# Design of the Java HotSpot™ Client Compiler for Java 6

THOMAS KOTZMANN, CHRISTIAN WIMMER
and HANSPETER MÖSSENBÖCK
Johannes Kepler University Linz
and
THOMAS RODRIGUEZ, KENNETH RUSSELL, and DAVID COX
Sun Microsystems, Inc.

Version 6 of Sun Microsystems' Java HotSpot™ VM ships with a redesigned version of the client just-in-time compiler that includes several research results of the last years. The client compiler is at the heart of the VM configuration used by default for interactive desktop applications. For such applications, low startup and pause times are more important than peak performance. This paper outlines the new architecture of the client compiler and shows how it interacts with the VM. It presents the intermediate representation that now uses static single-assignment (SSA) form and the linear scan algorithm for global register allocation. Efficient support for exception handling and deoptimization fulfills the demands that are imposed by the dynamic features of the Java programming language. The evaluation shows that the new client compiler generates better code in less time. The popular SPECjvm98 benchmark suite is executed 45% faster, while the compilation speed is also up to 40% better. This indicates that a carefully selected set of global optimizations can also be integrated in just-in-time compilers that focus on compilation speed and not on peak performance. In addition, the paper presents the impact of several optimizations on execution and compilation speed. As the source code is freely available, the Java HotSpot™ VM and the client compiler are the ideal basis for experiments with new feedback-directed optimizations in a production-level Java just-in-time compiler. The paper outlines research projects that add fast algorithms for escape analysis, automatic object inlining, and array bounds check elimination.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization, Code generation*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Java, compiler, just-in-time compilation, optimization, intermediate representation, register allocation, deoptimization

7

Authors' addresses: Thomas Kotzmann, Christian Wimmer, and Hanspeter Mössenböck, Institute for System Software, Christian Doppler Laboratory for Automated Software Engineering, Johannes Kepler University Linz, Austria; email: {kotzmann, wimmer, moessenboeck}@ssw.jku.at.
Thomas Rodriguez, Kenneth Russell, and David Cox, Sun Microsystems, Inc., 4140 Network Circle, Santa Clara, CA 95054; email: {thomas.rodriguez, kenneth.russell, david.cox}@sun.com.

**ACM Reference Format:**
Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., and Cox, D. 2008. Design of the Java HotSpot™ client compiler for Java 6. ACM Trans. Architec. Code Optim. 5, 1, Article 7 (May 2008), 32 pages. DOI = 10.1145/1369396.1370017 http://doi.acm.org/10.1145/1369396.1370017.

## 1. INTRODUCTION

About 2 years after the release of Java SE 5.0, Sun Microsystems completed version 6 of the Java platform. In contrast to the previous version, it did not introduce any changes in the language, but provided a lot of improvements behind the scenes [Coward 2006]. Especially the Java HotSpot™ client compiler, one of the two just-in-time compilers in Sun Microsystems' virtual machine, was subject to several modifications and promises a notable gain in performance.

Even Sun's development process differs from previous releases. For the first time in the history of Java, weekly source snapshots of the project have been published on the Internet [Sun Microsystems, Inc. 2006b]. This approach is part of Sun's new ambitions in the open-source area [OpenJDK 2007] and encourages developers throughout the world to submit their contributions and bugfixes.

The collaboration on this project requires a thorough understanding of the virtual machine, so it is more important than ever to describe and explain its parts and their functionality. At the current point in time, however, the available documentation of internals is rather sparse and incomplete. This paper gives an insight into those parts of the virtual machine that are responsible for or affected by the just-in-time compilation of bytecodes. It contributes the following:

—It starts with an overview of how just-in-time compilation is embedded in the virtual machine and when it is invoked.
—It describes the structure of the client compiler and its compilation phases.
—It explains the different intermediate representations and the operations that are performed on them.
—It discusses ongoing research on advanced compiler optimizations and details how these optimizations are affected by just-in-time compilation.
—It evaluates the performance gains both compared to the previous version of Sun's JDK and to competitive products.

Despite the fact that this paper focuses on characteristics of Sun's VM and presupposes some knowledge about compilers, it not only addresses people who are actually confronted with the source code but everybody who is interested in the internals of the JVM or in compiler optimization research. Therefore, it will give both a general insight into involved algorithms and describe how they are implemented in Sun's VM.

### 1.1 Architecture of the Java HotSpot™ VM

Java achieves portability by translating source code [Gosling et al. 2005] into platform-independent bytecodes. To run Java programs on a particular platform, a *Java virtual machine* [Lindholm and Yellin 1999] must exist for that

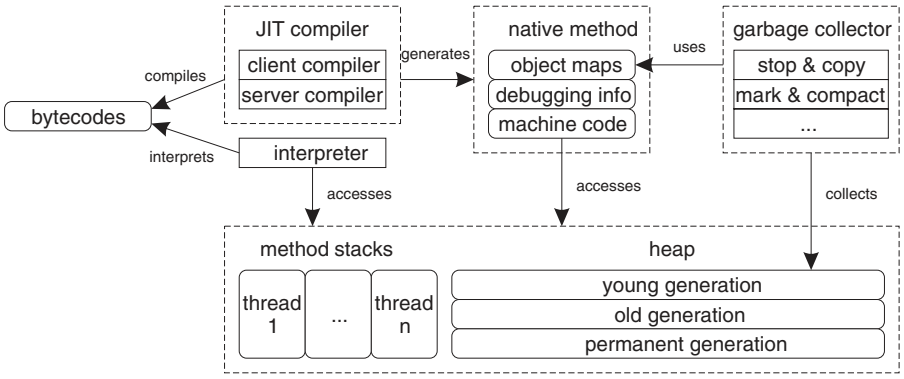



Fig. 1. Architecture of the Java HotSpot™ VM.

platform. It executes bytecodes after checking that they do not compromise the security or reliability of the underlying machine. Sun Microsystems' implementation of such a virtual machine is called Java HotSpot™ VM [Sun Microsystems, Inc. 2006a].

The overall architecture is shown in Figure 1. The execution of a Java program starts in the interpreter, which steps through the bytecodes of a method and executes a code template for each instruction. Only the most frequently called methods, referred to as *hot spots*, are scheduled for *just-in-time (JIT) compilation*. As most classes used in a method are loaded during interpretation, information about them is already available at the time of JIT compilation. This information allows the compiler to inline more methods and to generate better optimized machine code.

If a method contains a long-running loop, it may be compiled regardless of its invocation frequency. The VM counts the number of backward branches taken and, when a threshold is reached, it suspends interpretation and compiles the running method. A new stack frame for the native method is set up and initialized to match the interpreter's stack frame. Execution of the method then continues using the machine code of the native method. Switching from interpreted to compiled code in the middle of a running method is called *on-stack-replacement* (OSR) [Hölzle and Ungar 1994; Fink and Qian 2003].

The Java HotSpot™ VM has two alternative just-in-time compilers: the *server* and the *client compiler*. The server compiler [Paleczny et al. 2001] is a highly optimizing compiler tuned for peak performance at the cost of compilation speed. Low compilation speed is acceptable for long-running server applications, because compilation impairs performance only during the warm-up phase and can usually be done in the background if multiple processors are available.

For interactive client programs with graphical user interfaces, however, response time is more important than peak performance. For this purpose, the *client compiler* was designed to achieve a trade-off between the performance of the generated machine code and compilation speed [Griesemer and Mitrovic 2000]. This paper presents the architecture of the revised client compiler in the JDK 6.

All modern Java virtual machines implement synchronization with a thin lock scheme [Agesen et al. 1999; Bacon et al. 1998]. Sun's JDK 6 extends this concept by *biased locking* [Russell and Detlefs 2006], which uses concepts similar to Kawachiya et al. [2002]. Previously, an object was locked always atomically just in case that two threads synchronize on it at the same time. In the context of biased locking, a pointer to the current thread is stored in the header of an object when it is locked for the first time. The object is then said to be *biased* toward the thread. As long as the object is locked and unlocked by the same thread, synchronizations need not be atomic.

The generational garbage collector [Ungar 1984] of the Java HotSpot™ VM manages dynamically allocated memory. It uses exact garbage collection techniques, so every object and every pointer to an object must be precisely known at GC time. This is essential for supporting compacting collection algorithms. The memory is split into three generations: a young generation for newly allocated objects, an old generation for long-lived objects, and a permanent generation for internal data structures.

New objects are allocated sequentially in the young generation. Since each thread has a separate *thread-local allocation buffer* (TLAB), allocation operations are multithread-safe without any locking. When the young generation fills up, a *stop-and-copy* garbage collection is initiated. When objects have survived a certain number of collection cycles, they are promoted to the old generation, which is collected by a *mark-and-compact* algorithm [Jones and Lins 1996].

The Java HotSpot™ VM also provides various other garbage collectors [Sun Microsystems, Inc. 2006c]. Parallel garbage collectors for server machines with large physical memories and multiple CPUs distribute the work among multiple threads, thus decreasing the garbage collection overhead and increasing the application throughput. A concurrent mark-and-sweep algorithm [Boehm et al. 1991; Printezis and Detlefs 2000] allows the user program to continue its execution while dead objects are reclaimed.

Exact garbage collection requires information about pointers to heap objects. For machine code, this information is contained in *object maps* (also called *oop maps*) created by the JIT compiler. Besides, the compiler creates *debugging information* that maps the state of a compiled method back to the state of the interpreter. This enables aggressive compiler optimizations, because the VM can *deoptimize* [Hölzle et al. 1992] back to a safe state when the assumptions under which an optimization was performed are invalidated (see Section 2.6). The machine code, the object maps, and the debugging information are stored together in a so-called *native method object*. Garbage collection and deoptimization are allowed to occur only at some discrete points in the program, called *safepoints*, such as backward branches, method calls, return instructions, and operations that may throw an exception.

Apart from advanced JIT compilers, sophisticated mechanisms for synchronization, and state-of-the-art garbage collectors, the new Java HotSpot™ VM also features object packing functionality to minimize the wasted space between data types of different sizes, on-the-fly class redefinition, and full-speed debugging. It is available in 32-bit and 64-bit editions for the Solaris operating system on SPARC and Intel platforms, for Linux, and for Microsoft Windows.

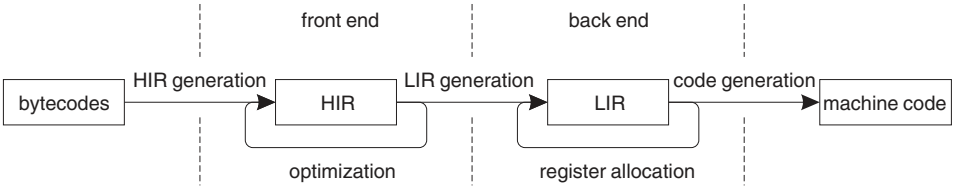



Fig. 2. Structure of the Java HotSpot™ client compiler.

