called) method's stack frame; the nonvolatile register save area might contain roots from some earlier stack frame. Roots are located by examining the compiler-built reference maps that correspond to the methods on the stack and tracking which stack frames save which nonvolatile registers.

The global work queue is implemented in virtual-processor-local chunks to avoid excessive synchronization. An object removed from the work queue is *scanned* for object references. (The offsets of these references are obtained from the class object that is the first entry in the object's TIB.) For each such reference, the collector tries to mark the object. If it succeeds, it adds the object to the work queue. In the copying collectors the marking (whether it succeeds or fails) returns the new address of the referenced object.

In the *completion phase*, copying collectors simply reverse the sense of the occupied and available portions of the heap. Collector threads obtain local chunks from the now empty "nursery" in preparation for the next mutator cycle. A noncopying collector thread performs the following steps:

- If this was a minor collection by the generation collector, mark all old objects as live (identified from the current allocation arrays).
- Scan all mark arrays to find free blocks, and return them to the free block list.
- For all blocks not free, exchange mark and allocation arrays: the unmarked entries in the old mark array identify slots available for allocation.

Performance issues. We are actively investigating both noncopying and copying memory managers to understand more fully the circumstances under which each is to be preferred and to explore possibilities for hybrid solutions. (The noncopying large-object space is an example of a hybrid solution.) The major advantages of a copying memory manager lie in the speed of object allocation, and the compaction of the heap performed during collection (providing better cache performance). The major advantages of a noncopying memory manager lie in faster collection (objects are not copied), better use of available space (copying managers waste half this space), and simpler interaction between mutators and collectors. (The optimizing compiler will be able to pursue more aggressive optimizations, if it does not have to be concerned that objects might move at every safe point.) A system with a copying manager would run overall faster between collections; a system with a noncopying manager would offer smaller pause times.

A noncopying policy would greatly simplify a *concurrent* memory manager (one in which mutators and collectors run at the same time): it would eliminate the need for a read barrier and simplify the write barrier.

Compiler subsystem. Jalapeño executes Java bytecodes by compiling them to machine instructions at run time. Three different, but compatible, compilers are in use or under development. Development of Jalapeño depended upon early availability of a transparently correct compiler. This is the role of Jalapeño's *baseline* compiler. However, by construction, it does not generate high-performance target code.

To obtain high-quality machine code for methods that are observed to be computationally intensive, Jalapeño's *optimizing* compiler (described in the next section) applies traditional static compiler optimizations as well as a number of new optimizations that are specific to the dynamic Java context. The cost of running the optimizing compiler is too high for

it to be profitably employed on methods that are only infrequently executed.

Jalapeño's *quick* compiler will compile each method as it executes for the first time. It balances compile-time and run-time costs by applying a few highly effective optimizations. Register allocation is the most important of these because the PowerPC has generous (32 fixed, 32 floating) register resources and most register operations are one cycle, while storage references may require several (sometimes very many) cycles.

The quick compiler tries to limit compile time by an overall approach of minimal transformation, efficient data structures, and few passes over the source and derived data. The source bytecode is not translated to an intermediate representation. Instead, the bytecode is "decorated," with the results of analysis and optimization, in objects related to each bytecode instruction. Optimizations performed include copy propagation to eliminate temporaries introduced by the stack-based nature of Java bytecode. The quick compiler's primary register allocator uses a graph coloring algorithm.[12] Coloring is not appropriate (due to long compile time) for some methods (long one-basic-block static initializers that need many symbolic registers, for example). For such methods, the quick compiler has a simpler, faster algorithm. We will investigate heuristics to detect these cases. We also plan to add inline compilation of short methods that are final, static, or constructors and to explore local-context (peephole) optimizations.

The code produced by all three compilers must satisfy Jalapeño's calling and preemption conventions. They ensure that threads executing the methods they compile will respond in a timely manner to attempts to preempt them. Currently, explicit yield points are compiled into method prologues. Eventually, yield points will be needed on the "back edges" of loops that cannot be shown to contain other yield points.

The compilers are also responsible for maintaining tables that support exception handling and that allow the memory managers to find object references on thread stacks. (These tables are also used by Jalapeño's debugger.) When a garbage collection event takes place, each of the methods represented on the thread stack will be at a garbage collection *safe point*. Safe points are the call sites, dynamic link sites, thread yield sites, possible exception-throw sites, and allocation request sites. For any given safe point within a method body, the compiler that created the method body must be able to describe where the live references exist. A *reference map* identifies, for each safe point, the locations of object references.

We have not yet implemented a comprehensive strategy to select compilers for methods. Switching from the quick to the optimizing compiler will be done based on run-time profiling in the manner of Self.[13,14]

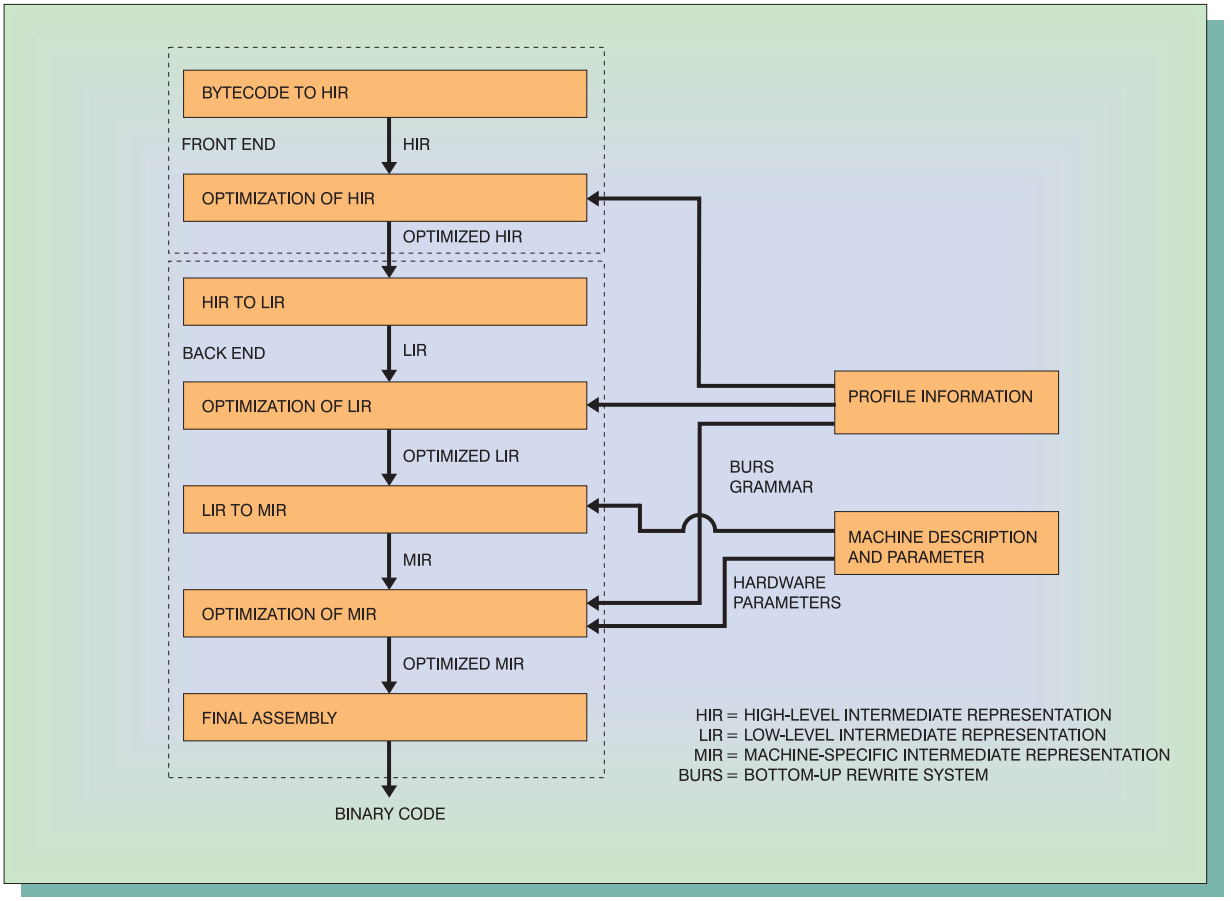
A dynamic optimizing compiler

We anticipate that the bulk of the computation on a Java application will involve only a fraction of the Java source code. Jalapeño's optimizing compiler is intended to ensure that these bytecodes are compiled efficiently. The optimizing compiler is *dynamic*: it compiles methods while an application is running. In the future, the optimizing compiler will also be *adaptive*: it will be invoked automatically on computationally intensive methods. The goal of the optimizing compiler is to generate the best possible code for the selected methods on a given compile-time budget. In addition, its optimizations must deliver significant performance improvements while correctly preserving Java's semantics for exceptions, garbage collection, and threads. Reducing the cost of synchronization and other thread primitives is especially important for achieving scalable performance on SMP servers. Finally, it should be possible to retarget the optimizing compiler to a variety of hardware platforms with minimal effort. Building a dynamic optimizing compiler that achieves these goals is a major challenge.

This section provides an overview of the Jalapeño optimizing compiler; further details are available elsewhere.[15–17] The optimizing compiler's structure is shown in Figure 4.

