Fast, Effective Code Generation in a Just-In-Time Java Compiler

Ali-Reza Adl-Tabatabai*, Michał Cierniak, Guei-Yuan Lueh,
Vishesh M. Parikh, James M. Stichnoth
Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95052

ABSTRACT

A "Just-In-Time" (JIT) Java compiler produces native code from Java byte code instructions during program execution. As such, compilation speed becomes much more important in a Java JIT compiler than in a traditional compiler, requiring optimization algorithms to be lightweight and effective. We present the structure of a Java JIT compiler for the Intel Architecture, describe the lightweight implementation of JIT compiler optimizations (e.g., common subexpression elimination, register allocation, and elimination of array bounds checking), and evaluate the performance benefits and tradeoffs of the optimizations. This JIT compiler has been shipped with version 2.5 of Intel's VTune for Java product.[1]

1. INTRODUCTION

The Java programming language [10] introduces new challenges to the compiler writer, because of the "Just-In-Time" (JIT) nature of the compilation model. A static compiler converts Java source code into a verifiably secure and compact architecture-neutral intermediate format, called *Java byte codes*. A Java Virtual Machine (JVM) interprets the byte code instructions at run time. To improve run-time performance, a JIT compiler converts byte codes into native code at run time.

Although offline compilation of byte codes into native code is possible, it cannot always be performed, because all Java class files are not guaranteed to be available at the start of program execution. Therefore, a byte code compiler needs to be prepared to execute dynamically at run time, hence the term "JIT compiler." For this reason, overall program execution time now includes JIT compilation time, in contrast to the traditional methodology of performance measurement, in which compilation time is ignored. As a result, it is extremely important for the compiler optimizations to be lightweight and effective. It is also important for the Java JIT compiler to interact with other parts of the system, such as the garbage collector and performance analysis tools (e.g., Intel's VTune [13] tool).

In this paper, we present the design and implementation of a production Java JIT compiler for the Intel IA32 architecture [11,12]. We describe our approach, called *lazy code selection*, for quickly generating good quality IA32 code. The key to the lazy code selection approach is that it generates native IA32 instructions directly from the byte codes, in a single pass. Other than a control-flow graph used for register allocation, the JIT does not generate an explicit intermediate representation. Rather, it uses the byte codes themselves to represent expressions and maintains additional structures that are managed on-the-fly. This is in contrast to other Java JIT implementations which transform byte codes to an explicit intermediate representation [21,16]. We describe our lightweight implementations of several standard compiler optimizations—lightweight in terms of both execution time and auxiliary data structures. We use several benchmark programs to show the impact of the optimizations on overall run-time performance.

The JIT that we describe in this paper, interfaces with the Microsoft JVM from SDK 1.5.1 [20] and is currently being shipped with version 2.5 of the Intel VTune for Java product [13], an application profiling tool for Java. The performance of the Intel JIT is comparable to that of the Microsoft JIT; the running times of several benchmarks, measured in seconds, are summarized in the table below (full results for the benchmarks are presented in Section 4).

	MS JIT SDK 1.5.1	Intel JIT SDK 1.5.1	MS JIT SDK 2.0	Intel JIT SDK 2.0
Backprop	40.94	41.51	31.78	41.72
Compress	3.33	2.72	3.00	2.94
Java Cup	1.29	1.20	0.94	1.00
Go	14.62	11.14	9.35	8.05
JPEG	12.03	11.84	6.61	7.12
Lisp	39.73	40.52	14.42	18.23

* Current affiliation is Oracle Corporation, Redwood Shores, CA.

[1] All third party trademarks, tradenames, and other brands are the property of their respective owners.

The rest of this paper is organized as follows. In Section 2, we describe the details of the code generator and of our optimization algorithms. In Section 3, we describe our technique for tracking the location of object references, so that the code generator can compute the root set of references at garbage collection sites. In Section 4, we present measurements of the effectiveness of the JIT's optimizations. Finally, in Section 5 we present our conclusions.

2. CODE GENERATION DETAILS

Figure 1 shows the five major phases of the Intel JIT. The prepass phase performs a linear-time traversal of the byte codes to collect information needed for the global register allocation and lazy code selection phases, and for implementing garbage collection support. The global register allocation phase assigns physical registers to local variables. The code generation phase generates IA32 instructions using the lazy code selection algorithm described in Section 2.2 and performs several optimizations: common subexpression elimination, array bounds check elimination, peephole optimizations, and frame pointer elimination. The code emission phase copies the generated code and data sections to their final locations in memory. The patching phase fixes up relocations in the emitted code and data sections; for instance, offsets of forward branches, addresses of code labels in switch table entries, and the address of switch tables in the read-only data section. With the exception of the global register allocation phase, all phases are linear in time and space.