### 1.2 Design Changes of the Client Compiler

The design changes of the client compiler for version 6 focus on a more aggressive optimization of the machine code. In its original design, the client compiler implemented only a few high-impact optimizations, whereas the server compiler performed global optimizations across basic block boundaries, such as global value numbering or loop unrolling. The goal for Java 6 was to adopt some of these optimizations for the client compiler.

Before Java 6, the high-level intermediate representation (HIR) of the client compiler was not suitable for global optimizations. It was not in static single-assignment (SSA) form [Cytron et al. 1991] and required local variables to be explicitly loaded and stored. The new HIR is in SSA form. Load and store instructions for local variables are eliminated by keeping track of the virtual registers that contain the variables' current values [Mössenböck 2000].

Another major design change was the implementation of a linear scan register allocator [Mössenböck and Pfeiffer 2002; Wimmer and Mössenböck 2005]. The previous approach was to allocate a register immediately before an instruction and free it after the instruction has been processed. Only if a register remained unassigned throughout a method, it was used to cache the value of a frequently accessed local variable. This algorithm was simple and fast, but resulted in a large number of memory loads and stores. The linear scan register allocation produces more efficient machine code and is still faster than the graph coloring algorithm used by the server compiler.

The design changes and the SSA form, in particular, facilitate a new family of global optimizations. Ongoing research projects deal with escape analysis and automatic object inlining to reduce the costs associated with memory management. These projects are described in Section 3.

## 2. STRUCTURE OF THE CLIENT COMPILER

The client compiler is a just-in-time compiler that aims at a low startup time and a small memory footprint. The compilation of a method is split into three phases, allowing more optimizations to be done than in a single pass over the bytecodes. All information communicated between the phases is stored in intermediate representations of the program.

Figure 2 shows the structure of the client compiler. First, a *high-level intermediate representation* (HIR) of the compiled method is built via an abstract interpretation of the bytecodes. It consists of a control-flow graph (CFG), whose basic blocks are singly linked lists of instructions. The HIR is in *static single-assignment (SSA) form*, which means that for every variable there is just a

single point in the program where a value is assigned to it. An instruction that loads or computes a value represents both the operation and its result, so that operands can be represented as pointers to previous instructions. Both during and after generation of the HIR, several optimizations are performed, such as constant folding, value numbering, method inlining, and null check elimination. They benefit from the simple structure of the HIR and the SSA form.

The back end of the compiler translates the optimized HIR into a *low-level intermediate representation* (LIR). The LIR is conceptually similar to machine code, but still mostly platform-independent. In contrast to HIR instructions, LIR operations operate on virtual registers instead of references to previous instructions. The LIR facilitates various low-level optimizations and is the input for the *linear scan register allocator*, which maps virtual registers to physical ones.

After register allocation, machine code can be generated in a rather simple and straightforward way. The compiler traverses the LIR, operation by operation, and emits appropriate machine instructions into a code buffer. This process also yields object maps and debugging information.

### 2.1 High-Level Intermediate Representation

The high-level intermediate representation (HIR) is a graph-based representation of the method using SSA form [Cytron et al. 1991]. It is platform-independent and represents the method at a high level where global optimizations are easy to apply. We build the SSA form at parse time of the bytecodes, similarly to Click and Paleczny [1995]. The modeling of instruction types as a C++ class hierarchy and the representation of operands are other similarities to this intermediate representation.

The control flow is modeled using an explicit CFG, whose nodes represent basic blocks, i.e. longest possible sequences of instructions without jumps or jump targets in the middle. Only the last instruction can be a jump to one or more successor blocks or represent the end of the method. Because instructions that can throw exceptions do not terminate a basic block, control can also be transferred to an exception handler in the middle of a block (see Section 2.5).

The instruction types of the HIR are represented by a class hierarchy with a subclass for each kind of instruction. The instruction nodes also form the data flow: Instructions refer to their arguments via pointers to other instructions. This way, an instruction represents both the computation of a result and the result itself. Because of this equivalence, an instruction is often referred to as a *value*. The argument need not be defined in the same block, but can also be defined in a dominator, i.e. a common predecessor on all input paths. Instead of explicit instructions for accessing local variables, instructions reference those instructions that compute the most recent value of the variables. Figure 3 shows an example for the control and data flow of a short loop.

The HIR is constructed in two passes over the bytecodes. The first pass detects the boundaries of all basic blocks and performs a simple loop analysis to mark loop header blocks. Backward-branch targets of irreducible loops are are also treated as loop headers. The basic blocks are created, but not linked

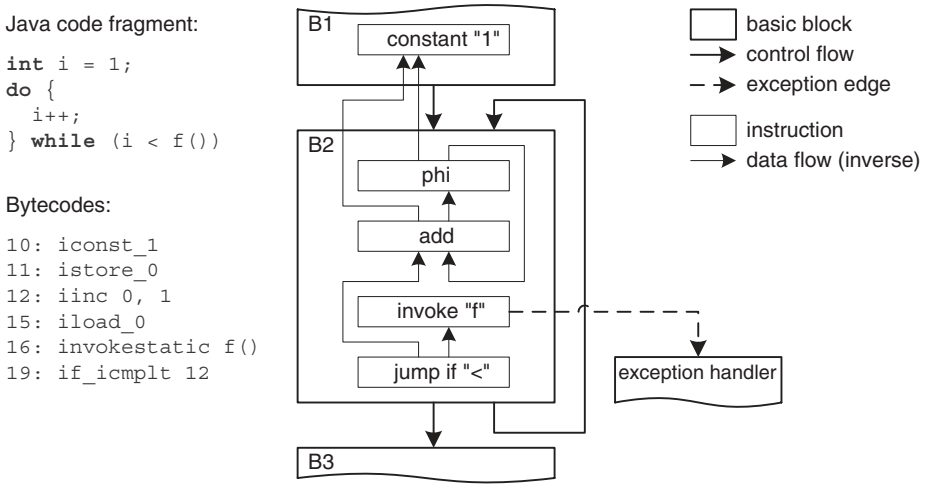



Fig. 3. HIR example with control and data flow.

together. The second pass creates the instructions by abstract interpretation of the bytecodes, appends them to their basic block, and links the blocks to build the CFG.

Inlining of methods is embedded into the analysis: When the bytecodes contain a call to a short method that can be statically bound, the HIR construction is called recursively for the callee and the resulting basic blocks and instructions are appended to the CFG.

The SSA form requires a single point of assignment for each variable. When control flow joins, so-called *phi functions* merge different values of the same variable. In the example, the phi function merges the initial value `1` and the result of the addition for the loop variable `i`. If a block has more than one predecessor, phi functions might be necessary at the beginning of this block. They are created before the instructions of the block are inserted using the following strategy that does not require data-flow analysis. The server compiler [Paleczny et al. 2001] uses a similar approach.

—When the block is no loop header, all predecessors are already filled with instructions. If a variable has different values at the end of the predecessors, a phi function is created. If the value is equal in all predecessors, no phi function is necessary and the value is used directly.

—For loop headers, the state of the variables for the backward branch is not yet known. Therefore, phi functions are created conservatively for all variables.

The loop analysis of the first pass is used for further optimizations: If a variable is never assigned a value inside a loop, the phi function for this variable can also be omitted in loop headers. With this, no phi functions are created for method parameters that are not changed inside the method, e.g. for the frequently accessed `this` pointer. This simplifies optimizations that are applied during HIR construction because the type of parameters and loop-invariant instructions is not hidden behind a phi function. Nevertheless, the conservative

creation can lead to phi functions where all operands are equal. Such phi functions are eliminated after the HIR is constructed.

### 2.2 Optimizations

Local optimizations are performed during the generation of the HIR: *Constant folding* simplifies arithmetic instructions with constant operands, as well as branches where the condition is always true or false. *Local value numbering* eliminates common subexpressions within a basic block. *Method inlining* replaces a method call by a copy of the method body.

Method inlining eliminates the overhead of method dispatching. The client compiler uses a static inlining strategy: Methods with a size less than 35 bytes are inlined. This size is decreased for each nested inlining because the number of inlining candidates grows at each level. However, the method must be bound statically to be inlined, i.e. the compiler must be able to unambiguously determine the actual target method, which is complicated because most methods in Java are virtual methods. A class hierarchy analysis (CHA) [Dean et al. 1995] is used to detect virtual call sites where currently only one suitable method exists. This method is then optimistically inlined. If a class is loaded later that adds another suitable method and, therefore, the optimistic assumption no longer holds, the method is deoptimized (see Section 2.6).

After the generation of the HIR, global optimizations are performed. *Null-check elimination* (see e.g. [Kawahito et al. 2000]) removes null checks if the compiler can prove that an accessed object is non null. *Conditional expression elimination* replaces the common code pattern of a branch that loads one of two values by a conditional move instruction. *Global value numbering* (see e.g. Briggs et al. [1997]) eliminates common subexpressions across basic block boundaries.

Global value numbering is an example for an optimization that was simplified by the introduction of SSA form. Our implementation builds on a scoped hash table and requires only a single pass over the CFG in reverse postorder. The processing of a basic block starts with the state of the hash table at the end of its immediate dominator. At the beginning of the block, a new scope of the hash table is opened to keep newly inserted instructions separate from the instructions that were added in previous blocks.

Two instructions are equivalent if they perform the same operation on the same operands. If an equivalent old instruction is found for a new instruction, the new instruction is deleted and uses are changed to the old instruction. Otherwise, the instruction is added to the hash table.

Value numbering of field and array loads requires an additional analysis: Two loads of the same field for the same object are not necessarily redundant because a store to the field can be in between. The field store is said to kill all previous loads of the same field. Some instructions, e.g. method calls, kill all memory loads because their exact memory behavior is unknown. The set of killed instructions is propagated down the control flow. Before a basic block is processed, the killed sets of all predecessors are used to remove instructions from the hash table of the dominator.

When a predecessor has not yet been processed, e.g. because it contains the backward branch of a loop, all memory loads must be killed. To avoid this for short loops consisting of only a few basic blocks, we analyze these blocks and kill only the necessary loads if the loop contains no method calls.

### 2.3 Low-Level Intermediate Representation

The low-level intermediate representation (LIR) is close to a three-operand machine code, augmented with some higher level instructions, e.g. for object allocation and locking. The data structures are shared between all target platforms, but the LIR generation already contains platform-dependent code. The LIR reuses the control flow of the HIR, so it contains no new data structures for basic blocks.

LIR instructions use explicit operands that can be virtual registers, physical registers, memory addresses, stack slots, or constants. The LIR is more suitable for low-level optimizations such as register allocation than the HIR, because all operands requiring a machine register are explicitly visible. Constraints of the target architecture, such as machine instructions that require a specific register, are already modeled in the LIR using physical register operands. After the register allocator has replaced all virtual registers by physical registers or stack slots, each LIR instruction can be mapped to a pattern of one or more machine instructions. The LIR does not use SSA form, so the phi functions of the HIR are resolved by register moves.

Many LIR instructions can be divided into a common and an uncommon case. Examples for uncommon cases are calling the garbage collector for memory allocations or throwing bounds-check exceptions for array accesses. Following the *focus on the common case* principle, the machine instructions for the common case are emitted in-line, while the instructions for the uncommon case are put out-of-line at the end of the method.