From bytecode to intermediate representations. The optimizing compiler begins by translating Java bytecodes to a *high-level intermediate representation (HIR)*. This is one of three register-based intermediate representations that share a common implementation. (Register-based representations provide greater flexibility for code motion and code transformation than do representations based on trees or stacks. They also allow a closer fit to the instruction sets of Jalapeño's target architectures.) Instructions of these intermediate representations are n-tuples: an *operator* and zero or more *operands*. Most operands represent symbolic registers, but they can also represent physical registers, memory locations, constants, branch targets, or types. Java's type structure

Figure 4 Jalapeño's optimizing compiler



is reflected in these intermediate representations: there are distinct operators for similar operations on different primitive types, and operands carry type information.[17] Instructions are grouped into extended basic blocks that are not terminated by method calls or by instructions that might cause an exception to be thrown. (Extra care is required when performing data flow analysis or code motion on these extended basic blocks.)[16,17] These intermediate representations also include space for the caching of such optional auxiliary information as reaching-definition[18] sets, dependence graphs, and encodings of loop-nesting structure.

The translation process discovers the extended-basic-block structure of a method, constructs an exception table for the method, and creates HIR instructions for bytecodes. It discovers and encodes type information that can be used in subsequent optimizations and that will be required for reference maps. Certain simple "on-the-fly" optimizations—copy propagation, constant propagation, register renaming for local variables, dead-code elimination, etc.—are also performed.[19] (Even though more extensive versions of these optimizations are performed in later optimization phases, it is worthwhile to perform them here because doing so reduces the size of the generated HIR and hence subsequent compile time.) In addition, suitably short final or static methods are moved in line.

Copy propagation is an example of an on-the-fly optimization performed during the translation. Java bytecodes often contain instruction sequences that perform a calculation and store the result into a local variable. A naive approach to intermediate rep-

resentation generation results in the creation of a temporary register for the result of the calculation and an additional instruction to move the value of this register into the local variable. A simple copy propagation heuristic eliminates many of these unnecessary temporary registers. When storing from a temporary into a local variable, the most recently generated instruction is inspected. If this instruction created the temporary to store its result, it is modified to write this result directly to the local variable instead.

Translation proceeds by *abstract interpretation* of the bytecodes. The types (and values if known at compile time) of local variables and the entries on the execution stack as defined by the *Java Virtual Machine Specification*[20] form the *symbolic state* of the abstract interpretation. (Because these types are not statically available from Java bytecodes, all Jalapeño's compilers must, in effect, track this symbolic state.) Abstract interpretation of a bytecode involves generating the appropriate HIR instruction(s) and updating the symbolic state.

The main loop of the translation algorithm uses a work list containing blocks of code with their starting symbolic states. Initially, this work list contains the entries for code beginning at bytecode 0 and for each of the method's exception handlers (with empty symbolic states). Code blocks are successively removed from the work list and interpreted as if they were extended basic blocks. If a branch is encountered, the block is split and pieces of it are added to the work list. (At control-flow join points, the values of stack operands may differ on different incoming edges, but the types of these operands must match.[21] An element-wise *meet* operation is used on the stack operands to update the symbolic state at these points.[19]) If a branch is forward, the piece from the beginning of the block to the branch is tentatively treated as a completed extended basic block. The pieces from the branch to its target and from the target to the end of the block are added to the work list. If the branch is backward, the piece from the branch to the end of the block is added to the work list. If the target of a backward branch is in the middle of an already-generated extended basic block, this block is split at the target point. If the stack is not empty at the target point, the block must be regenerated because its start state may be incorrect.

To minimize the number of times HIR is generated for the same bytecodes, a simple greedy algorithm selects the block with the lowest starting bytecode index for abstract interpretation. This simple heuristic relies on the fact that, except for loops, all control-flow constructs are generated in topological order, and that the control-flow graph is reducible. Fortuitously, this heuristic seems to obtain optimal extended-basic-block orderings for methods compiled with current Java source compilers.

High-level optimization. The instructions in the HIR are closely patterned after Java bytecodes, with two important differences—HIR instructions operate on symbolic register operands instead of an implicit stack, and the HIR contains separate operators to implement explicit checks for run-time exceptions (e.g., array-bounds checks). The same run-time check can often cover more than one instruction. (For example, incrementing A[i] may involve two separate array accesses, but requires only a single bounds check.) Optimization of these check instructions reduces execution time and facilitates additional optimization.

Currently, simple optimization algorithms with modest compile-time overheads are performed on the HIR. These optimizations fall into three classes:

1. *Local optimizations*. These optimizations are local to an extended basic block, e.g., common subexpression elimination, elimination of redundant exception checks, and redundant load elimination.

2. *Flow-insensitive optimizations*. To optimize across basic blocks, the Java Virtual Machine Specification assurance that "every variable in a Java program must have a value before it is used"[20] is exploited. If a variable is only defined once, then that definition reaches every use. For such variables, "def-use" chains are built, copy propagation performed, and dead code eliminated without any expensive control-flow or data-flow analyses. Additionally, the compiler performs a conservative flow-insensitive escape analysis for scalar replacement of aggregates and semantic expansion transformations of calls to standard Java library methods.[22]

   This technique catches many optimization opportunities, but other cases can only be detected by flow-sensitive algorithms.

3. *In-line expansion of method calls*. To expand a method call in line at the HIR level, the HIR for the called method is generated and patched into the HIR of the caller. A static size-based heuristic is currently used to control automatic in-line ex-

pansion of calls to static and final methods. For nonfinal virtual method calls, the optimizing compiler predicts the receiver of a virtual call to be the declared type of the object. It guards each inline virtual method with a run-time test to verify that the receiver is predicted correctly, and to default to a normal virtual method invocation if it is not. This run-time test is safe in the presence of dynamic class loading.

Since Jalapeño is written in Java, the same framework used to expand application methods in line can also be used to expand calls to run-time methods in line (notably for synchronization and for object allocation). In general, it is possible to expand calls in line all the way from the application code through Java libraries down to the Jalapeño run-time system, providing excellent opportunities for optimization.

Low-level optimization. After high-level analyses and optimizations have been performed, HIR is converted to a *low-level intermediate representation* (LIR). The LIR expands HIR instructions into operations that are specific to the Jalapeño virtual machine's object layout and parameter-passing conventions. For example, virtual method invocation is expressed as a single HIR instruction analogous to the invokevirtual bytecode. This single HIR instruction is converted into three LIR instructions that obtain the TIB pointer from an object, obtain the address of the appropriate method body from the TIB, and transfer control to the method body.

Since field and header offsets are now available as constants, new opportunities for optimization are exposed. In principle, any high-level optimization could also be performed on the LIR. However, since the LIR can be two to three times larger than the corresponding HIR, more attention needs to be paid to compile-time overhead when performing LIR optimizations. Currently, local common subexpression elimination is the only optimization performed on LIR. Since HIR and LIR share the same infrastructure, the code that performs common subexpression elimination on HIR can be reused without modification on LIR.

Also, as the last step of low-level optimization, a *dependence graph* is constructed for each extended basic block.[17] The dependence graph is used for instruction selection (see next subsection). Each node of the dependence graph is an LIR instruction, and each edge corresponds to a dependence constraint between a pair of instructions. Edges are built for true, anti, and output dependences[17] for both registers and memory. Control, synchronization, and exception dependence edges are also built. Synchronization constraints are modeled by introducing *synchronization dependence* edges between synchronization operations (monitor_enter and monitor_exit) and memory operations. These edges prevent code motion of memory operations across synchronization points. Java exception semantics[20] is modeled by *exception dependence edges* connecting different exception points in an extended basic block. Exception dependence edges are also added between these exception points and register write operations of local variables that are "live" in exception handler blocks, if there are any in the method. This precise modeling of dependence constraints enables aggressive code reordering in the next optimization phase.

Instruction selection and machine-specific optimization. After low-level optimization, the LIR is converted to *machine-specific intermediate representation* (MIR). The current MIR reflects the PowerPC architecture. (Additional sets of MIR instructions can be introduced if Jalapeño is ported to different architectures.) The dependence graphs for the extended basic blocks of a method are partitioned into trees. These are fed to a *bottom-up rewriting system* (BURS),[23] which produces the MIR. Then symbolic registers are mapped to physical registers. A *prologue* is added at the beginning, and an *epilogue* at the end, of each method. Finally, executable code is emitted.

BURS is a code-generator generator, analogous to scanner and parser generators. Instruction selection for a desired target architecture is specified by a *tree grammar*. Each rule in the tree grammar has an associated cost (reflecting the size of instructions generated and their expected cycle times) and code-generation action. The tree grammar is processed to generate a set of tables that drive the instruction selection phase at compile time.

There are two key advantages of using BURS technology for instruction selection. First, the tree-pattern matching performed at compile time uses dynamic programming to find a least-cost parse (with respect to the costs specified in the tree grammar) for any input tree. Second, the cost of building the BURS infrastructure can be amortized over several target architectures. The architecture-specific component is relatively short; Jalapeño's PowerPC tree grammar is about 300 rules.