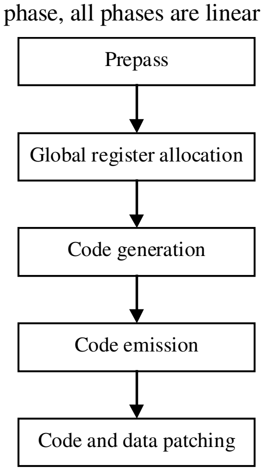


Figure 1: Compiler passes.

2.1 The prepass phase

The prepass phase builds a control-flow graph, and collects three pieces of information: (1) the depth of the Java operand stack at the entry of each basic block; (2) the static reference count of each local variable; (3) the Java operand stack locations containing references at each point where garbage collection may occur; and (4) a list of those variables that alternately hold reference and non-reference values at different points in the method. The stack depth information is needed by the code selector to initialize the locations of operands on the Java operand stack at the beginning of a basic block. The static reference count information is needed by the global register allocator to assign priorities to variables. The information collected for garbage collection allows the JIT to compute the root set of live objects reachable from stack frame locations and from registers. Variables that hold reference and non-reference values are treated in a special way by the garbage collector. We discuss the details of garbage collection in Section 3.

2.2 Lazy code selection

The lazy code selection algorithm is a single pass code selection algorithm. It emits assembled native instructions directly into a temporary code buffer that is later copied by the code emission phase. The code selector also uses a temporary data buffer to assemble read-only constant data, such as floating-point constants and switch tables.

The goal of the lazy code selection algorithm is twofold: (1) to keep intermediate values (i.e., Java operand stack values) in scratch registers, and (2) to reduce register pressure and take advantage of the IA32 addressing modes by folding loads of immediate operands and accesses to memory operands into the compute instructions that use them. Lazy code selection achieves these goals by propagating information about source operands via an auxiliary data structure called the *mimic stack*. The mimic stack simulates the Java run-time operand stack at JIT time: for each byte code's selected instruction sequence, the source operands of the instruction sequence are popped from the mimic stack and the result operand of the instruction sequence is pushed onto the mimic stack.

Instruction operands are modeled in a C++ class hierarchy (Figure 2); the base of this hierarchy is the Operand class. There are four main types of operands: (1) register operands (Register), which are values in physical integer registers and directly addressable by most integer compute instructions, (2) immediate operands (Immediate), which are constant values that can be folded into the immediate fields of integer compute instructions, (3) memory operands (Memory), which are values in memory that can be folded into floating-point or integer compute instructions using one of IA32's memory addressing modes, and (4) floating point operands (FP), which are values on top of the IA32 floating-point register stack. Memory operands are further classified according to the kind of data being accessed: (1) object field references (Field), which use the offset addressing mode (base register plus constant offset), (2) array elements (Array), which use the indexed addressing mode (base register plus scaled index register), (3) static class variables that are not declared as `final` (Static), which use the absolute addressing mode, (4) floating point constants and static class variables that are declared as `final` (Constant), which also use the absolute addressing mode (IA32 floating-point instructions do not have an immediate form), and (5) stack frame locations (Stack), which use the offset addressing with either the stack or frame pointer register as the base. Stack frame locations are used for spilling and for those local variables that are not allocated a register. The JIT eliminates the frame pointer in most cases so that most Stack operands use the stack pointer register as the base register. Frame pointer elimination frees up an additional register for use by the global register allocator, and reduces the number of instructions executed in a method's prolog.

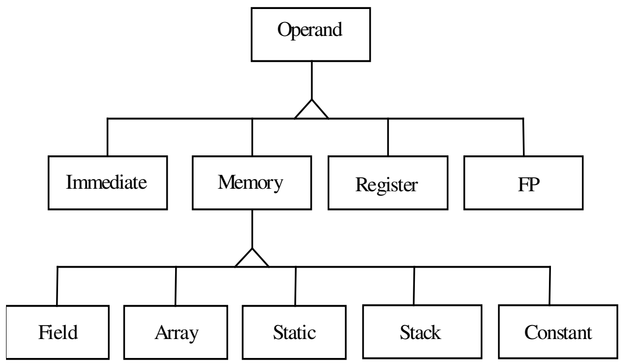


Figure 2: Operand class hierarchy.