### 2.4 Linear Scan Register Allocation

Register allocation is the task of assigning physical registers to local variables and temporary values. The most commonly used approach is based on graph-coloring algorithms (see for example [Briggs et al. 1994]). In comparison, the linear-scan algorithm is simpler and faster. The basic algorithm of Poletto and Sarkar [1999] assigns registers to values in a single linear pass. Our implementation is influenced by the extended version of Traub et al. [1998].

For register allocation, all basic blocks of the LIR are topologically sorted into a linear order. All blocks belonging to a loop are consecutively emitted and rarely executed blocks, such as exception handlers, are placed at the end of the method or the loop.

For each virtual register of the LIR, a *lifetime interval* is constructed that stores the lifetime of the register as a list of disjoint ranges. *Fixed intervals* are built for physical register operands to model register constraints of the target architecture and method calls. *Use positions* of an interval refer to the instructions where the register is read or written. Depending on the kind of the use position, our register allocation algorithm either guarantees that the interval has a register assigned at this position, or at least tries to assign a

```
10 move   v1 = 1
12 label  B2
14 move   v2 = v1
16 add    v2 = v2 + 1
18 call   eax = f()
20 move   v3 = eax
22 cmp    v2, v3
24 branch GE, B3
26 label  B4
28 move   v1 = v2
30 branch AL, B2
32 label  B3
```

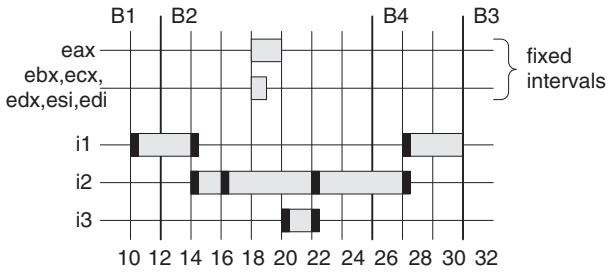



Fig. 4. LIR example with lifetime intervals and use positions.

register. Between two use positions, the algorithm can freely decide if the value is kept in a register or spilled to the stack frame. The spilling and reloading of a value is implemented by splitting the interval.

Figure 4 continues the example started in Figure 3 and shows the slightly simplified LIR and the lifetime intervals for the code fragment. The virtual registers `v1`–`v3` are represented by the intervals `i1`–`i3`. Use positions are denoted by black bars. According to typical IA-32 calling conventions, the method call at instruction 18 returns its value in the fixed register `eax`. The fixed intervals of all other registers are merged to a single line. To allow live ranges that do not extend to the next instruction, the instruction numbers are incremented by 2. The phi function of the HIR has been resolved to the move instructions 14 and 28. The block `B4` was created to place the move instruction 28 at the backward branch.

The allocation algorithm processes all lifetime intervals in the order of increasing start positions. Each interval gets a register assigned that is not used by another, simultaneously live interval. The algorithm is documented in Wimmer [2004]. When more intervals are live than physical registers are available, a heuristic is used to select an interval for being spilled to the stack frame. Usually, this interval is split twice to get three shorter intervals: The first part remains in the assigned register, the second part is spilled, and the third part, which must start after the current position, is processed when the algorithm advances to its start position.

The decision which interval is spilled and where this interval is split affects the performance of the resulting machine code. Because finding the optimal solution would be too time consuming, we apply several heuristics. In general, the interval that is not used for the longest time after the current position is spilled. If spilling has to occur inside a loop, intervals that are not used in this loop are spilled first and their split positions are moved out of the loop.

Other optimizations implemented in our register allocator are register hints, which are a lightweight replacement for interval coalescing used by other register allocators, and spill store elimination, which eliminates redundant stores to the same spill slot. Both optimizations reduce the number of move instructions necessary to copy values between registers and spill slots. In the example of Figure 4, the intervals `i1` and `i2` get the same register assigned, because they are connected by a register hint and do not overlap. This eliminates the instructions 14 and 28.

The interval `i2` must be split before instruction 18: Because this call instruction destroys all registers, `i2` must be spilled to memory. The splitting and spilling is automatically enforced by the fixed intervals at instruction 18. Interval `i2` is reloaded before instruction 22, because the use position requires a register. The optimizations are described in more detail in Wimmer and Mössenböck [2005].

Because the linear scan algorithm operates on a flattened CFG, the splitting of intervals can cause conflicts at control-flow edges: For example, if an interval is in a register at the end of a block, but expected to be on the stack at the start of a successor block, a move instruction must be inserted to resolve the conflict. This can happen if an interval flows into a block via two branches and is split in one of them. To have a safe location for these moves, all critical edges of the CFG, i.e. edges from blocks with multiple successors to blocks with multiple predecessors, are split by inserting blocks before register allocation.

### 2.5 Exception Handling

In the HIR, instructions that can throw exceptions do not end a basic block because that would fragment the nonexceptional control flow and lead to a large number of short basic blocks. Our concept is similar to the *factored control-flow graphs* of Choi et al. [1999]. At the beginning of an exception handler, a phi function for a local variable is created if the variable has different values at two or more throwing instructions that lead to this handler. These phi functions are not resolved by moves when the LIR is constructed, because adapter blocks would have to be created for each edge from an instruction to an exception handler. Instead, the phi functions are resolved after register allocation: If the register of an input operand differs from the register of the output operand of such a phi function, a new adapter block is created for the necessary register move. All exception handler entries are collected in a table and stored in the meta data together with the machine code.

When an exception is thrown by the generated machine code at runtime, the corresponding exception handler entry is searched by a runtime function using these tables. This frees the compiler from generating the dispatch logic that searches the correct exception handler using the dynamic type of the exception.

The compiler can add exception handlers that are not present in the bytecodes. This allows the inlining of synchronized methods: The lock that is acquired before the method is executed must be released both at normal returns and at exceptional exits of the method. To catch exceptional exits, the inlined method is surrounded by a synthetic exception handler that catches all exceptions, unlocks the receiver, and then rethrows the exception.

### 2.6 Deoptimization

Although inlining is an important optimization, it has traditionally been very difficult to perform for dynamic object-oriented languages like Java. A method can only be inlined if the compiler identifies the called method statically despite polymorphism and dynamic method binding. Apart from static and final callees, this is possible if class hierarchy analysis (CHA) [Dean et al. 1995] can prove

```
void foo() {
  A p = create();
  p.bar();
}
```

```
A create() {
  if (...) {
    return new A();
  } else {
    return new B();
  }
}
```

```
class A {
  void bar() { ... }
}

class B extends A {
  void bar() { ... }
}
```

Fig. 5. Example for a situation where deoptimization is required.

that only one suitable method exists. If, however, a class is loaded later that overrides that method, possibly the wrong implementation was inlined.

Assume that class `B` has not been loaded yet when machine code for the method `foo`, shown in Figure 5, is generated. The compiler optimistically assumes that there is only one implementation of the method `bar` and inlines `A.bar` into `foo`. If the method `create` later loads class `B` and returns an instance of it, the inlining decision turns out to be wrong and the machine code for `foo` is invalidated. In this situation, the Java HotSpot™ VM is forced to stop the machine code, undo the compiler optimizations and continue execution of the method in the interpreter. This process is called *deoptimization* [Hölzle et al. 1992; Paleczny et al. 2001].

Every time the compiler makes an inlining decision based on class hierarchy information, it records a dependency between the caller and the class that defines the callee. When the VM loads a class later, it examines its superclasses, and marks dependent methods for deoptimization. It also iterates over the interfaces implemented by the new class and looks for methods depending on the fact that an interface had only one implementor.

Since interpreter and machine code use different stack frame layouts, a new stack frame must be set up before execution of a marked method can be continued in the interpreter. Instead of immediately deoptimizing a frame when a class is loaded, the machine instruction that will be executed next for this frame is patched to invoke a runtime stub. The actual deoptimization takes place when the frame is reactivated after all callees have returned (*lazy deoptimization*). The entry point of the native method is also modified so that new invocations are redirected to the interpreter.

In contrast to machine code, interpreted methods do not inline other methods. Therefore, a stack frame of a native method may correspond to multiple interpreter frames. The runtime stub that is called by the patched machine code first creates an array of *virtual stack frames*, one for the method to be deoptimized and one for each currently active inlined callee. A virtual frame does not exist on the stack, but stores the local variables, the operand stack, and monitors of a particular method. The debugging information, which is generated by the compiler and describes the stack frames from the interpreter's point of view, is used to fill the virtual frames with the correct values from registers and memory. This complicates some compiler optimizations because values needed for deoptimization must be kept alive even if they are not used by optimized code. This is accomplished by extending the lifetime intervals of the register allocator.

In the next phase of deoptimization, the method stack is adjusted, as shown in Figure 6. The frames of the runtime stub and the method to be deoptimized

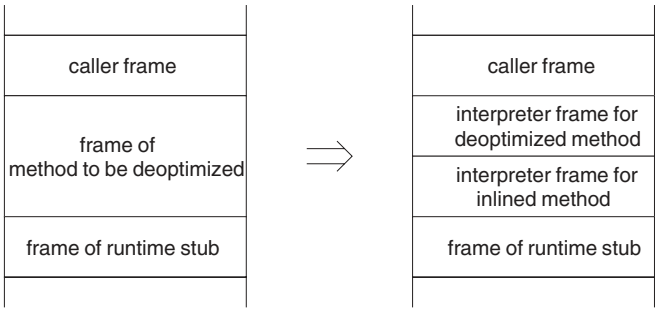



Fig. 6. Unpacking of virtual frames.

are removed and the virtual frames are unpacked onto the stack. Finally, a frame for the continuation of the runtime stub is pushed back onto the stack.

During unpacking of a virtual frame, the bytecode to be executed next is looked up in the debugging information. The address of the interpreter code that handles the bytecode is pushed onto the stack as the return address. This way, execution automatically continues in the interpreter when the runtime stub has completed deoptimization and returns.

Converting a compiled frame to interpreted frames is an expensive operation. It could be avoided if the compiler inlined virtual calls only when the receiver is *preexisting* [Detlefs and Agesen 1999], i.e. allocated before execution of the caller begins. However, deoptimization occurs very rarely: For all benchmarks presented in Section 4, no running methods are deoptimized. Therefore, the runtime costs of deoptimization are irrelevant in practice.

## 3. ONGOING RESEARCH PROJECTS

The relatively simple structure of the client compiler makes it an ideal platform for experiments with compiler optimizations in a production-level Java VM. The challenge is to develop optimizations that improve the quality of the generated code without degrading compilation speed significantly. The projects of this section are not part of the JDK 6, but possible candidates for a future version.

### 3.1 Escape Analysis

Escape analysis is a modern compiler optimization technique for the identification and optimization of objects that are accessible only within one method or thread [Choi et al. 2003; Blanchet 2003]. Such objects are said to *not escape* the method or thread. In the context of a research project, the Java HotSpot™ client compiler was extended by an escape analysis algorithm that is fast enough for just-in-time compilation and capable of coping with dynamic class loading [Kotzmann and Mössenböck 2005; Kotzmann 2005].