The tree-pattern matching in BURS was originally developed for code generation from tree-based intermediate representations, usually in the absence of global optimizations. Previous approaches to partitioning directed acyclic graphs for tree-pattern matching considered only graphs containing register-true-dependence edges.[24] Our approach is more general because it considers partitioning in the presence of both register and nonregister dependences. The legality constraints for this partitioning are nontrivial.[25]

The optimizing compiler's front end is independent of Jalapeño's object layout and calling conventions.

After the MIR is constructed, live variable analysis is performed to determine the live ranges of symbolic registers and the stack variables that hold object references at garbage-collection-safe points. The standard live variable analysis[18] has been modified to handle the extended basic blocks of the *factored control flow graph* as described by Choi, et al.[16]

Next, the optimizing compiler employs the *linear scan* global register-allocation algorithm[26] to assign physical machine registers to symbolic MIR registers. This algorithm is not based on graph coloring, but greedily allocates physical to symbolic registers in a single linear time scan of the symbolic registers' live ranges. This algorithm is several times faster than graph coloring algorithms and results in code that is almost as efficient. More sophisticated (and more costly) register allocation algorithms will eventually be used at higher levels of optimization (see next subsection). (The irony of currently using a more expensive algorithm in the quick compiler than in the optimizing compiler is not lost on the authors.)

A method prologue allocates a stack frame, saves any nonvolatile registers needed by the method, and checks to see if a yield has been requested. The epilogue restores any saved registers and deallocates the stack frame. If the method is synchronized, the prologue locks, and the epilogue unlocks, the indicated object.

The optimizing compiler then emits binary executable code into the array of ints that is the method body. This assembly phase also finalizes the exception table and the reference map of the instruction array by converting intermediate-instruction offsets into machine-code offsets.

Levels of optimization. The optimizing compiler can operate at different levels of optimization. Each level encompasses all the optimizations at the previous levels and some additional ones. *Level 1* contains exactly the optimizations described above. (Primarily for debugging purposes, there is a *Level 0*, which is like Level 1 without any high-level or low-level optimizations.) Two levels of more aggressive optimization are planned.

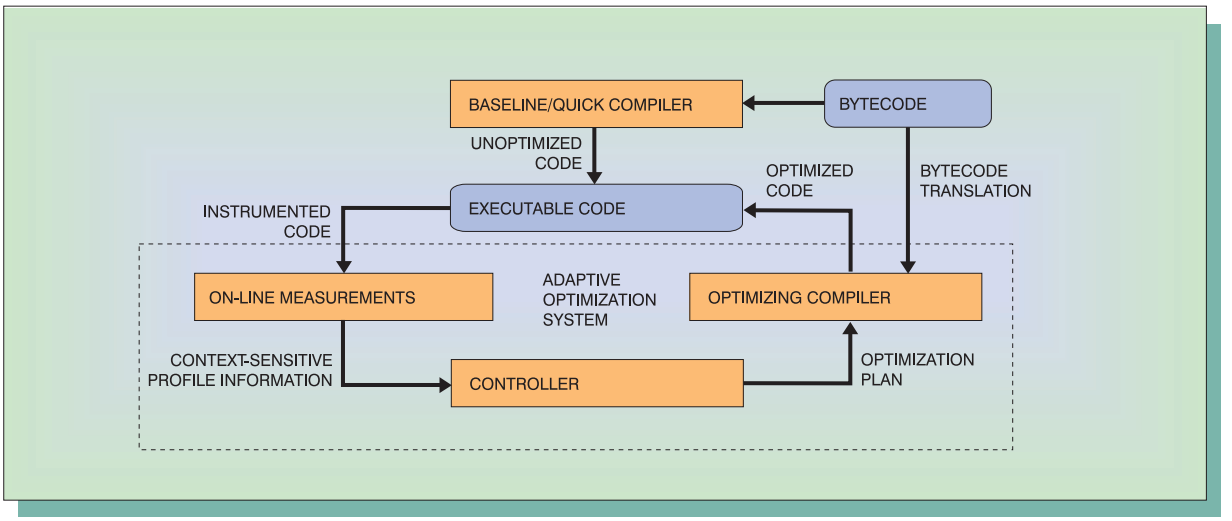
Level 2 optimizations will include code specialization, intraprocedural flow-sensitive optimizations based on *static single assignment* (SSA) form (both scalar[27] and array[28]) sophisticated register allocation, and instruction scheduling. Instruction scheduling is currently being implemented. It uses an MIR dependence graph built with the same code that builds the LIR dependence graph used by BURS.

Level 3 optimizations will include interprocedural analysis and optimizations. Currently, interprocedural escape analysis[29] and interprocedural optimization of register saves and restores[15] are being implemented.

Modalities of operation. The optimizing compiler's front end (translation to HIR and high-level optimizations) is independent of Jalapeño's object layout and calling conventions. This front end is being used in a bytecode optimization project.[30]

The intended mode of operation for the optimizing compiler is as a component of an adaptive Jvm. Figure 5 shows the overall design of such a virtual machine. The optimizing compiler is the key constituent of Jalapeño's adaptive optimization system, which will also include on-line measurement and controller subsystems currently under development. The on-line measurement subsystem will monitor the performance of individual methods using both software sampling and profiling techniques and information from a hardware performance monitor. The controller subsystem will be invoked when the on-line measurement subsystem detects that a certain perfor-

Figure 5 The optimizing compiler in an adaptive Jvm

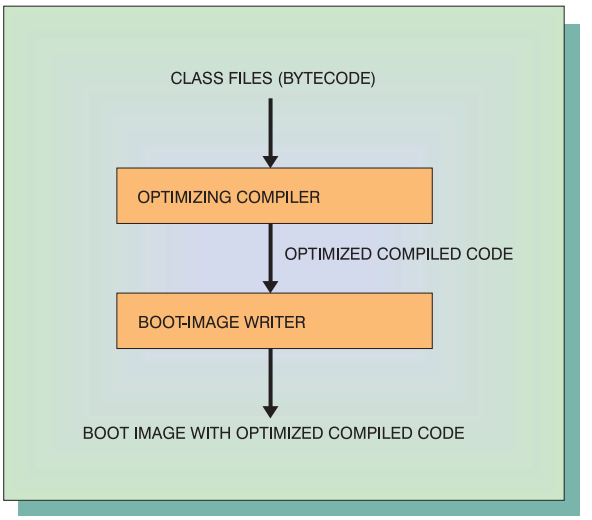


mance threshold has been reached. The controller will use the profiling information to build an "optimization plan" that describes which methods should be compiled and with what optimization levels. The optimizing compiler will then be invoked to compile methods in accordance with the optimization plan. The on-line measurement subsystem can continue monitoring individual methods, including those already optimized, to trigger further optimization passes as needed.

In addition to the dynamic compilation mode described above, the optimizing compiler can be used as a static compiler as shown in Figure 6. In this mode, the optimized code generated by the optimizing compiler is stored in the boot image (see Appendix B). The optimized compilation is performed off line, prior to execution of the Jalapeño virtual machine. (Eventually, we hope to be able to combine both modes. An application would run for a while. The adaptive optimization system would optimize the Jvm for that application. Finally, this optimized Jvm would get written out as a boot image specialized for the application.)

The optimizing compiler can also be used as a JIT compiler compiling all methods the first time they are executed. When benchmarking the performance of the optimizing compiler, it is used both as a static boot-image compiler (for Jvm code in the boot image) and as a JIT compiler (for the benchmark code and any remaining JVM code).

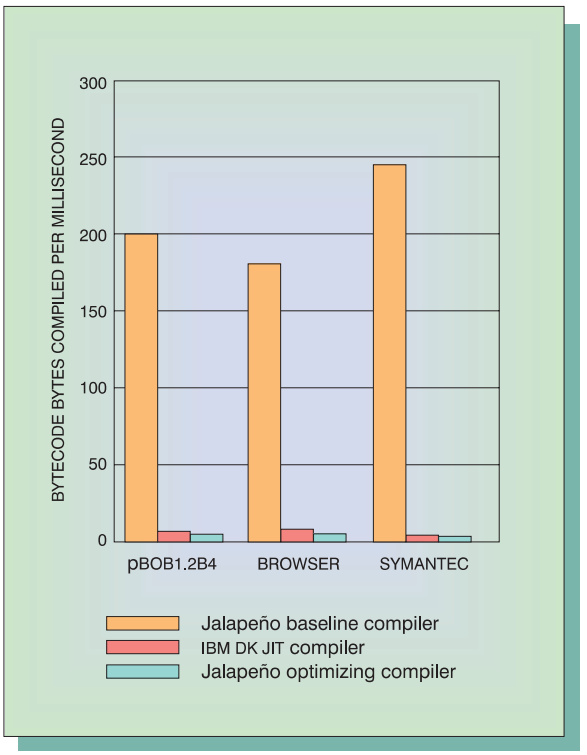
Figure 6 The optimizing compiler as a static compiler



Current status

The core functionality required to implement all Java language features is all but complete. Some of the more esoteric thread function—suspend, resume,

Figure 7 Compilation speeds



timed wait, etc.—have yet to be implemented. The load-balancing algorithm is rudimentary. Support for finalization, weak references, and class verification is not yet in place. The quick compiler is nearing completion. The basic framework of the optimizing compiler and some of its Level 1 optimizations are now up and running. More advanced optimizations are being developed. The on-line measurement and controller subsystems are in the design stage.