To select code for a byte code B that pops source values from the Java operand stack, the code selector first pops the corresponding source operands from the mimic stack, and then tries to fold the source operands into the compute instruction selected for B. If the attempt is successful, then the folded compute instruction is selected. Otherwise, if an operand O cannot be folded into the compute instruction, the code selector selects an instruction that loads O into a scratch register R, and then generates a compute instruction that uses R as the source operand. The result of the compute instruction (which often is a register) is pushed onto the mimic stack to make it available for folding into subsequent instructions. The register manager, which we discuss in Section 2.4.1, handles allocation of scratch registers.

The floating-point registers of IA32 are organized as a stack [11]; a floating-point compute instruction pops one operand from the register stack (the other operand can be a memory operand or another location on the register stack) and pushes its result onto the register stack. This maps perfectly to the Java Virtual Machine's stack-based architecture: whenever an FP operand is popped from the mimic stack, this operand must represent the top of the floating-point register stack. The only complication is that the code selector needs to keep track of the floating-point register stack depth and generate spill code if overflow occurs. The floating-point register stack has only 8 registers but we have found that none of our applications cause floating-point register stack overflow.

At a call site, the code selector generates spills for those operands on the mimic stack that are live across the call site, since the calling conventions consider the FP stack to be caller-saved. Only those operands that may be killed by the call need to be saved; that is, mimic stack operands that are of type Field, Array, Static, FP, and caller-saved Register; operands of type Immediate, Constant, Stack, and callee-saved Register do not need to be spilled.

One problem for the code generator is that the Java operand stack can be non-empty at the entry or exit of a basic block. This condition occurs mainly because of conditional expressions (i.e., question mark colon expressions such as `a>b?a:b`). The problem is that the code generator must guarantee that the operands on the mimic stack are the same at the merge point of two paths.

To guarantee that mimic stack operands are the same at the merge point of several paths, all values that remain on the mimic stack at the end of a basic block are spilled to canonical spill locations in the stack frame. Similarly, if the Java operand stack depth is non-zero at a label (i.e., branch target), then for each Java operand stack location that contains a value, the corresponding mimic stack location is initialized to its canonical spill location. This is the reason the prepass phase computes the Java operand stack depth.

The lazy code selector performs several simple optimizations during code selection. First, if one of the operands of a compute instruction is an Immediate or Constant operand, then the code selector attempts to perform strength reduction (for multiply, divide, and mod operators) or constant folding on the compute instruction. Second, the code selector detects compare followed by branch byte code sequences so that it can generate the corresponding IA32 compare and branch instruction sequence. Third, the code selector performs redundant load-after-store elimination by tracking values loaded into registers; this optimization replaces the use of a memory operand with the use of a register that already contains the memory value.

Although the design of the lazy code selector is tailored to take advantage of IA32's CISC architecture, this style of code selection can also benefit a RISC architecture. Computation instructions in RISC architectures cannot operate directly on memory operands; thus the code selector needs to propagate only register and immediate operands. The benefits for a RISC architecture are that the code selector will eliminate unnecessary moves from registers assigned to local variables and unnecessary loads of immediate operands. Since only register and immediate operands are propagated, an implementation for a RISC architecture may not require an operand class hierarchy.

2.3 Common subexpression elimination

Traditional common subexpression elimination (CSE) algorithms, which are based on data flow analysis [7, 15] and *value numbering* [7, 2], are expensive in both time and space, and we prefer to avoid them in a JIT compiler. We have developed a fast, lightweight CSE algorithm that focuses on common subexpressions within extended basic blocks.

Our CSE algorithm uses the Java byte codes themselves as a compact representation of expressions. Consider the expression "`x+y`". Assuming `x` and `y` are local variables 1 and 2, respectively, the expression's byte code sequence is [`iload_1`, `iload_2`, `iadd`]. Because the byte codes of the `iload_1`, `iload_2`, and `iadd` instructions are 0x1b, 0x1c, and 0x60, respectively, the value 0x1b1c60 represents the expression "`x+y`". Note that the value 0x1b1c60 appears as a subsequence in the byte code instruction stream; as such, the expression can be represented using the pair <*offset,length*>, which we call an *expression tag*. To detect whether the tags <*offset1,n*> and <*offset2,n*> represent the same syntactic expression, we simply compare the subsequences of length *n* starting at *offset1* and *offset2*. Because the maximum size of the stream is 2^{16} [17], an expression tag can be represented concisely using a single word (16 bits for the offset and 16 bits for the length).