Escape analysis is performed in parallel with the construction of the HIR. Objects whose escape states depend on each other are inserted into so-called equi-escape sets, which are implemented as instruction trees based on the union-find algorithm [Sedgewick 1988]. If one of the objects escapes, only the escape state

of the set and not of all contained objects must be adjusted. After completion of the HIR, three different optimizations are applied:

—Scalar replacement: If an object does not escape the creating method, its fields can be replaced by scalar variables. The compiler can eliminate both the allocation of memory on the heap and the initialization of the object. The access of a scalar variable is cheaper than the access of a field, because it does not require any dereferencing.
—Stack allocation: An object that is accessed only by one method and its callees can be allocated on the stack instead of the heap. Stack objects reduce the burden of the garbage collector, because they are implicitly deallocated at the end of the method when the stack frame is removed. Their fields can usually be accessed relative to the base of the current stack frame.
—Synchronization removal: If an object does not escape a thread, synchronization on it can safely be omitted because it will never be locked by another thread.

When objects are passed to another method, the compiler uses interprocedural escape information to determine if the objects escape in the callee. This information is computed during the compilation of a method and stored together with the machine code to avoid reanalyzing the method at each call site. The interprocedural analysis not only allows allocating actual parameters in the stack frame of the caller, but also supports the compiler in inlining decisions: Even if a callee is too large to be inlined by default, the compiler may decide to inline it if this opens new opportunities for scalar replacement.

As long as a method is still interpreted, no interprocedural escape information is available for it. When the compiler reaches a call of a method that has not yet been compiled, it does a quick analysis of the method's bytecodes to determine if a parameter escapes. The bytecode analysis considers each basic block separately and checks if it lets one of the parameters escape. This is more conservative than full escape analysis, but faster to compute because control flow is ignored. When the method is compiled later, the provisional escape information is replaced with more precise data.

Figure 7 gives an example for the escape analysis and the optimizations described above. A point is specified by two floating-point coordinates and a line stores the four coordinates of the start and the end point (Figure 7a). After inlining the constructors (Figure 7b), the object `p2` does not escape the method, so its allocation can be eliminated. The fields are replaced by scalar variables and are probably mapped to registers later (Figure 7c). The `length` method is too large to be inlined. Assume that interprocedural escape information yields that the object `l` does not escape from it and can thus be allocated on the stack.

Since objects for the concatenation of strings are usually not shared among threads, Java 5.0 introduced the unsynchronized class `StringBuilder` for this purpose. Older programs, however, still use the synchronized `StringBuffer` class. Escape analysis and synchronization removal optimize these programs during just-in-time compilation and thus achieve the same performance as if a `StringBuilder` had been chosen.

```
float foo(Point p1) {
  Point p2 = new Point(3, 4);
  Line l = new Line(p1, p2);
  return l.length();
}
```

(a) original method

```
float foo(Point p1) {
  Point p2 = new Point();
  p2.x = 3; p2.y = 4;
  Line l = new Line();
  l.x1 = p1.x; l.y1 = p1.y;
  l.x2 = p2.x; l.y2 = p2.y;
  return l.length();
}
```

(b) after inlining

```
float foo(Point p1) {
  float x2 = 3, y2 = 4;
  Line l = new Line(); // on stack
  l.x1 = p1.x; l.y1 = p1.y;
  l.x2 = x2; l.y2 = y2;
  return l.length();
}
```

(c) optimized method

Fig. 7. Example for scalar replacement and stack allocation.

Escape analysis relies heavily on inlining to prevent objects from escaping the allocating method. Therefore, the compiler inlines methods whose size exceeds the usual threshold if it thus expects to optimize more objects and it uses class hierarchy information to inline methods. If dynamic class loading invalidates an inlining decision of the compiler, the optimizations based on escape analysis must be undone. The debugging information was extended to describe optimized objects and their fields. This enables the deoptimization framework to reallocate scalar-replaced and stack-allocated objects on the heap and relock them if synchronization on them was removed [Kotzmann and Mössenböck 2007]. If the VM runs out of memory during reallocation, it terminates because the program does not expect an exception at this point. This situation is rare though and could be avoided by guaranteeing at method entries that enough heap memory is free to cover a potential reallocation.

### 3.2 Automatic Object Colocation and Inlining

Automatic object inlining reduces the access costs of fields and arrays by improving the cache behavior and by eliminating unnecessary field loads. The order of objects on the heap is changed in such a way that objects accessed together are consecutively placed in memory, so that their relative position is fixed. Object inlining for static compilers is already well studied (see for example [Dolby and Chien 2000] and [Lhoták and Hendren 2005]), but applying it dynamically at runtime is complicated, because dynamic class loading and time constraints preclude a global data-flow analysis.

Changing the order of objects at runtime is called *object colocation* [Wimmer and Mössenböck 2006]. We integrated it into garbage collection where live objects are moved to new locations. Thereby, a *parent object* and those *child objects* that are referenced by the parent's most frequently accessed fields are placed next to each other. When a parent and a child object are colocated, i.e. when the memory offset between the parent and the child is fixed, the child can be treated as a part of the parent (*object inlining* [Wimmer and Mössenböck

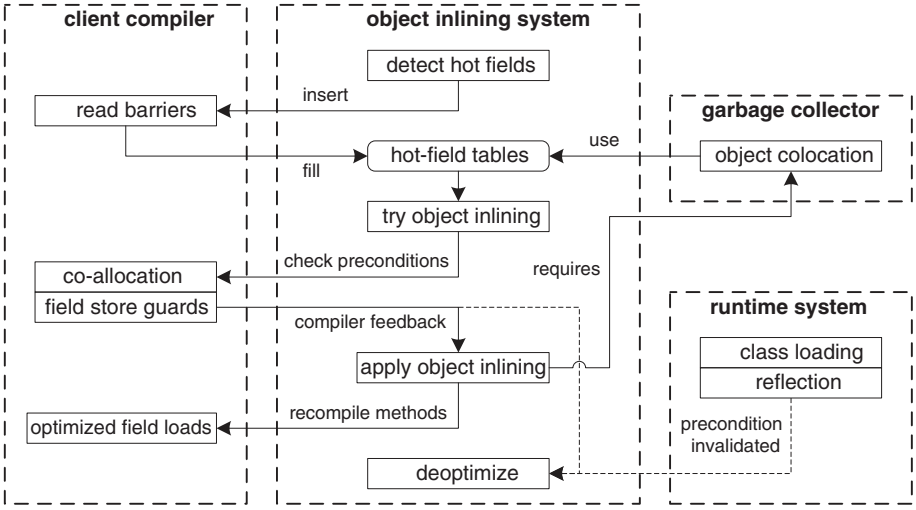



Fig. 8. System structure of automatic object inlining.

2007]). Dereferencing operations for field accesses can be replaced by address arithmetic.

Figure 8 shows the structure of our feedback-directed object inlining system that optimizes an application automatically at runtime and requires no actions on the part of the programmer. It uses the client compiler both to collect information necessary for the optimization and to apply the optimization.

Because the field access pattern depends on how the program is used and which classes are dynamically loaded, the most frequently accessed *hot fields* must be detected at runtime. We use read barriers that are inserted by the client compiler: For each field, a separate counter is incremented when the field is loaded. Because the compiler has all information about an accessed field, the read barrier can be implemented efficiently with a single increment instruction using a fixed counter address. To reduce the runtime overhead, read barriers that are no longer necessary are removed by recompiling methods.

If the counter of a field reaches a threshold, the field is worth for optimizations and added to the *hot-field tables*. The table for a parent class stores a list of child entries for its hot fields. The garbage collector uses these tables for object colocation. The hot fields are also candidates for object inlining. However, a safe execution requires strong guarantees: The objects must stay colocated from allocation until deallocation. To ensure this, we check the following sufficient preconditions for each hot field:

—Co-allocation of objects: A parent object and its child objects must be allocated in the same compiled method. Together with the field stores that install the child objects into the parent, these allocations form a single co-allocation.

—No subsequent field stores: We do not allow the modification of fields that reference inlined objects after the co-allocation. If the field were overwritten later with a new value, the new object would not be colocated to the parent and the optimized field load would still access the old child object. Such field stores

are guarded with a runtime call that triggers deoptimization (see Section 2.6) before the field is changed.

Instead of performing a global data-flow analysis that checks these preconditions, we compile all methods that are relevant for a certain field, i.e. methods that allocate the parent object or store the field. The client compiler reports feedback data if the co-allocation was possible and if all other field stores are guarded.

In this case, object inlining is possible from the current point of view and we apply it optimistically: Methods loading the hot field are recompiled and the access to inlined objects is optimized, i.e. it is replaced by address arithmetic. If a precondition is later invalidated, e.g. by class loading or a field store using reflection, we use deoptimization to undo the changes.

Even if object inlining of a field is not possible, object colocation during garbage collection improves the cache behavior. Therefore, object colocation in the garbage collector processes all hot fields regardless of their object inlining state.

### 3.3 Array Bounds Check Elimination

Array bounds check elimination removes checks of array indices that are proven to be within the valid range. In contrast to other approaches that perform an extensive analysis to eliminate as many bounds checks as possible (see e.g. Bodík et al. [2000] and Qian et al. [2002]), we adhere to the design principle of the client compiler to optimize only the common case. Our algorithm takes advantage of the SSA form and requires only a single iteration over the dominator tree [Würthinger et al. 2007].

The bounds, i.e. the minimum and maximum value, of all HIR instructions that compute an integer result are inferred from the values of constants, the conditions of branches, and arithmetic invariants. Bounds are propagated down the dominator tree: A block inherits all bounds of its immediate dominator. For loops, we focus on the common pattern where a loop variable is incremented by a constant. When the loop variable is not checked against the array length, as in the code fragment of Figure 9, the bounds check cannot be eliminated completely, but it is possible to check the bounds once before the loop.

Figure 9 shows the HIR instructions created for the code fragment and the bounds that are necessary to eliminate the bounds check of the array access inside the loop. The bounds of the initial value of `i` are trivially known to be [0, 0]. Because `i` is modified in the loop, a phi function is created in the loop header block. The phi function is part of a simple cycle that only involves an addition with the constant `1`. Therefore, the value can only increase and the lower bound of the start value is still guaranteed, and the bounds are [0, ?[. The upper bound is not known at this time. It is inferred from the comparison instruction: At the beginning of the loop body block, the upper bound must be `cnt`, because the loop is exited otherwise. Therefore, the bounds of `i` are [0, cnt[.