Jalapeño's support for Java library code is limited by the fact that Jalapeño is written in Java code. Jalapeño can handle library methods written in Java code, but native methods must be rewritten. Implementing the Java Native Interface (JNI) will allow Jalapeño to call native methods written to that interface, but some native methods are not. JNI is an especially difficult issue to address because it is a C language interface to a virtual machine that, in the case of Jalapeño, is not written in C. We do not yet understand the performance or implementation issues that will arise when we attempt to provide JNI services in Jalapeño.

The Jalapeño project is in transition. The initial function is mostly in place. Many of Jalapeño's mechanisms are still rudimentary. It is time to measure performance, identify bottlenecks, and replace them with more efficient implementations. Some of the "low-hanging fruit" has already been picked: uncontended lock acquisitions have been moved in line, for example. However, the performance measurements of baseline compiled code were so inconclusive that we have been reluctant to trust our measurements until the optimizing compiler was available.

There are also bugs, of both function and performance, to be isolated, identified, and fixed.

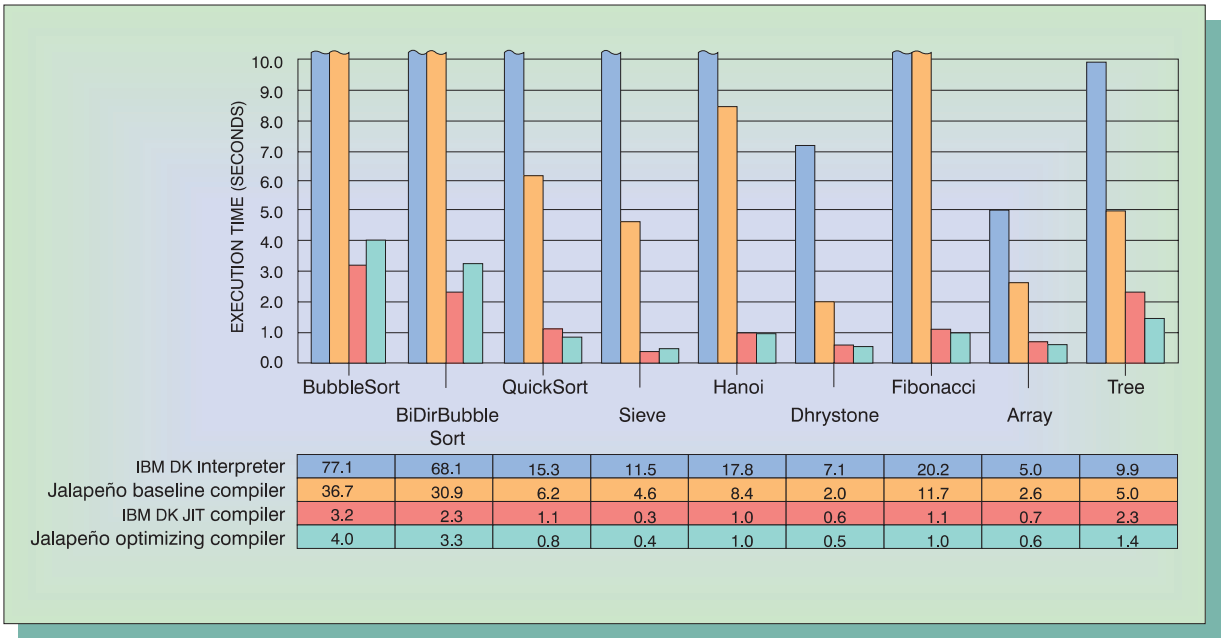
In trying to assess the current performance of Jalapeño, it is useful to make comparisons with the Jvm in the IBM Developer Kit (DK) for AIX, Java Technology Edition, Version 1.1.8 that uses the JIT compiler developed in IBM Tokyo.[31] It should be noted that while Jalapeño has the luxury of being targeted to SMP servers, the IBM Jvm must accommodate all PowerPC computers running AIX. The reader should also keep in mind that the performance figures quoted here represent a snapshot in time: both Jalapeño and the IBM Jvm are constantly being improved.

Performance figures are given for Jalapeño's baseline and optimizing compilers. In both cases, the boot image has been compiled with the optimizing compiler, and the indicated compiler is used primarily for the indicated application (and for any dynamically linked classes of the Jvm). The optimizing compiler figures reflect currently implemented Level 1 optimizations. A nongenerational copying memory manager is used in both cases.

Figure 7 compares the time spent in compilation by Jalapeño's baseline and optimizing compilers (written in the Java language and optimized by Jalapeño's optimizing compiler) and the IBM DK JIT compiler (implemented in native code). The baseline compiler is the clear winner, running 30 to 45 times faster than the JIT compiler. The optimizing compiler is nearly as fast as the JIT compiler, but not quite.

Figure 8 compares the performance of code produced by the three compilers to interpreted code by the IBM DK without a JIT compiler on microbenchmarks from Symantec.[32] (The graph has been truncated to facilitate comparison of the performance of Jalapeño's optimizing compiler and the IBM DK JIT compiler.) The baseline compiled code is consistently

Figure 8 Symantec microbenchmarks (166 MHz PowerPC 604e, AIX 4.3, copying garbage collector, Level 1 optimization); each measurement is an average of 10 runs



	BubbleSort	BiDirBubble Sort	QuickSort	Sieve	Hanoi	Dhrystone	Fibonacci	Array	Tree
IBM DK Interpreter	77.1	68.1	15.3	11.5	17.8	7.1	20.2	5.0	9.9
Jalapeño baseline compiler	36.7	30.9	6.2	4.6	8.4	2.0	11.7	2.6	5.0
IBM DK JIT compiler	3.2	2.3	1.1	0.3	1.0	0.6	1.1	0.7	2.3
Jalapeño optimizing compiler	4.0	3.3	0.8	0.4	1.0	0.5	1.0	0.6	1.4

twice as fast as interpreted code. The IBM JIT-compiled code is much better: between four and 40 times faster than the interpreted code. Jalapeño's optimizing compiler is roughly competitive with the JIT compiler.

Figure 9 makes the same comparison on the SPECjvm98 benchmarks[33] run on the medium (10 percent) problem size.[34] Again the baseline compiler is usually about twice as good as the interpreter. Again the JIT compiler is much better. Again the optimizing compiler is usually competitive with the JIT compiler.

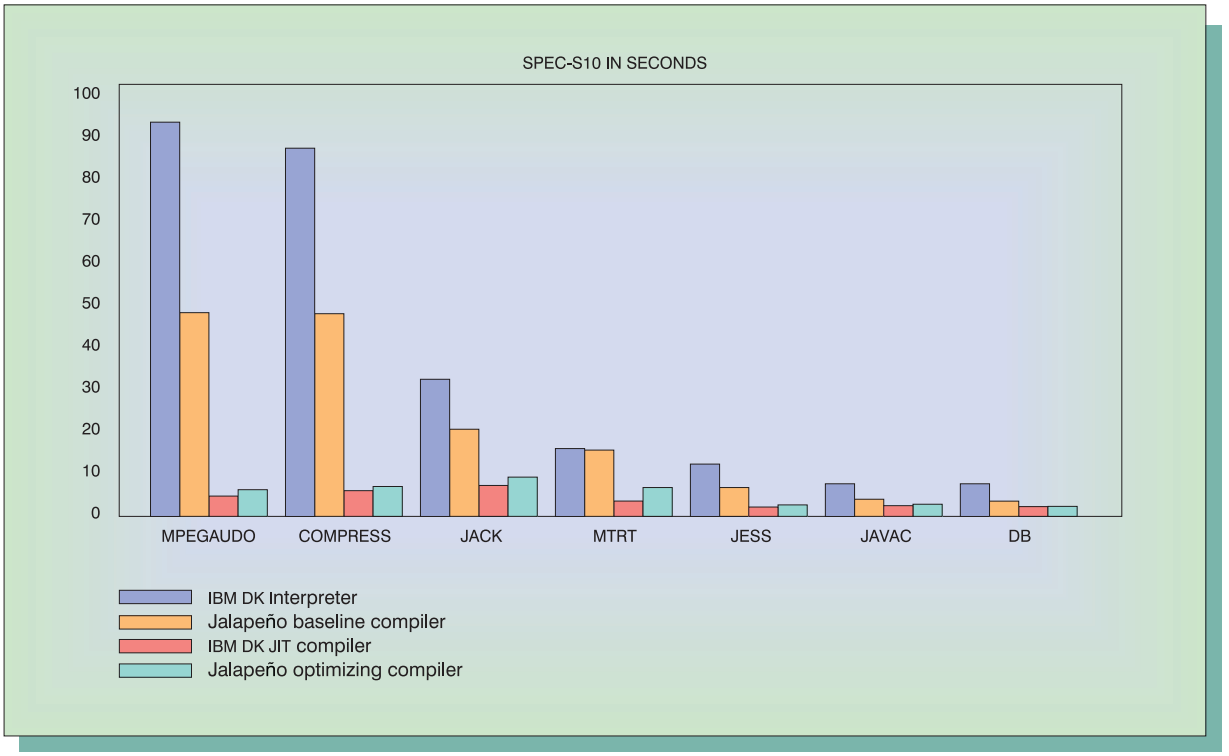
Figure 10 shows the performance of the Jalapeño optimizing compiler running 12 virtual processors on a 12-way SMP (with 262 MHz PowerPC S7a processors running AIX 4.3) using the portable business object benchmark (pBOB v 2.0a). This benchmark (see Baylor et al.[35] in this issue for details), modeled after the TPC-C** specification, mimics the business logic in a transactional workload. Performance improves almost linearly to 10 warehouses, peaks at 13, and then degrades very slowly. This shows that, on this benchmark at least, Jalapeño scales very well.