To detect common subexpressions, the code selector tracks the expression values held in the scratch registers by annotating each scratch register R with the tag of the expression that R contains. Before selecting code for a byte code B, the code selector looks ahead in the stream to see whether the expression starting from B matches one already associated with a scratch register; if so, it pushes the register onto the mimic stack and skips over the common subexpression in the byte code stream. The registers are checked in decreasing order of subsequence length to match the largest sized expression. To keep the compilation time linear, expression lengths are limited to 16 byte codes, an empirically sufficient limit. If a match is not found, the code selector selects an instruction sequence for the byte code B and updates the expression tag of the register R containing the result of the instruction sequence. The expression tags are initialized to hold no values at the beginning of basic blocks that have branch labels.

There are two ways that the availability of an expression E held in a register R is killed:

1. By instructions that modify the value of R. If register R is a caller-saved register, then a call site kills the availability of E in R; in this case, the expression tag of R is updated to indicate that R contains no value. The availability of E in R is also killed when the register manager allocates R for reuse by the code selector; in this case, the code selector updates the expression tag of R to indicate that it contains a new expression (or no value if R was used to hold a temporary value).
2. By assignments or method calls that (potentially) modify a value loaded by E. E can contain loads of variables, array elements, and object fields, which can be modified by method calls and by assignments. At a method call, the code selector kills the availability of all expressions that contain loads of array elements or object fields. At an assignment, the code selector kills the availability of all expressions that load (or may load) the assigned variable, object field, or array element.

The information about the set of variables and object fields loaded by an expression is held in *kill sets*; there is one kill set associated with each register managed by the register manager. Each variable has a unique index, and each object field has a unique constant pool index. This allows a kill set to be maintained as a bit vector with the first few bits dedicated to variable indices and the rest of the bits dedicated to object field indices. Whenever an object field assignment byte code (i.e., a `putfield` byte code) assigns a new value to an object field with index I, the code selector kills the availability of a register R if the I'th object field index is set in R's kill set bit vector. The code selector performs similar bookkeeping for assignments to variables. To save memory space and compilation time, the size of each kill set bit vector is limited to 256 bits; the code selector gives up on CSE opportunities for indices that fall outside of this limit.

The code selector takes a more conservative approach to killing expressions that load array elements. Rather than performing expensive alias analysis, the JIT takes advantage of the Java feature that there is no aliasing between arrays with different element types. Each register R has a fixed-size bit vector that contains the set of array element types loaded by the expression held in R. When the code selector encounters an assignment to an array element of type T (e.g., an assignment to an integer element), it kills all registers containing expressions that load array elements of type T. This bit vector, in conjunction with an additional bit flag that indicates whether an expression has any object field references, is used by the code selector to detect expressions that are killed by method calls.

Traditional CSE approaches generate temporaries to hold the values of common subexpressions, which may cause high register pressure and introduce more spill code. Our CSE approach does not increase register pressure because the availability of E in R is killed immediately once R is used by the code selector.

Our CSE approach has some limitations. First, it cannot reassociate expressions; for example, "`x+y`" and "`y+x`" are treated as distinct expressions because they have different subsequences. Second, the approach cannot detect expressions that are syntactically different but have the same value; for instance, "`x=w; x+y`" and "`w+y`". Third, the approach can only represent expressions that are contiguous in the byte code stream (no bubble/gap is allowed).

2.4 Register allocation

The IA32 architecture includes only 7 general-purpose integer registers that can be used for register allocation. By convention, the 7 registers are partitioned into 3 caller-saved scratch registers (`eax`, `ecx`, and `edx`) and 4 callee-saved registers (`ebx`, `ebp`, `esi`, and `edi`). The Intel JIT uses the 3 caller-saved registers for local register allocation and the 4 callee-saved registers for global register allocation.

2.4.1 Local register allocation

The local register allocator, or *register manager*, allocates the registers that the lazy code selector uses for expression evaluation. When the code selector requires a scratch register (i.e., a register to hold a temporary expression value), it requests one from the register manager. If there are multiple registers available, the register manager returns the register that was least recently allocated (i.e., a *circular* allocation strategy). This simple heuristic benefits the CSE optimization, described in Section 2.3, by trying to equalize the lifetimes of common subexpressions within scratch registers (i.e., a *fair* policy). However, if the code selector requests a register but none are currently available, the register manager finds the least recently allocated register that can be used, generates code to spill the register to the stack frame, and returns that register. To find the least recently allocated register, the register manager searches the operands on the mimic stack, starting from the bottom-most operand; in this manner, the register manager spills the register with the most distant use in the past. After producing the instruction sequence for evaluating an expression, the scratch registers used in the source operands of the instruction sequence are given back to the register manager.