The upper bound `cnt` is not related to the length of the array `val`, but it is loop-invariant. Therefore, a range check that compares `cnt` to the array length can be inserted before the loop. This eliminates the bounds check in each loop

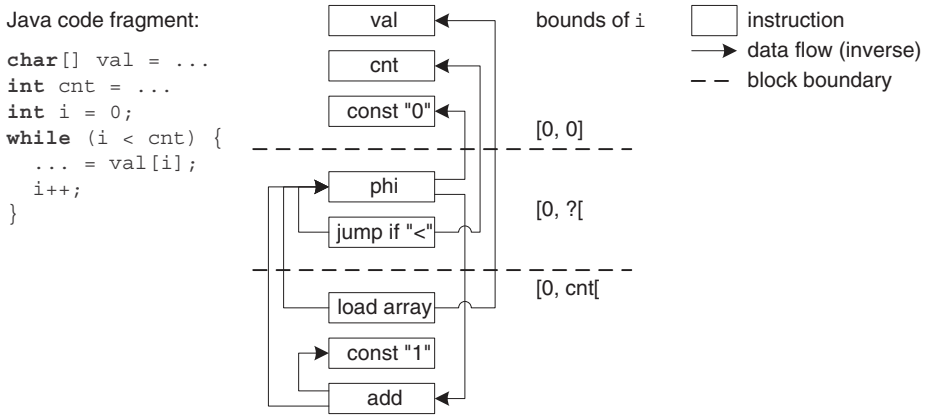



Fig. 9. Example for array bounds check elimination.

iteration. However, it is not allowed to throw an exception in front of the loop because Java requires strict exception semantics. Instead of applying loop versioning and producing an optimized and an unoptimized version of the loop, we trigger a deoptimization (see Section 2.6) when the new bounds check fails. Execution continues in the interpreter that throws the exception in the correct loop iteration.

## 4. EVALUATION

This section first evaluates the product version of the Java HotSpot™ client compiler that was described in Section 2. Section 4.7 then presents benchmark results for the research projects that were described in Section 3. In general, the impact of the just-in-time compiler is difficult to isolate because other parts of the virtual machine, such as the garbage collector and the runtime library, affect the overall performance. However, it is possible to draw conclusions from the comparison of the Java HotSpot™ client and server VMs, because they share the same runtime library and large parts of the virtual machine.

### 4.1 Benchmark Methodology

We evaluate the JVMs using three standard benchmarks that cover different areas: SPECjvm98 [SPEC 1998] evaluates the performance for client application, SPECjbb2005 [SPEC 2005] for server applications, and SciMark 2.0 [Pozo and Miller 1999] for mathematical computations.

The SPECjvm98 benchmark suite consists of seven benchmarks derived from real-world client applications. They are executed several times and the slowest and the fastest runs are reported. The slowest run, usually the first one, indicates the startup speed of a JVM, while the fastest run measures the peak performance after all JVM optimizations have been applied. The reported metric is the speedup compared to a reference platform.

SPECjbb2005 emulates a client/server application and reports the executed number of transactions per second. Since the default heap size of the Java HotSpot™ client VM is too small for this benchmark, the heap was enlarged to

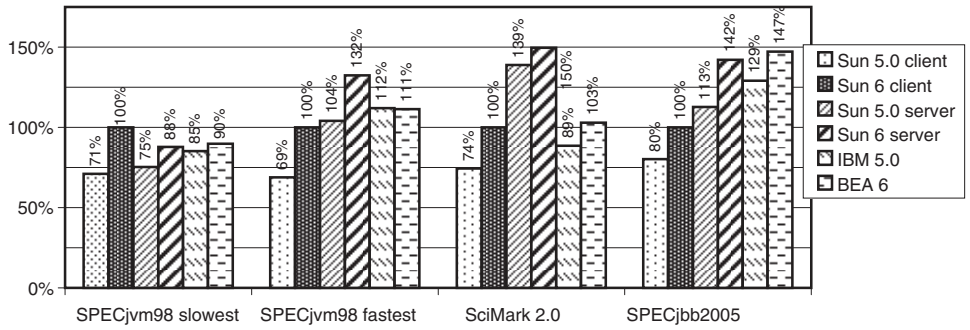



Fig. 10. Out-of-the-box performance comparison of JDKs (taller bars are better).

| | |
|---|---|
| Sun 5.0 client: | Java HotSpot(TM) Client VM (build 1.5.0_12-b04, mixed mode, sharing) |
| Sun 6 client: | Java HotSpot(TM) Client VM (build 1.6.0_02-b06, mixed mode, sharing) |
| Sun 5.0 server: | Java HotSpot(TM) Server VM (build 1.5.0_12-b04, mixed mode) |
| Sun 6 server: | Java HotSpot(TM) Server VM (build 1.6.0_02-b06, mixed mode) |
| IBM 5.0: | IBM J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 Windows XP x86-32 j9vmwi3223-20070426 (JIT enabled)) |
| BEA 6: | BEA JRockit(R) (build R27.3.1-1-85830-1.6.0_01-20070716-1248-windows-ia32, compiled mode) |

Fig. 11. Version numbers of the different virtual machines (output of `java -version`).

512 MB via a VM flag. To make the results comparable, this heap size was also used for all other JVMs, even though their default heap size is sufficient. SciMark 2.0 consists of five numerical kernels that perform floating-point computations on arrays and reports a score in Mflops. We use the flag `-large` because we believe that this better reflects actual scientific applications.

All measurements except the architecture comparisons were performed on an Intel Pentium D processor 830 with two cores at 3.0 GHz and 2 GB main memory, running Microsoft Windows XP Professional. The results of SPECjvm98 and SPECjbb2005 are no approved metrics, but adhere to the run rules for research use. The input size 100 was used for SPECjvm98, and SPECjbb2005 was measured with one JVM instance. All results were obtained using 32-bit JVMs.

### 4.2 Out-of-the-Box Performance

Figure 10 compares the out-of-the-box performance of the Sun JDKs, the *IBM Developer Kit, Java 2 Technology Edition, Version 5.0*, and *BEA JRockit 5.0*. For the Sun JDKs, we compare the client and the server configuration of both the current version JDK 6 (the *new* version) and the previous version JDK 5.0 (the *old* version). The baseline for the comparison is the new client configuration. Figure 11 shows the version numbers of the virtual machines. These were the most recent versions available for download at the time of writing. Because of the different runtime systems and garbage collectors, this is not a comparison of the just-in-time compilers, but reflects the performance a user gets from a JVM.

The new Java HotSpot™ client VM shows the best performance for the slowest run of SPECjvm98. The other JVMs achieve a better peak performance for

the fastest run of SPECjvm98 and SPECjbb2005. In contrast, the server compiler has a high startup overhead for compilation, therefore, the slowest run of SPECjvm98 shows bad performance. This matches the compilation policies of the two configurations: The client compiler performs only high-impact compiler optimizations to achieve low compilation overhead, while the server compiler is a slower, but more aggressively optimizing compiler. The JVMs of IBM and BEA apply a dynamic optimization schema with different levels of compiler optimizations. They rank between the client and the server compiler with respect to the startup performance.

The peak performance of JVMs is an active area of development. The results shown in Figure 10 represent only a snapshot of a particular version on a particular platform, so the results can be different in future versions or on different platforms. It is not possible to conclude from the results that one JVM is better than another.

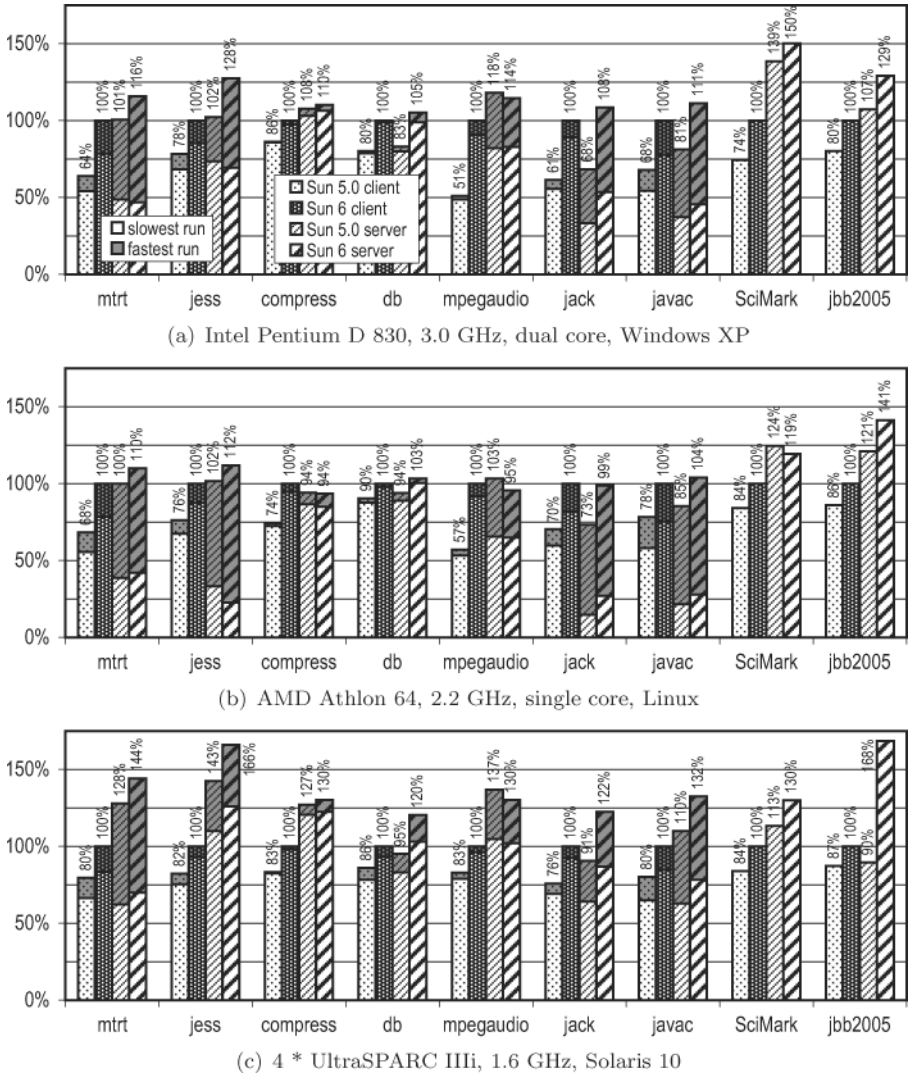
The comparison of the Sun JDK 5.0 and JDK 6 results shows that the client VM has a higher speedup than the server VM: For SPECjvm98, the new client VM is 45% faster than the old client VM, while the new server VM is about 25% faster. This difference is a result of the new architecture of the client compiler. It is also notable that the speedup of the client VM is equal for the slowest and the fastest run of SPECjvm98. The new client compiler reduces the gap in peak performance to the server compiler, but retains the excellent startup behavior.

The IBM JVM shows an unusually low performance for SciMark. We suppose that the optimized compilation of some methods with long running loops comes too late. When executing SciMark without the `-large` flag and, therefore, fewer loop iterations, or when lowering the compilation threshold of the IBM JVM, the performance of the IBM compiler is about 22% better than the new client VM. The performance of the Sun JVMs is not affected by long running loops because of on-stack-replacement (see Section 1.1).