Related work

Implementing a Java virtual machine and its related subsystems (including the optimizing compiler) in Java code presents several challenges. Taivalsaari[36] also describes a "Java in Java" Jvm implementation designed to examine the feasibility of a high quality virtual machine written in Java. One drawback of this approach is that it runs on another Jvm, which adds performance overhead because of the two-level interpretation process. The Rivet Jvm[37] from MIT (Massachusetts Institute of Technology) also runs on top of another Jvm. Our approach avoids the need for another Jvm by bootstrapping the system (see Appendix B). The Jvm of IBM's VisualAge* for Java[38] is written in Smalltalk. Other Jvms[39–43] are written in native code.

Perhaps the most exciting conventional Jvm is HotSpot.[40] The object models of HotSpot and Jalapeño are somewhat similar: objects are referenced directly (rather than through handles) and objects have a two-word header. In both models, information about the object's class is available through a reference in the object header.

Figure 9 SPECjvm98 benchmarks (medium size)



HotSpot initially interprets bytecodes, compiling (and moving in line) frequently called methods. Jalapeño's quick compiler will play a role similar to HotSpot's interpreter. All else being equal, this should give a start-up advantage to HotSpot and a performance advantage to Jalapeño. We do not expect either advantage to be dramatic, but this remains to be seen. If unoptimized Jalapeño code performs better than interpreted HotSpot code, this will allow the Jalapeño optimizing compiler to focus more resources on the code that it optimizes. Implementing Jalapeño in Java code allows the optimizing compiler to move in line and optimize frequently called run-time services that HotSpot accesses through calls to native methods (heavily optimized C routines).

HotSpot implements Java threads as host operating system threads. These threads are fully preemptive. Jalapeño schedules its own quasi-preemptive threads. We expect that this will allow support for more threads, lighter-weight synchronization, and smoother transition from normal operation to garbage collection (especially in the presence of a large number of threads). HotSpot's per-thread method activation stacks conform to host operating system calling conventions. This should give Jalapeño a minor space and performance advantage (although Jalapeño will take a performance hit when it *does* call C code).

Both HotSpot and Jalapeño support type-accurate garbage collection. Jalapeño supports a family of memory managers. None of Jalapeño's collectors is as sophisticated as HotSpot's, but on an SMP Jalapeño's collectors run in parallel using all available CPUs. HotSpot uses a generational scheme with "mark-and-compact" for major collections. To minimize pause times, HotSpot can use an incremental "train" collector.[44] This collector makes frequent short collections. Note that this will exacerbate any transition-to-collection delays.

We do not have information on HotSpot's locking mechanisms.

Squeak[45] is a Smalltalk virtual machine that is written in Smalltalk. It produces a production version by translating the virtual machine to C for compi-

lation and linking. The translator is also written in Smalltalk.

Dynamic compilation (called dynamic translation or just-in-time compilation) has been a key ingredient in a number of previous implementations of object-oriented languages. Deutsch and Schiffman's high-performance implementation of Smalltalk-80 dynamically translated Smalltalk bytecodes to native code;[46] their compiler was quite similar to Jalapeño's baseline compiler. Implementations of the Self language also relied on dynamic compilation to achieve high performance.[47] Self compilers utilized register-based intermediate representations that are roughly equivalent to the one used by Jalapeño's optimizing compiler. Recently, a number of just-in-time compilers have been developed for the Java language.[31,48] Some of these compilers translate bytecodes to a three-address code, perform simple optimizations and register allocation, and then generate target machine code.

DAISY[49] is a VLIW (very long instruction word) emulator that performs "on-the-fly" translation of different architecture instruction sets, including Java bytecodes, to a VLIW architecture. It uses a VLIW tree-like representation for instruction scheduling and register allocation.

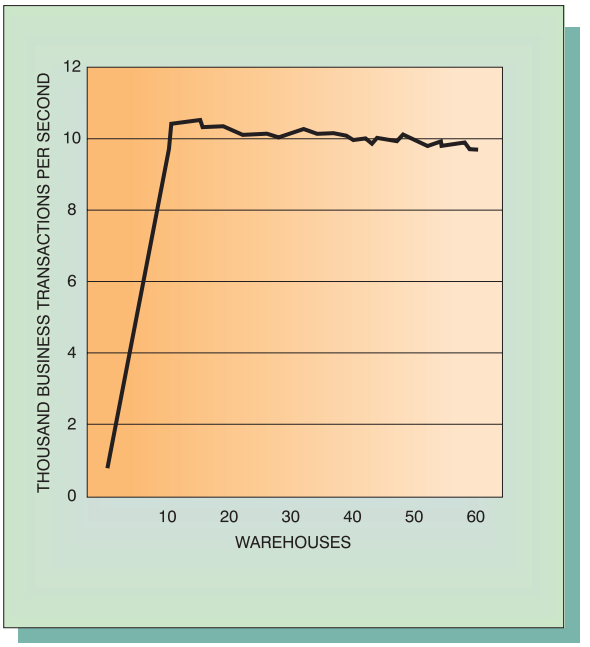
A number of previous systems have utilized more specialized forms of dynamic compilation to selectively optimize program hot spots by exploiting "run-time constants."[50–53] In general, these systems emphasize extremely fast dynamic compilation, often performing extensive off-line precomputations to avoid constructing any explicit representation of the program fragment being compiled.

A large collection of work addresses optimizations specific to object-oriented languages, such as class analysis, both intraprocedural[54] and interprocedural,[55] class hierarchy analysis and optimizations,[56,57] receiver class prediction,[46,58,59] method specialization,[56] and call graph construction.[55] Other optimizations relevant to Java compilation include bounds check elimination[60] and semantic expansion.[22]

Conclusions

Jalapeño is a virtual machine for Java servers written in the Java programming language. Run-time services, conventionally supported with native methods, are implemented primarily in Java code.

Figure 10 Jalapeño performance for the pBOB v 2.0a benchmark on a 12-way SMP (1–60 threads, 12 virtual processors)



Jalapeño's object layout supports single-instruction field access, three-instruction access to array elements, hardware null-pointer checks, and four-instruction virtual-method dispatches. Fast access to static fields and methods through a global JTOC array is also achieved.

Jalapeño's threads are multiplexed by virtual processors. Thread switching is quasi-preemptive. Three different locking mechanisms provide light-weight synchronization without operating system support.

Jalapeño's memory management subsystem supports a family of memory managers, each consisting of a concurrent object allocator and a parallel, type-accurate, stop-the-world garbage collector. Generational and nongenerational, copying and noncopying collectors are supported. Incremental and concurrent collectors are being investigated.

Jalapeño's three interoperable compilers provide different levels of dynamic optimization, ensure timely thread preemption, and produce tables that support exception handling, location of references in stacks, and debugging.

Jalapeño's optimizing compiler produces high-quality code for methods that have been identified as frequently executed or computationally intensive. Methods to be recompiled will be selected dynamically based on run-time profiling.

We have established the feasibility of building a virtual machine for Java servers in the Java language. We have not yet demonstrated that such a virtual machine can achieve and sustain world-class performance. We are working on it.

Appendix A: MAGIC

To allocate an object, Jalapeño's memory managers must access raw memory to obtain a piece of available space of the required size. They "walk" the thread stacks to identify object references in the stack frames. A copying manager accesses object headers to mark objects during garbage collection and accesses raw memory to copy an object. Exception handling requires an unstructured transfer of control to the appropriate catch block ("go to" is forbidden in the Java language). Static data and methods are accessed through a dedicated machine register that cannot itself be accessed from Java instructions. Input and output require access to operating system services unknown to the Java language. Thread switching depends on receiving periodic interrupts from the operating system. Jalapeño's locking mechanisms are implemented using PowerPC instructions that cannot be expressed as Java bytecodes. None of these operations can be performed without breaching Java's programming model.

To implement Jalapeño in Java code, it is necessary to augment Java's functionality to include capabilities conventionally required by native methods:

- To invoke operating system services
- To use architecture-specific machine instructions
- To access machine registers and memory
- To coerce object references to raw addresses and *vice versa*
- To transfer execution to an arbitrary address

These capabilities must be granted to Jalapeño, but Jalapeño must prevent them from becoming available to user applications.

Jalapeño's compilers enable such transgressions with the help of a special MAGIC class. The methods of this class correspond to the extra-Java operations Jalapeño must be able to perform. The bodies of these methods are empty. Java's source compilers can compile them. However, Jalapeño's compilers ignore the resulting bytecodes. Rather, they recognize the name of the MAGIC class and insert the necessary machine code in line. To make sure that user code does not evade Java's restrictions, Jalapeño's compilers will verify, when they encounter a call to a MAGIC method, that the method they are compiling is an authorized part of the Jvm.