All JVMs offer command line flags that select different garbage collectors, enable additional optimizations, or tune the algorithms for specific types of applications. Enabling such flags can improve the speed of some benchmarks, especially SPECjbb2005, but also degrade the speed of other benchmarks. We believe that the out-of-the-box performance is important because many Java users have neither the time nor the knowledge to discover the optimal tuning flags for their application. Therefore, the numbers of this section do not claim to show the maximum possible peak performance of the JVMs. For such numbers, we refer to the benchmark homepages where vendors submit results for tuned JVMs [SPEC 2005, 1998].

### 4.3 Architecture Comparison

This section compares the old and new Java HotSpot™ client and server compiler using two additional configurations: an AMD Athlon 64 at 2.2 GHz and 1 GB main memory, running Debian GNU/Linux 4.0, representing a single core configuration, and a *Sun Fire V440 Server* with four 1.6 GHz UltraSPARC IIIi processors and 16 GB main memory, running Sun Solaris 10, showing the performance when multiple cores are available.




(a) Intel Pentium D 830, 3.0 GHz, dual core, Windows XP

(b) AMD Athlon 64, 2.2 GHz, single core, Linux

(c) 4 * UltraSPARC IIIi, 1.6 GHz, Solaris 10

Fig. 12. Client and server compiler on different architectures (taller bars are better).

The server VM selects the garbage collection algorithm based on heuristics that include the processor count and the size of the main memory. To eliminate this impact, we forced all configurations to use the same garbage collector. This also affects the result on the dual-core Intel configuration. Therefore, the difference between the client and the server configuration in Figure 12 is lower than the difference in Figure 10.

In comparison to the previous section, the composite result of SPECjvm98 is split into the seven benchmark applications in Figure 12 to show the individual behaviors. The slowest and the fastest runs of SPECjvm98 are shown on top of each other: The grey bars refer to the fastest runs, and the white bars to the slowest. Both runs are shown relative to the same baseline, i.e. the fastest

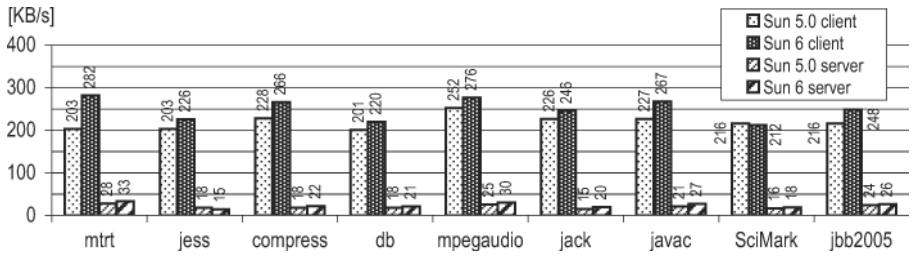



Fig. 13. Compilation speed of the client and the server compiler (taller bars are better).

run of the new client compiler. Therefore, a big grey fraction indicates a high compilation overhead at startup.

Two results can be inferred when comparing the speed of the old and the new client compiler: First, the speedup for the single core AMD configuration is lower than for the dual-core Intel configuration. Biased locking is not effective on single core architectures, because no atomic instructions are necessary. Second, the speedup of the SPARC configuration is lower than the Intel configuration because the new client compiler includes optimizations specifically for the IA-32 architecture. It uses the SSE2 instruction set for scalar floating-point arithmetic that better matches the semantics required by the Java specification.

The gap between the client and the server configuration depends on the architecture and the number of cores. The server compiler benefits from multiple cores, because the expensive compilation can be done in parallel, and from the better instruction scheduling. Therefore, the gap is lower for the AMD configuration and bigger for the SPARC configuration. Similarly, the difference between the slowest and fastest runs of SPECjvm98 is lower when more cores are available.

### 4.4 Compilation Speed

Figure 13 compares the compilation speed of the different compilers, i.e. the time spent in the compiler divided by the physical size of compiled bytecodes. It shows that the new client compiler is up to 40% faster than the old client compiler, and between 9 to 16 times faster than the server compiler.

An analysis of the compilation phases of the old and the new client compiler shows that the time spent in the front end is similar for all benchmarks: The overhead introduced by the new phi functions is compensated by the removal of instructions that load and store local variables.

However, the global register allocation is faster than the old heuristics: The old compiler executes the same code generator twice, first to detect registers that are unused in a method and then to assign these registers to the most frequently used local variables. Peephole optimizations remove unnecessary LIR instructions. The linear scan algorithm for global register allocation replaces these phases and produces better code in less time.

Figure 14 shows the distribution of the compilation phases for the new client compiler. About 33–51% of the time is spent in the front end. This

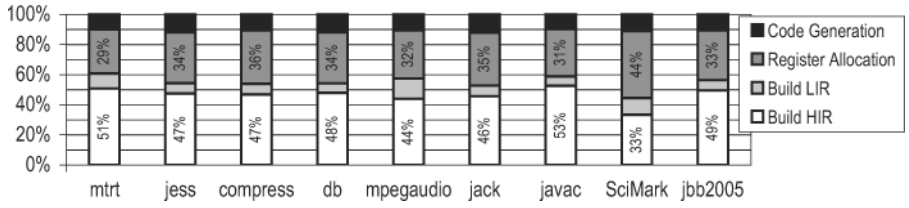



Fig. 14. Compilation phases of the new client compiler.

includes abstract interpretation of the bytecodes and global optimizations on the HIR. Linear scan register allocation takes 29–44% of the time. This includes also the preparation of the meta data for garbage collection and deoptimization. The generation of the LIR and the final code generation are both fast because they are straightforward mapping phases, where one instruction is mapped to one or more other instructions and no optimizations are applied.

The distribution of the compilation phases in the SPECjbb2005 benchmark, about 1/2 of the time for the front end and 1/3 of the time for register allocation, represents the average distribution for most applications. Only for SciMark, the distribution is significantly different: The front end requires less time because only few methods are inlined, and register allocation requires more time because of the high amount of floating-point operations.

### 4.5 Impact of Optimizations

Most compiler optimizations affect both the compilation time and the execution speed of the resulting code. Figure 15 shows the impact of optimizations on the new client compiler when selectively disabling them. The baseline for the relative numbers is the unmodified new client compiler. Note that shorter bars in Figure 15a mean that the execution speed decreases without an optimization, i.e. it increases with this optimization. Shorter bars in Figure 15b mean that the compilation time improves if an optimization is disabled, i.e. it deteriorates if it is enabled.

Null check elimination has a low impact on the execution speed on IA-32 architectures because most null checks are implicitly integrated in a memory access. However, it improves compilation time: The low overhead in the front end for the analysis is superseded by less work in the back end because no meta data must be generated for eliminated null checks. Value numbering, as introduced in Section 2.2, improves some benchmarks by up to 15%, with a fairly low compilation overhead of 5% or below.

The optimizations for the linear scan register allocator that were mentioned in Section 2.4 (optimization of split positions, register hints, and spill store elimination) are effective for scientific applications with a large amount of computations such as SciMark: A 10% slower compilation leads to a 30% faster execution speed. Completely disabling the linear scan-register allocation is not possible. The old heuristics for register allocation have been removed from the source code for Java 6, because they are incompatible with the new SSA form of the HIR.

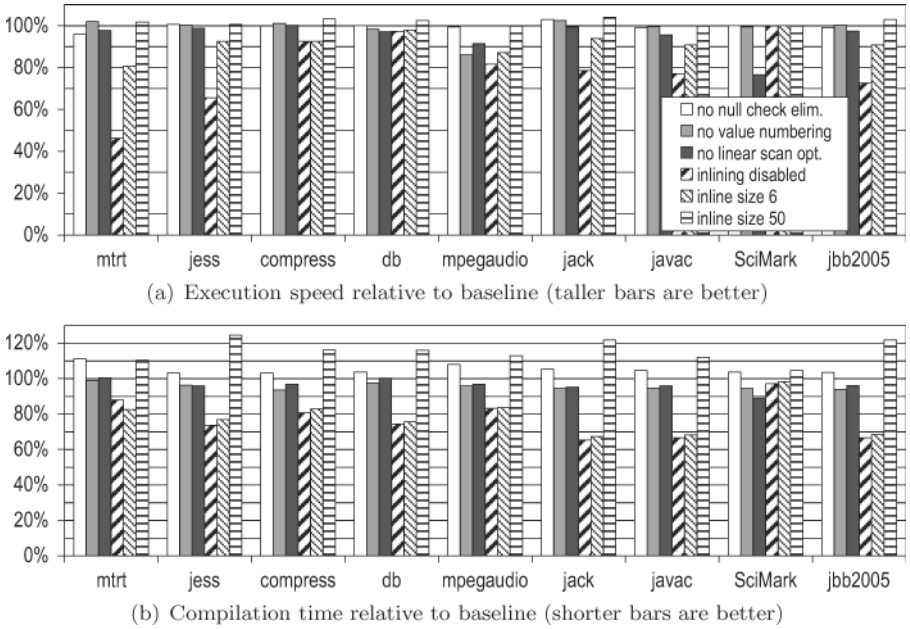



Fig. 15. Impact of compiler optimizations of the new client compiler.

Method inlining is the most expensive, but also the most profitable optimization. The three rightmost bars of Figure 15 show that the inlining strategy described in Section 2.2 is a good compromise: Increasing the inlining size from 35 to 50 bytes degrades compilation time by up to 24% without a significantly better execution speed. Inlining only trivial methods with a maximum size of 6 bytes improves the compilation time, but at the cost of an up to 20% degraded execution speed. Disabling inlining is no reasonable option, because it reduces the execution speed without a further reduction of compilation time.

### 4.6 Summary

Our evaluation showed that the new client compiler achieves good peak performance with remarkably low compilation overhead. Both criteria were improved by the new design: The comparison of the old and the new client compiler shows that the execution speed is 45% higher and that also the compilation speed is up to 40% higher for the popular SPECjvm98 benchmark suite that represents typical client applications.

For JDK 6, not only the client compiler, but also the server compiler, was updated. Both execution and compilation speed improved for most benchmarks. There are no architectural changes or new optimization phases, but internal data structures were optimized. However, the improvements of the server compiler are below the client compiler. For example, SPECjvm98 shows a speedup of about 25% for the server compiler, compared to 45% for the client compiler.

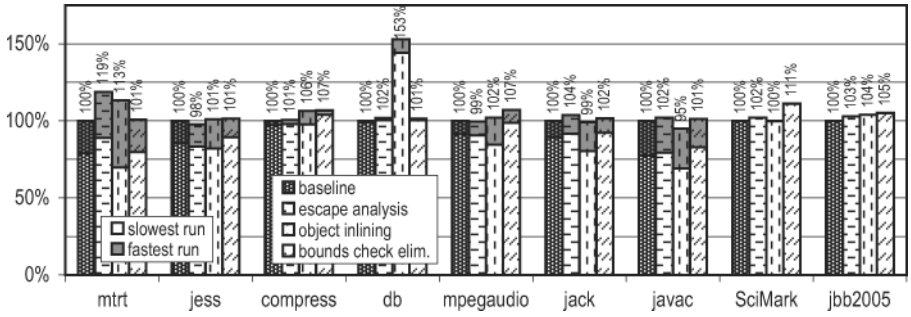



Fig. 16. Benchmark results for ongoing research projects (taller bars are better).