Code that needs to exploit the MAGIC class must do so with extreme caution. The rules that are being circumvented are there for a reason. Certain operations require great care. Computing with raw addresses is particularly delicate. The MAGIC method objectAsAddress transmutes an object reference into a raw address (an int). This functionality is needed, for instance, to perform dynamic linking. It is, however, problematic. Jalapeño's copying memory managers update object references when they move the referenced object, but raw addresses are not updated. Care must be taken to avoid garbage collection when computing with raw addresses lest a copying collector invalidate them. This is prevented by calling a method that disables garbage collection.

A thread that has disabled garbage collection cannot try to create an object, because the system would hang if there were insufficient memory. (Note that other threads are free to request memory. If it is unavailable, these threads are delayed and a collection will be initiated as soon as garbage collection is reenabled.) There are subtle implications of this restriction. Classes cannot be loaded, since objects are created during class loading. This means dynamic linking must be avoided. Type casts (and stores into object arrays) cannot be allowed either, since these might also entail class loading. Similarly, if the thread were to try to enter a monitor on a shared object currently owned by a thread waiting for garbage collection, the system would be in deadlock. Thus, a thread must operate in a tightly restricted subset of Java capability when computing with raw addresses.

It would also be somewhat problematic for a thread to yield (explicitly or implicitly) while its garbage collection is disabled. Such a yield might arbitrarily delay needed garbage collection. Implicit thread switching is postponed (and explicit thread switching prohibited) while a thread's garbage collection is disabled.

There are approximately 650 Java classes in the Jalapeño system, of which approximately 110 access the

MAGIC class. Of these only 12 classes need to disable garbage collection.

Appendix B: Getting started

A fairly substantial set of services—a class loader, an object allocator, a compiler—must exist before a Jvm can load all remaining services required for normal operation. The initial services for a Jvm written in native code, or a Jvm that runs on top of another Jvm, are available from an underlying run-time routine. Jalapeño is not written in native code and it has no underlying run-time routines. Therefore, we assemble the essential core services into an executable *boot image* prior to running the Jvm. This boot image is a snapshot of a Jalapeño virtual machine written into a file. Later, this file is loaded into memory and executed.

The boot image is created by a Java program called a *boot-image writer*. It constructs a mock-up of a running Jalapeño virtual machine and then packages it into a boot image. The boot-image writer is an ordinary Java program and it can run on any Jvm. The Jvm that runs the boot-image writer will be called the *source* Jvm, and the resulting Jalapeño virtual machine, the *target* Jvm.

The boot-image writer resembles a cross compiler and linker: it compiles bytecodes to machine code and rewrites machine addresses to bind program components into a runnable image. However, since Jalapeño's compilers, class loaders, and run-time data structures are all in Java code, unlike most compilers, it must also bind "live" objects into the boot image.

The boot-image writer instantiates, in the source Jvm, Java objects that represent the target Jvm. Then it uses Java's built-in reflection facility to translate these mock-up objects from the object model of the source Jvm to Jalapeño's object model. This self-referencing aspect of the boot-image writer makes it relatively simple—it is really just an object model translator.

Since Jalapeño is a Java program, each of its components is a Java object and the boot-image writer can construct the mock-up by executing special init methods in each of Jalapeño's major subsystems. A customized class loader makes sure that any classes needed to execute this code are loaded into the mock-up as well as into the source Jvm. As a class is loaded, its methods are compiled (by a Jalapeño compiler running in the source Jvm) and included in the mock-up.

This strategy of loading classes into both the source Jvm and its mock-up of the target Jvm requires a complete class list to succeed. If, when Jalapeño starts running, a method of the core run-time environment references any class not in the boot image, an endless recursion results: the run-time environment needs to load part of itself in order to load part of itself . . . and so on.

The problem of determining the minimal set of classes needed in the mock-up to prevent this was solved using a combination of careful planning and trial and error. All of Jalapeño's core classes were named with a VM_ prefix. These are the classes needed to provide enough machinery to allow the virtual machine to perform compilation, memory management, and dynamic class loading. The special prefix is recognized by Jalapeño's compilers and used to suppress normal dynamic linking rules: they never generate dynamic linking code between methods whose classes have this prefix. The core classes were also carefully written to avoid unnecessary use of Java library classes. The fundamental classes—java.lang.Object, java.lang.Class, java.lang.String, and a few I/O classes—were unavoidable exceptions. Together, the VM_ classes and fundamental Java classes formed a starting set of classes that we thought needed to appear in the boot image.

A small number of additional dependencies (for example, Integer, Float, Double, and various array and exception classes) were then identified by trial and error. We built a boot image and attempted to execute it. If it crashed trying to (recursively) load class X, then we added X to the list of classes written into the boot image and repeated the exercise. This process converged with a small number of retries and did not prove to be a maintenance problem once the implementation of the core VM_ classes stabilized.

When the mock-up is complete, it is transformed into a boot image. This involves finding all the objects in the mock-up, converting them to Jalapeño's object format, and storing them in a *boot-image* array. All components of a running Jalapeño virtual machine can be reached from a single JTOC array (see section on static fields and methods). In the mockup, the JTOC is encoded by three parallel arrays: an array of ints (for primitive values), an array of Object instances (for references), and a Boolean array to discriminate between the two. The structure rooted in the

JTOC array is walked recursively and the values, both reference and primitive, encountered are translated into the boot-image array. Since the Type Information Block (see section on object headers) for each loaded class is referenced from the JTOC, all necessary compiled method bodies will be included in the boot image.

The translation process uses reflection. The boot-image writer obtains the java.lang.Class object for each object in the mock-up and iterates over the fields returned by the getFields method. For each field, it extracts the field value from the source object and extracts the target field offset from Jalapeño's class description for the object. Then, it writes the value at that offset from the index of the object in the boot image. When object references are encountered, we cannot use any value from the mock-up. The references in the mock-up are converted to boot-image addresses using a hash table maintained as boot-image space is allocated. (An array containing the addresses of all references in the boot image can be included in the boot image to support relocation of the image at boot time.)

Overall, the boot-image writer copies Java objects, field by field, from the mock-up into the boot image, simultaneously translating from the source Jvm's to the target Jvm's object model. Relying on Java's reflection capability, we ran into one inconvenience: Sun's Java Development Kit, v 1.1.4 did not permit reflective access to private fields. This is not a problem in the Java 2 Software Development Kit, which allows such access. We solved the problem in the earlier version by preprocessing the class files, turning the private bits off.

In addition to the objects reachable from the JTOC array, two other objects are needed in the boot image: an initial thread object containing an empty stack ready to run the first instruction of the boot( ) method when Jalapeño starts up and a "boot record" to interface the boot image with the boot-image runner (described next). This boot record contains the start, end, and last-used addresses in the image, four register values used to start Jalapeño, the address of the boot( ) method, and the addresses for AIX's system calls. When these values are stored in the boot-image array, it is written to disk.

A short program called a *boot-image runner* starts Jalapeño running. It reads the boot image into memory, sets the four registers to the indicated values, and branches to the boot( ) method. The boot-image runner is written in C (with a little assembler to set the registers and perform the final branch), not Java code, so *it* does not require a Jvm to run on.

When the boot( ) method starts executing, the virtual machine is in a fragile state: it can run a single thread of machine instructions, but it has not yet created the external operating system resources it needs to support its own execution. These operating system resources cannot be created by the boot-image writer, because they refer to external state that will not exist until the boot image is executed. Thus, Jalapeño must perform additional initialization.

At boot time, the virtual machine initializes hardware-specific addresses (for example, it will eventually establish a hardware guard page on its own stack), opens files corresponding to the Java library's System.in, System.out, and System.error stream objects, parses command line arguments, and creates a System.Properties object corresponding to the current execution environment. Then, the multithreading subsystem is initialized by creating operating system threads to serve as the virtual processors upon which Java threads are multiplexed. Finally, timer interrupts are enabled to support thread preemption and a Java thread is spawned to run the application program specified on the command line.

Jalapeño runs until the last (nondaemon) Java thread terminates or System.exit( ) is called.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc. or Transaction Processing Performance Council.