### 4.7 Ongoing Research Projects

Figure 16 presents the benchmark results for the research projects that were described in Section 3. All implementations are based on the client compiler of the JDK 6, but not part of the product version. They are split up in different development workspaces, so the optimizations and the results are independent.

Escape analysis, i.e. scalar replacement, stack allocation, and synchronization removal, optimizes programs that allocate a lot of short-lived objects. The mtrt benchmark, for example, is accelerated by about 19%. Scientific applications usually operate on large arrays that cannot be eliminated. However, the Monte Carlo benchmark that is a part of SciMark makes heavy use of synchronization on thread local objects that can be eliminated by our escape analysis, leading to a speedup of nearly 50% for this benchmark. Escape analysis imposes a compilation overhead, but this overhead is more than outweighed by the additional optimizations, so that even the slowest runs of SPECjvm98 show a speedup.

Object inlining is orthogonal to escape analysis, because it optimizes long-living data structures. The maximum speedup measured is more than 50% for the object-intensive db benchmark. The mtrt and the mpegaudio benchmarks are improved as well. The better peak performance justifies the startup overhead that is caused by the read barriers and the additional compilation of methods.

Array bounds check elimination is especially effective for scientific applications, so there is a high speedup for SciMark that comes close to the theoretical maximum when array bounds checks are completely omitted. The array-intensive mpegaudio benchmark also shows a significant speedup. The compilation overhead is low, so there is no negative impact on the slowest runs of the benchmarks.

## 5. RELATED WORK

This section compares the compilation policies of different production-quality and research Java virtual machines. It summarizes the intermediate representations of their just-in-time compilers and evaluates the suitability for research projects and the availability of the source code.

The IBM Development Kit for Java uses a dynamic optimization framework that consists of an interpreter and a dynamic compiler with different optimization levels [Suganuma et al. 2004]. When the invocation counter of an interpreted method reaches a threshold, the method is compiled at the lowest optimization level with only basic optimizations. Later, a recompilation controller, which uses information from a sampling profiler, recompiles hot methods at higher optimization levels. The source code of this production-quality system is not available, but research results on new adaptive and dynamic optimizations are published [Suganuma et al. 2005].

The compiler uses three different intermediate representations that all have a basic block structure [Ishizaki et al. 2003]: The stack-based *extended bytecode* representation is similar to bytecodes, but augmented with explicit type information. It is used, e.g. for method inlining. The register-based *quadruple* representation uses a tuple format where an operation has several operands. Its operations are platform independent, but platform-dependent code generates different sequences of operations, similar to our LIR. In the *directed acyclic graph*, which uses SSA form, nodes correspond to these operations and edges represent data dependencies. The register allocation mechanism depends on the platform: Either a local heuristic, a linear scan algorithm, or a graph-coloring algorithm is used.

BEA JRockit is a production-quality JVM that uses the compile-only approach [Shiv et al. 2003]: A fast just-in-time compiler generates machine code before a method is executed the first time. Based on sampling profiles, frequently executed methods are recompiled by an optimizing compiler. To our knowledge, the intermediate representations of the compilers are not published and the source code of the VM is not available.

The Jikes RVM, previously called Jalapeño [Alpern et al. 2000], is a research Java VM developed by IBM. Because it is written in Java and available as open source, it is used as the basis for many research project. The Jikes homepage [Jikes 2007] lists more than 150 publications that are related to Jikes. It follows the compile-only approach with different compilers. The baseline compiler assembles code patterns for each bytecode similar to an interpreter. The quick compiler also generates code directly from the bytecodes, but prior to that, it annotates the bytecodes with the results of optimizations.

Jikes' optimizing compiler [Burke et al. 1999] is activated by an adaptive optimization system [Arnold et al. 2000] that uses a sampling profiler to detect hot methods. It uses a high-level, a low-level, and a machine-specific intermediate representation. All three share the same structure: They are register-based $n$-tuple representations, where each instruction consists of an operator and some operands. High-level instructions that are generated from the bytecodes are successively expanded to sequences of simpler instructions when the intermediate representations are lowered. The structure is similar to the LIR of the client compiler, but there is no graph-based intermediate representation such as our HIR. The register allocator uses a linear scan algorithm, similar to our client compiler.

The Open-Runtime Platform [Cierniak et al. 2005] developed by Intel is a research VM targeted not only toward multiple architectures, but also toward

multiple languages. It can run either Java or Common Language Infrastructure (CLI) applications by applying the compile-only approach. It is available as open source from ORP [2007], but this version seems to be quite old and does not reflect the current development version of Intel.

Originally, ORP was designed with two just-in-time compilers: The fast *O1 JIT* [Adl-Tabatabai et al. 1998] does not use an intermediate representation, but generates code directly from the bytecodes. The *O3 JIT* [Cierniak et al. 2000] applies aggressive optimizations and can also use profile information collected by O1 compiled methods. Because ORP offers a flexible compiler interface, different other compilers were developed. An example is the *StarJIT* compiler [Adl-Tabatabai et al. 2003] that generates aggressively optimized code for the Intel IA-32 and Itanium architectures. It uses an intermediate representation based on a control-flow graph and instruction triples, as well as architecture-specific intermediate representations and register allocation algorithms.

The Java HotSpot™ server compiler [Paleczny et al. 2001] focuses on peak performance for long-running server applications. It shares no code with the client compiler, but is embedded in the same runtime system. Methods start being executed by the interpreter that collects profile information. When a method activation counter reaches a threshold, the method is compiled by the server compiler using the profile information. The source code of the Sun JDK that is available for research [Sun Microsystems, Inc. 2006b] and as open source [OpenJDK 2007] includes both the client and the server compiler.

The server compiler uses a graph-based intermediate representation in static single assignment form where both control flow and data flow are modeled by edges between instruction nodes [Click and Paleczny 1995]. The instructions form a "sea of nodes" without explicit block boundaries. The architecture-specific low-level intermediate representation uses the same graph-based structure. Global register allocation is performed by a graph-coloring register allocator.

## 6. CONCLUSIONS

We presented the architecture of the Java HotSpot™ client compiler of Sun Microsystems that is part of the Sun JDK 6. In comparison to previous versions of the JDK, the architecture has changed substantially: The high-level intermediate representation of the front end now uses SSA form, which facilitates data-flow optimizations, such as global value numbering. The back end uses the linear scan algorithm for global value numbering, which produces better code in less time compared to the old heuristics. Our evaluation demonstrates significant speedups for a variety of applications on a multitude of architectures.

We expect that the research of just-in-time compiler optimizations advances in two directions: Feedback-directed optimizations [Arnold et al. 2005] that build on profiling data collected at runtime will allow virtual machines to adapt to the actual workload of applications. Second, the general availability of cheap multicore processors will probably lead to a revival of long-known algorithms for automatic parallelization [Allen and Kennedy 2002], with the challenging problem of making them suitable for just-in-time compilers.

The client compiler is not only a fast production-quality compiler that is ideal for desktop applications, but also an excellent basis for research projects. Exemplary, this paper presented an overview of dynamic optimizations for the client compiler that perform escape analysis, object inlining, or array bounds check elimination. These optimizations make use of the deoptimization framework of the Java HotSpot™ VM, which allows to undo optimistic optimizations if class loading invalidates assumptions of the compiler.

## APPENDIX SOURCE CODE HINTS

The source code of the Java HotSpot™ VM is available as part of the Java SE 6 Source Snapshot Releases [Sun Microsystems, Inc. 2006b]. This appendix provides a link between the concepts presented in our paper and the source code of the virtual machine. The complete VM is contained in one project, which is located in the directory `hotspot/src/` of the download bundle. More details on the source code as well as compilation examples that show the HIR and LIR can be found in Wimmer [2004] and Kotzmann [2005].

The source-code files of the client compiler start with the prefix `c1_`. All instructions of the HIR (see Section 2.1) are subclasses of the class `Instruction` (more than 50 in total). A basic block consists of a linked list of instructions. The first instruction is always of class `BlockBegin` and represents the block itself. The last instruction is a subclass of `BlockEnd`. For historic reasons, the HIR is mostly referred to as IR in the source code. The instructions of the HIR can be processed following the visitor design pattern by implementing a subclass of `InstructionVisitor`.

During abstract interpretation in the `GraphBuilder`, the current values of the local variables and the operand stack are stored in a `ValueStack` object. All instructions that can throw an exception or may cause deoptimization maintain a copy of the value stack. This information is serialized and saved as meta data of the machine code. It allows the reconstruction of an interpreter stack frame when deoptimization is necessary at runtime.

The LIR (see Section 2.3) is constructed by the instruction visitor `LIRGenerator`. The list of LIR operations, which are subclasses of `LIR_Op`, is stored in `BlockBegin`. The LIR operands are encoded efficiently using a mixture of objects and direct representation: Physical registers, virtual registers, and stack slots are encoded as bit fields, whereas addresses and constants are represented as objects because they are too large to be encoded in a single integer value. To allow a consistent use of both kinds, the bit fields are directly encoded as a mock pointer.

Register allocation (see Section 2.4) is implemented in the class `LinearScan`. After this, machine code is created by the class `LIR_Assembler` with the help of a `MacroAssembler`. To separate the platform-independent code from the code for a specific platform, the file name of platform-specific files is marked with a suffix, e.g. `_i486` or `_sparc`.

A tool that shows the control flow and data flow of the intermediate representations as well as the lifetime intervals of the register allocator is available as an open-source project [C1Visualizer 2007]. It parses a log file that is written by

a special debug version of the Java HotSpot™ VM and visualizes all methods that are compiled during the execution of a Java application.

All phases of deoptimization (see Section 2.6) are implemented in the class `Deoptimization`. It converts a physical stack frame, wrapped by the class `frame`, to multiple virtual frames of the class `vframe`, which represent invocations of individual Java methods. To interpret the layout of a physical frame and to determine the inlined methods that it combines, the deoptimization framework uses the meta data created by the compiler, which consists of `ScopeValue` objects.

## ACKNOWLEDGMENTS

We would like to thank Robert Griesemer and Srdjan Mitrovic for their initial design of the client compiler and for establishing and supporting the long-standing research collaboration between Sun Microsystems and the Institute for System Software, and Thomas Würthinger for the work on array bounds check elimination. We also thank Robert Griesemer, as well as the anonymous reviewers, for helpful comments on the paper.

## REFERENCES

ADL-TABATABAI, A.-R., CIERNIAK, M., LUEH, G.-Y., PARIKH, V. M., AND STICHNOTH, J. M. 1998. Fast, effective code generation in a just-in-time Java compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York. 280–290.

ADL-TABATABAI, A.-R., BHARADWAJ, J., CHEN, D.-Y., GHULOUM, A., MENON, V., MURPHY, B., SERRANO, M., AND SHPEISMAN, T. 2003. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Tech. J. 7,* 1, 19–31.