Cited references and notes

1. C. May, E. Silha, R. Simpson, and H. Warren, *The PowerPC Architecture*, Morgan Kaufmann Publishers, Inc., San Francisco, CA (1994).
2. *IBM AIX Version V4.3 Technical References*, SBOF-1878-00, IBM Corporation (1998).
3. B. Alpern, C. R. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, T. Ngo, M. Mergen, J. C. Shepherd, and S. E. Smith, "Implementation of Jalapeño in Java," *Proceedings, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Denver, CO (November 1–5, 1999), pp. 314–324.
4. AIX pthreads conform to the POSIX (Portable Operating System Interface for UNIX®) standard.
5. In AIX, it is at least theoretically possible for another process to cause a shared system library to get loaded into very high memory. This remote possibility is not a concern in a research project, but would need to be addressed by a commercial Jvm. It would be sufficient to forbid read and write

access to the last page of addressable memory. (Accesses to some of the fields of objects bigger than a page could be checked explicitly without having a major impact on performance.)
6. B. Alpern, M. Charney, J.-D. Choi, A. Cocchi, and D. Lieber, "Dynamic Linking on a Shared-Memory Microprocessor," *Proceedings, International Conference on Parallel Architectures and Compilation Technologies,* Newport Beach, CA (October 12–16, 1999), pp. 177–182.
7. D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano, "Thin Locks: Featherweight Synchronization for Java," *Proceedings, SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada (June 17–19, 1998), pp. 258–268.
8. T. Onodera and K. Kawachiya, "A Study of Locking Objects with Bimodal Fields," *Proceedings, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Denver, CO (November 1–5, 1999), pp. 223–227.
9. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM* **17**, No. 10, 549–557 (October 1974).
10. R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons, Inc., New York (1996).
11. All papers in *Proceedings, International Symposium on Memory Management*, *ACM Special Interest Group on Memory Management*, Vancouver, BC (October 17–19, 1998).
12. G. J. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein, "Register Allocation via Coloring," *Computer Languages* **6**, 47–57 (January 1981).
13. D. Ungar and R. B. Smith, "Self: The Power of Simplicity," *Proceedings, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Orlando, FL (October 4–8, 1987), pp. 227–242.
14. C. Chambers, D. Ungar, and E. Lee, "An Efficient Implementation of Self—A Dynamically-Typed Object-Oriented Language Based on Prototypes," *Proceedings, OOPSLA '89* (October 1989), pp. 49–70. Published as *ACM SIGPLAN Notices* **24**, No. 10.
15. M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The Jalapeño Dynamic Optimizing Compiler for Java," *Proceedings, ACM Java Grande Conference*, San Francisco, CA (June 12–14, 1999).
16. J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, "Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs," ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Toulouse, France (September 6, 1999), pp. 21–31.
17. C. Chambers, I. Pechtchanski, V. Sarkar, M. J. Serrano, and H. Srinivasan, "Dependence Analysis for Java," *Proceedings, 12th International Workshop on Languages and Compilers for Parallel Computing*, San Diego, CA (August 4–6, 1999).
18. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Co., Reading, MA (1986).
19. J. Whaley, *Dynamic Optimization Through the Use of Automatic Runtime Specialization*, M. Eng. thesis, Massachusetts Institute of Technology, Cambridge, MA (May 1999).
20. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley Publishing Co., Reading, MA (1996).
21. K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley Publishing Co., Reading, MA (1996).
22. P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta, "Efficient Support for Complex Numbers in Java," *Proceedings, ACM Java Grande Conference*, San Francisco, CA (June 12–14, 1999).
23. R. R. Henry, C. W. Fraser, and T. A. Proebsting, "Burg—Fast Optimal Instruction Selection and Tree Parsing," *Proceedings, SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA (June 17–19, 1992).
24. M. A. Ertl, "Optimal Code Selection in DAGs," *Proceedings, 26th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, San Antonio, TX (January 20–22, 1999).
25. V. Sarkar, M. J. Serrano, and B. B. Simons, "Retargeting Optimized Code by Matching Tree Patterns in Directed Acyclic Graphs," patent application (December 1998).
26. M. Poletto and V. Sarkar, "Linear Scan Register Allocation," *ACM TOPLAS* (July 1999).
27. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Transactions on Programming Languages and Systems* **13**, No. 4, 451–490 (October 1991).
28. K. Knobe and V. Sarkar, "Conditional Constant Propagation of Scalar and Array References Using Array SSA Form," G. Levi, Editor, *Lecture Notes in Computer Science, 1503*, Springer-Verlag, New York (1998), pp. 33–56.
29. J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape Analysis for Java," *Proceedings, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Denver, CO (November 1–5, 1999), pp. 1–19.
30. F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter, "Practical Experience with an Application Extractor for Java," *Proceedings, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Denver, CO (November 1–5, 1999), pp. 292–305.
31. T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, "Overview of the IBM Java Just-in-Time Compiler," *IBM Systems Journal* **39** No. 1, 175–193 (2000, this issue).
32. Just-In-Time Compilation (see http://www.symantec.com/cafe/analysis1.html#jitcomp).
33. The Standard Performance Evaluation Corporation, *SPECjvm98 Benchmarks* (see http://www.spec.org/osg/jvm98/).
34. Note that these results do *not* follow the official SPEC reporting rules, and therefore should not be treated as official SPEC results.
35. S. J. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. J. Munroe, "Java Server Benchmarks," *IBM Systems Journal* **39**, No. 1, 57–81 (2000, this issue).
36. A. Taivalsaari, "Implementing a Java Virtual Machine in the Java Programming Language," *Technical Report SMLI TR-98-64*, Sun Microsystems (March 1998).
37. J. Chapin, personal communication regarding the Rivet project at MIT. See http://sdg.lcs.mit.edu/rivet.html for further information.
38. John Duimovich, personal communication.
39. Java Development Kit 1.1 (see http://java.sun.com/marketing/collateral/jdk_sc.html).
40. The Java Hotspot Performance Engine Architecture (April 1999). White paper available at http://java.sun.com/products/hotspot/whitepaper.html.
41. See http://www.kaffe.org/.
42. A. Krall and R. Grafl, "CACAO—A 64 bit Java VM Just-

in-Time Compiler," *Concurrency: Practice and Experience* **9**, No. 11 (1987).
43. B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman, "LaTTe: A Java VM Just-In-Time Compiler with Fast and Efficient Register Allocation," *Proceedings, IEEE International Conference on Parallel Architectures and Compilation Techniques* (PACT'99), Newport Beach, CA (October 12–16, 1999), pp. 128–138.
44. With the train algorithm, all mutators are halted, but garbage collection is done on only part of the heap. Thus the mutators have only a short "pause time." In Jalapeño, we reduce the pause time by running a parallel collector, using multiple CPUs.
45. D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, "The Story of Squeak, A Practical Smalltalk Written in Itself," *Proceedings, ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Atlanta, GA (October 5–9, 1997), pp. 318–326.
46. L. P. Deutsch and A. M. Schiffman, "Efficient Implementation of the Smalltalk-80 System," *Proceedings, 11th Annual ACM Symposium on the Principles of Programming Languages*, Salt Lake City, Utah (January 15–18, 1984), pp. 297–302.
47. C. Chambers, *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*, Ph.D. thesis, Stanford University (March 1992). Published as technical report STAN-CS-92-1420.
48. A.-R. Ald-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth, "Fast, Effective Code Generation in a Just-in-Time Java Compiler," *Proceedings, SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada (June 17–19, 1998).
49. K. Ebcioglu and E. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," IBM Technical Report RC 20538 (1996).
50. C. Consel and F. Noël, "A General Approach for Run-Time Specialization and Its Application to C," *Proceedings, 23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL (January 21–24, 1996), pp. 145–156.
51. J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad, "Fast, Effective Dynamic Compilation," *Proceedings, SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA (May 21–24, 1996), pp. 149–159.
52. M. Poletto, D. R. Engler, and M. Frans Kaashoek, "tcc: A System for Fast, Flexible, and High-Level Dynamic Code Generation," *Proceedings, SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NE (June 16–18, 1997), pp. 109–121.
53. B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers, "DyC: An Expressive Annotation-Directed Dynamic Compiler for C," *Theoretical Computer Science*, to appear.
54. C. Chambers and D. Ungar, "Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs," *Proceedings, SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY (June 20–22, 1990), pp. 150–164.
55. D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call Graph Construction in Object-Oriented Languages," *Proceedings, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Atlanta, GA (October 5–9, 1997), pp. 108–124.
56. F. Tip and P. F. Sweeney, "Class Hierarchy Specialization," *Proceedings, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Atlanta, GA (October 5–9, 1997), pp. 271–285.
57. P. F. Sweeney and F. Tip, "A Study of Dead Data Members in C++ Applications," *Proceedings, SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada (June 17–19, 1998), pp. 324–332.
58. C. Chambers, J. Dean, and D. Grove, *Whole-Program Optimization of Object-Oriented Languages*, Technical Report UW-CSE-96-06-02, University of Washington, Department of Computer Science and Engineering (June 1996).
59. U. Hölzle and D. Ungar, "Optimizing Dynamically Dispatched Calls with Run-Time Type Feedback," *Proceedings, SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL (June 20–24, 1994), pp. 326–336.
60. S. P. Midkiff, J. E. Moreira, and M. Snir, "Optimizing Array Reference Checking in Java Programs," *IBM Systems Journal* **37**, No. 3, 409–453 (1998).

Accepted for publication October 5, 1999.

Bowen Alpern *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: alpern@watson.ibm.com).* Dr. Alpern joined the IBM research staff after receiving a Ph.D. degree from Cornell University in 1986. His research interests include virtual machine implementation, Java technology, concurrent and parallel programming, synchronization mechanisms, and the parallel memory hierarchy (PMH) model of computation.

C. R. Attanasio *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: dick@watson.ibm.com).* Mr. Attanasio joined IBM in 1965 as a systems programmer and has been involved in various systems programming and analysis activities ever since, including security evaluation of VM/370, AIX-based networking enhancements, and a bytecode verifier for Java. As part of the Jalapeño project he has implemented the noncopying nongenerational and generational storage allocators and garbage collectors.