AGESEN, O., DETLEFS, D., GARTHWAITE, A., KNIPPEL, R., RAMAKRISHNA, Y. S., AND WHITE, D. 1999. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, New York. 207–222.

ALLEN, J. R. AND KENNEDY, K. 2002. *Optimizing compilers for modern architectures*. Morgan Kaufmann Publi., Burlington, MA.

ALPERN, B., ATTANASIO, C.BURLINGTON, MA. R., BARTON, J. J., BURKE, M. G., P.CHENG, CHOI, J.-D., COCCHI, A., FINK, S. J., GROVE, D., HIND, M., HUMMEL, S. F., LIEBER, D., LITVINOV, V., MERGEN, M. F., NGO, T., RUSSELL, J. R., SARKAR, V., SERRANO, M. J., SHEPHERD, J. C., SMITH, S. E., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. 2000. The Jalapeño virtual machine. *IBM Sys. J. 39,* 1, 211–238.

ARNOLD, M., FINK, S. J., GROVE, D., HIND, M., AND SWEENEY, P. F. 2000. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM Press, New York. 47–65.

ARNOLD, M., FINK, S. J., GROVE, D., HIND, M., AND SWEENEY, P. F. 2005. A survey of adaptive optimization in virtual machines. *Proc. IEEE 93,* 2, 449–466.

BACON, D. F., KONURU, R., MURTHY, C., AND SERRANO, M. 1998. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York. 258–268.

BLANCHET, B. 2003. Escape analysis for Java: Theory and practice. *ACM Trans. Programming Languages Syst. 25,* 6, 713–775.

BODÍK, R., GUPTA, R., AND SARKAR, V. 2000. ABCD: Eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York. 321–333.

BOEHM, H.-J., DEMERS, A. J., AND SHENKER, S. 1991. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York. 157–164.

BRIGGS, P., COOPER, K. D., AND TORCZON, L. 1994. Improvements to graph coloring register allocation. *ACM Trans. Programming Languages Sys. 16,* 3, 428–455.

BRIGGS, P., COOPER, K. D., and SIMPSON, L. T. 1997. Value numbering. *Softw. Practice Exp. 27,* 6, 701–724.

BURKE, M. G., CHOI, J.-D., FINK, S. J., GROVE, D., HIND, M., SARKAR, V., SERRANO, M. J., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. 1999. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings of the ACM Conference on Java Grande*. ACM Press, New York. 129–141.

C1Visualizer 2007. *Java HotSpot™ Client Compiler Visualizer.* https://c1visualizer.dev.java.net/.

CHOI, J.-D., GROVE, D., HIND, M., AND SARKAR, V. 1999. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM Press, New York. 21–31.

CHOI, J.-D., GUPTA, M., SERRANO, M. J., SREEDHAR, V. C., AND MIDKIFF, S. P. 2003. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Programming Languages Sys. 25,* 6, 876–910.

CIERNIAK, M., LUEH, G.-Y., AND STICHNOTH, J. M. 2000. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York. 13–26.

CIERNIAK, M., ENG, M., GLEW, N., LEWIS, B., AND STICHNOTH, J. 2005. The open runtime platform: a flexible high-performance managed runtime environment. *Concurrency Comput. Practice Exp. 17,* 5–6, 617–637.

CLICK, C. AND PALECZNY, M. 1995. A simple graph-based intermediate representation. In *Papers from the ACM SIGPLAN Workshop on Intermediate Representations*. ACM Press, New York. 35–49.

COWARD, D. 2006. *What's New in Java SE 6.* http://java.sun.com/developer/technicalArticles/J2SE/Desktop/javase6/beta2.html.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Programming Languages Sys. 13,* 4, 451–490.

DEAN, J., GROVE, D., AND CHAMBERS, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming*. LNCS 952, Springer-Verlag, New York. 77–101.

DETLEFS, D. AND AGESEN, O. 1999. Inlining of virtual methods. In *Proceedings of the European Conference on Object-Oriented Programming*. LNCS 1628, Springer-Verlag, New York. 258–277.

DOLBY, J. AND CHIEN, A. 2000. An automatic object inlining optimization and its evaluation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York. 345–357.

FINK, S. J. AND QIAN, F. 2003. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Los Alamitos, CA. 241–252.

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2005. *The Java Language Specification*, 3rd ed. Addison-Wesley, Reading, MA.

GRIESEMER, R. AND MITROVIC, S. 2000. A compiler for the Java HotSpot™ virtual machine. In *The School of Niklaus Wirth: The Art of Simplicity*, L. Böszörményi, J. Gutknecht, and G. Pomberger, Eds. dpunkt.verlag. Heidelberg, Germany, 133–152.

HÖLZLE, U. AND UNGAR, D. 1994. Optimizing dynamically-dispatched calls with runtime type feedback. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York. 326–336.

HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York. 32–43.

ISHIZAKI, K., TAKEUCHI, M., KAWACHIYA, K., SUGANUMA, T., GOHDA, O., INAGAKI, T., KOSEKI, A., OGATA, K., KAWAHITO, M., YASUE, T., OGASAWARA, T., ONODERA, T., KOMATSU, H., AND NAKATANI, T. 2003. Effectiveness of cross-platform optimizations for a Java just-in-time compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM Press, New York. 187–204.

Jikes 2007. *Jikes RVM*. http://jikesrvm.org/.

Jones, R. and Lins, R. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, New York.

Kawachiya, K., Koseki, A., and Onodera, T. 2002. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, New York. 130–141.

Kawahito, M., Komatsu, H., and Nakatani, T. 2000. Effective null pointer check elimination utilizing hardware trap. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, New York. 139–149.

Kotzmann, T. 2005. Escape analysis in the context of dynamic compilation and deoptimization. Ph.D. thesis, Johannes Kepler University Linz.

Kotzmann, T. and Mössenböck, H. 2005. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*. ACM Press, New York. 111–120.

Kotzmann, T. and Mössenböck, H. 2007. Runtime support for optimizations based on escape analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Los Alamitos, CA. 49–60.

Lhoták, O. and Hendren, L. 2005. Runtime evaluation of opportunities for object inlining in Java. *Concurrency Comput. Practice Exp. 17,* 5–6, 515–537.

Lindholm, T. and Yellin, F. 1999. *The Java Virtual Machine Specification*, 2nd ed. Addison-Wesley, Reading, MA.

Mössenböck, H. 2000. Adding static single assignment form and a graph coloring register allocator to the Java HotSpot™ client compiler. Tech. Rept. 15, Institute for Practical Computer Science, Johannes Kepler University Linz.

Mössenböck, H. and Pfeiffer, M. 2002. Linear scan register allocation in the context of SSA form and register constraints. In *Proceedings of the International Conference on Compiler Construction*. LNCS 2304, Springer-Verlag, New York. 229–246.

OpenJDK 2007. *Homepage of the Open-Source JDK Community*. Sun Microsystems, Inc. http://openjdk.java.net/.

ORP 2007. *Open Runtime Platform*. Intel Corp. http://sourceforge.net/projects/orp/.

Paleczny, M., Vick, C., and Click, C. 2001. The Java HotSpot™ server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*. USENIX. 1–12.

Poletto, M. and Sarkar, V. 1999. Linear scan register allocation. *ACM Trans. Programming Languages Sys. 21,* 5, 895–913.

Pozo, R. and Miller, B. 1999. *SciMark 2.0*. http://math.nist.gov/scimark2/.

Printezis, T. and Detlefs, D. 2000. A generational mostly-concurrent garbage collector. In *Proceedings of the International Symposium on Memory Management*. ACM Press, New York. 143–154.

Qian, F., Hendren, L. J., and Verbrugge, C. 2002. A comprehensive approach to array bounds check elimination for Java. In *Proceedings of the International Conference on Compiler Construction*. LNCS 2304. Springer-Verlag, New York. 325–342.

Russell, K. and Detlefs, D. 2006. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, New York. 263–272.

Sedgewick, R. 1988. *Algorithms*, 2nd ed. Addison-Wesley, Reading, MA. 441–449.

Shiv, K., Iyer, R., Newburn, C., Dahlstedt, J., Lagergren, M., and Lindholm, O. 2003. Impact of JIT/JVM optimizations on Java application performance. In *Proceedings of the 7th workshop on Interaction Between Compilers and Computer Architectures*. IEEE Computer Society, Los Alamitos, 5–13.

SPEC 1998. *The SPECjvm98 Benchmarks*. Standard Performance Evaluation Corporation. http://www.spec.org/jvm98/.

SPEC 2005. *The SPECjbb2005 Benchmark*. Standard Performance Evaluation Corporation. http://www.spec.org/jbb2005/.

Suganuma, T., Ogasawara, T., Kawachiya, K., Takeuchi, M., Ishizaki, K., Koseki, A., Inagaki, T., Yasue, T., Kawahito, M., Onodera, T., Komatsu, H., and Nakatani, T. 2004. Evolution of a Java just-in-time compiler for IA-32 platforms. *IBM J. Res. Develop. 48,* 5/6, 767–795.

SUGANUMA, T., YASUE, T., KAWAHITO, M., KOMATSU, H., AND NAKATANI, T. 2005. Design and evaluation of dynamic optimizations for a Java just-in-time compiler. *ACM Trans. Programming Languages Sys. 27,* 4, 732–785.

Sun Microsystems, Inc. 2006a. *The Java HotSpot™ Performance Engine Architecture*. Sun Microsystems, Inc. http://java.sun.com/products/hotspot/whitepaper.html.

Sun Microsystems, Inc. 2006b. *Java Platform, Standard Edition 6 Source Snapshot Releases*. Sun Microsystems, Inc. http://download.java.net/jdk6/.

Sun Microsystems, Inc. 2006c. *Memory Management in the Java HotSpot™ Virtual Machine*. Sun Microsystems, Inc. http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepa per.pdf.

TRAUB, O., HOLLOWAY, G., AND SMITH, M. D. 1998. Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York. 142–151.

UNGAR, D. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM Press, New York. 157–167.

WIMMER, C. 2004. Linear scan register allocation for the Java HotSpot™ client compiler. M.S. thesis, Johannes Kepler University Linz.

WIMMER, C. AND MÖSSENBÖCK, H. 2005. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*. ACM Press, New York. 132–141.

WIMMER, C. AND MÖSSENBÖCK, H. 2006. Automatic object colocation based on read barriers. In *Proceedings of the Joint Modular Languages Conference*. LNCS 4228, Springer-Verlag, New York. 326–345.

WIMMER, C. AND MÖSSENBÖCK, H. 2007. Automatic feedback-directed object inlining in the Java HotSpot™ virtual machine. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*. ACM Press, New York. 12–21.

WÜRTHINGER, T., WIMMER, C., AND MÖSSENBÖCK, H. 2007. Array bounds check elimination for the Java HotSpot™ client compiler. In *Proceedings of the International Conference on Principles and Practice of Programming in Java*. ACM Press, New York. 125–133.

Received December 7 2006; revised May 3, 2007 and August 8, 2007; accepted Septmber 8, 2007