John J. Barton *Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, California 94304-1126 (electronic mail: John_Barton@hpl.hp.com).* Dr. Barton wrote the Jalapeño boot image writer and managed the Java Technology group during the initial stages of the Jalapeño project. He also worked on the research project that led to IBM's VisualAge C++ v 4.0 product. He has a Ph.D. degree in chemistry from the University of California at Berkeley and a master's degree in applied physics from the California Institute of Technology. Dr. Barton now works for Hewlett-Packard Laboratories.

Michael G. Burke *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: mgburke@us.ibm.com).* Dr. Burke is currently a research staff member and manager of the Object-Oriented Optimization group at the IBM Thomas J. Watson Research Center. He received a B.A. degree in philosophy from Yale University in 1973. His studies at the Courant Institute of New York University resulted in an M.S. degree in 1979 and a Ph.D. degree in 1983 in computer science. His current research interests include compiler optimization, program analysis, compiling object-oriented languages, and e-business language and performance technology.

Perry Cheng *Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania 15213 (electronic mail: pscheng@cs.cmu.edu)*. Mr. Cheng received his B.S. degree (1994) in mathematics and computer science from Rice University and his M.S. degree (1997) in computer science from Carnegie Mellon University. He is currently working on his doctoral thesis, "Scalable Garbage Collection for Shared-Memory Multiprocessors," at Carnegie Mellon University.

Jong-Deok Choi *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: jdchoi@us.ibm.com).* Dr. Choi received the B.S. degree in electronic engineering from Seoul National University (SNU), Seoul, Korea, in 1979; the M.S. degree in electrical engineering from Korea Advanced Institute of Science and Technology (KAIST) in 1981; and the M.S. and Ph.D. degrees, both in computer science, from the University of Wisconsin, Madison, in 1985 and 1989, respectively. Since September 1989, he has been at the IBM Thomas J. Watson Research Center as a research staff member. His research interests include optimizing compilers, programming environments for parallel and distributed systems, and parallel program debugging.

Anthony Cocchi *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: tony@watson.ibm.com).* Mr. Cocchi is a senior software engineer at the Thomas J. Watson Research Center. He received B.S. and M.S. degrees in electrical engineering from Pratt Institute, Brooklyn, New York, and an M.S. degree from New York Polytechnic University. On the Jalapeño project, he has worked on the run-time and garbage collection components and on system performance.

Stephen J. Fink *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: sjfink@us.ibm.com).* Dr. Fink is a research staff member at the Thomas J. Watson Research Center. He received the B.S. degree from Duke University, Durham, North Carolina, in 1992, and the M.S. and Ph.D. degrees from the University of California, San Diego, in 1994 and 1998, respectively. His research interests include dynamic compilation, run-time systems, object-oriented programming, and parallel scientific computation.

David Grove *IBM Research Division, Thomas J. Watson Research Center, Box 704, Yorktown Heights, New York 10598 (electronic mail: groved@us.ibm.com)*. Dr. Grove is a research staff member at the IBM Thomas J. Watson Research Center. He received the Ph.D. degree from the University of Washington, Seattle in 1998, where he worked on the Cecil/Vortex research project under Professor Craig Chambers. His primary research interests are in the design, analysis, and optimization of object-oriented programming languages.

Michael Hind *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: hindm@us.ibm.com).* Dr. Hind is a research staff member at the Thomas J. Watson Research Center. He received his Ph.D. degree from New York University in 1991. He is currently working on the Jalapeño Jvm as a member of the Dynamic Optimization group. His research interests include program analysis, adaptive compilation, and programming languages.

Susan Flynn-Hummel *IBM Research Division, Thomas J. Watson Research Center, Box 218, Yorktown Heights, New York 10598 (electronic mail: hummel@watson.ibm.com)*. Dr. Flynn-Hummel is a research staff member at the IBM Thomas J. Watson Research Center. She received her B.A. degree in mathematics from McGill University in 1980, and her Ph.D. degree in computer science from New York University in 1989. Her research interests include parallel computing and computer visualization.

Derek Lieber *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: derek@watson.ibm.com).* Mr. Lieber is a senior software engineer at the Thomas J. Watson Research Center. His interests include interactive graphical debuggers, Java virtual machine architectures, and operating systems.

Vassily Litvinov *University of Washington, Computer Science and Engineering, Box 352350, Seattle, Washington 98195 (electronic mail: vass@cs.washington.edu)*. Mr. Litvinov is a graduate student in computer science at the University of Washington. He works on a flexible type system for an advanced object-oriented programming language, pursuing his passion for tools that reduce the cost of software development. In the "real world" he has enjoyed two summer internships working on optimizing JIT compilers for the Java language. Mr. Litvinov holds an undergraduate degree with distinction from Moscow Institute of Physics and Technology, Russia.

Mark F. Mergen *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: mergen@us.ibm.com).* Dr. Mergen manages the Jalapeño virtual machine, garbage collection, and nonoptimizing compilers group and is responsible for code generation in the quick compiler. He previously managed research work that led to the High-Performance Compiler for Java (HPCJ) product. He is also interested in software simplification and operating systems, and has previously worked on 64-bit AIX, PowerPC virtual memory architecture, and paging systems. He has a B.S. degree (in mathematics) and an M.D. degree, both from the University of Wisconsin at Madison.

Ton Ngo *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: ton@us.ibm.com).* Dr. Ngo received his Ph.D. degree (1997) and M.S. degree (1992) in computer science from the University of Washington, his M.S. degree (1986) in electrical engineering from the Florida Institute of Technology, and his B.S. degree (1982) in electrical engineering from the Georgia Institute of Technology. He joined the IBM System Products Division in 1982, then joined the IBM Research Division in 1987. His past research included parallel systems and parallel languages. Currently, he develops the dynamic debugger and other run-time subsystems for the Jalapeño project.

James R. Russell *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: jrr@us.ibm.com).* Dr. Russell is a research staff member at the Thomas J. Watson Research Center, and is department group manager of the Software Technology department there. His research over the past several years has progressed over distributed systems, tools for developing distributed applications, Web application servers, and applications of Java technology. He holds a Ph.D. degree in computer science from Cornell University, Ithaca, New York.

Vivek Sarkar *IBM Research Division, Thomas J. Watson Research Center, Box 704, Yorktown Heights, New York 10598 (electronic mail: vsarkar@us.ibm.com)*. Dr. Sarkar is a research staff member and manager of the Dynamic Compilation group at the IBM Thomas J. Watson Research Center. He joined IBM in 1987, after obtaining a Ph.D. degree from Stanford University. His previous work at IBM includes being a member of the PTRAN research project, and leading a product development project for including high-order transformations in the XL FORTRAN compilers. He has been a member of the IBM Academy of Technology since 1995.

Mauricio J. Serrano *IBM Research Division, Thomas J. Watson Research Center, Box 704, Yorktown Heights, New York 10598 (electronic mail: mserrano@us.ibm.com)*. Dr. Serrano received his Ph.D. degree in computer engineering from the University of California, Santa Barbara, in 1994. He is currently visiting the Thomas J. Watson Research Center for the Jalapeño project. His current interests are in object-oriented optimizations and trade-offs between dynamic and static compilation.

Janice C. Shepherd *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: janshep@us.ibm.com).* Ms. Shepherd is a senior software engineer at the Thomas J. Watson Research Center. She is on a one-year assignment at the IBM Tokyo Research Laboratory. She received her B.S. degree from Queens University in 1980 and her master's degree from the University of Toronto in 1983. Ms. Shepherd also represents IBM on the European Computer Manufacturers Association (ECMA) TC41 Java standards committee.

Stephen E. Smith *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: steve@watson.ibm.com).* Dr. Smith is a research staff member at the Thomas J. Watson Research Center. He received his Ph.D. degree from Northwestern University, Evanston, Illinois, in 1970. Since joining IBM he has worked on a number of different projects, primarily in the areas of operating systems and databases. He is currently with the Jalapeño project, working in the areas of storage allocation and garbage collection.

V. C. Sreedhar *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: vugranam@us.ibm.com).* Dr. Sreedhar received his Ph.D. degree from McGill University in 1995. He worked for the Hewlett-Packard Company for three years before joining IBM. His research focus is in the areas of programming languages and systems.

Harini Srinivasan *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: harini@us.ibm.com).* Dr. Srinivasan received her Ph.D. degree from the University of Colorado at Boulder. Her interests include program analysis and program understanding tools for explicitly parallel and concurrent object-oriented programs. She is currently a research staff member in the Jalapeño optimizing compiler group.

John Whaley *IBM Tokyo Research Laboratory, IBM Japan, Ltd., 1623-14 Shimotsuruma, Yamato-shi, Kanagawa-ken 242-8502 Japan (electronic mail: jwhaley@alum.mit.edu)*. Mr. Whaley is currently a research intern at the IBM Tokyo Research Laboratory in Yamato, Japan, working in the Network Computing Platform group on their Java JIT compiler. From January through August, 1998, he worked on the Jalapeño virtual machine, primarily on the optimizing compiler. He has a B.S. degree in computer science and an M.Eng. degree in electrical engineering and computer science, both from the Massachusetts Institute of Technology. His research interests include program analysis, dynamic compilation, and virtual machines.

Reprint Order No. G321-5724.