2.4.2 Global register allocation

The global register allocator allocates the 4 callee-saved registers to local variables within a single method (i.e., no interprocedural register allocation). Global register allocation has been an active area of research resulting in several effective algorithms [6, 3, 4, 8, 5, 22, 9, 19, 18]. A JIT compiler, however, introduces a new challenge—how to balance the cost of running the potentially expensive register allocation algorithm against the expected performance gains. The key is to use an algorithm that is both fast and effective.

The Intel JIT provides two different register allocation algorithms. The first algorithm is extremely simple and cheap to execute: the register allocator allocates the 4 callee-saved registers to the 4 variables with the highest static reference counts. The complexity of the algorithm is $O(B)$, where B is the number of byte codes in the method. This simple allocation scheme is limited, however, because it does not allow two variables with non-overlapping live ranges to share the same register. Thus fewer variables may be allocated registers, and register save/restore costs may increase.

The second register allocation algorithm is similar to the priority-based scheme described by Chow [8] but with two differences: our scheme does not use an interference graph (which is expensive in terms of time and space) and does not perform live range splitting. Priority-based register allocation is effective in allocating registers to the most important variables in a function and can easily take into account call costs [18]. (An alternative linear-time approach that does not consider priorities is described in [23].) Our algorithm is as follows: For each variable *u*, ordered by priority, the allocator performs a backwards depth-first search through the flow graph, starting the search at all basic blocks in which *u* is used (representing a possible end of the live range), and terminating the search at a basic block in which *u* is defined (representing the start of the live range). This depth-first search visits the set of basic blocks covering the entire live range of *u*, and keeps track of the callee-saved registers that are unavailable (i.e., already allocated) in these blocks. If there is a register *R* available in all of these basic blocks, then the allocator assigns *R* to *u* and marks *R* as unavailable in the basic blocks comprising the live range of *u*. The complexity of the algorithm is $O(B+NV)$, where B is the number of byte codes in the method, N is the number of basic blocks, and V is the number of local variables considered for register allocation.

Call cost is an important issue in register allocation [18]. If callee-save cost is not taken into account (i.e., the cost to save and restore a callee-saved register at the prolog and epilog of a function), the register allocator may assign a register with high callee-save cost to a variable with low spill cost. Both register allocation algorithms assign registers to variables if the spill costs are greater than the callee-save cost. In the priority-based approach, the first live range that is assigned a given callee-saved register pays the callee-save cost; the register allocator does not include the callee-costs in the cost benefit analysis of subsequent live ranges that are assigned the same register.

After the global register allocation is completed, there are typically some basic blocks in which not all registers can be allocated; the code selector can use these leftover registers to reduce spill code. Before generating code for a basic block *b*, the code selector first notifies the register manager of the set of callee-saved registers that are available (i.e., not allocated to any variable) in *b* so that the register manager can add these registers to its pool of scratch registers available inside *b*. This has three benefits: First, the register manager generates less spill code. Second, the callee-saved registers are used as spill locations for operands that are live across call sites, thereby reducing the number of stack frame accesses around call sites. Third, more common subexpressions are found because registers containing expression values are live longer.

2.5 Array bounds check elimination

The Java language specifies that all array accesses are checked at run time; an attempt to use an index that is out of bounds causes an exception (`ArrayIndexOutOfBoundsException`) to be thrown. The JIT can eliminate bounds checks if it can prove that the index is always within the correct range, or if it can prove that an earlier check will throw an exception.

The Intel JIT uses a simple mechanism to eliminate bounds checks of indices that are constant. For each Java operand stack location and variable that contains a reference to an array A, the code generator keeps track of the maximum constant bound for which no bounds check is needed for A. This information is updated when a bounds check is generated for a constant index or when an array of constant size is created. For example, if the code selector has already generated bounds checking for A[7], then a subsequent access to A[5] does not require bounds checking. In addition, when the array is created (using the `newarray` byte code), the JIT can use the creation size to eliminate bounds checking on subsequent array accesses. This approach is especially effective during array initialization (e.g., A[0] = A[1] = … = A[9] = 1).

This algorithm is limited in two ways. First, it is applied only locally to each extended basic block, and not globally. Second, only constant operands are used; more bounds checks could be eliminated if symbolic information were used as well.

2.6 Out-of-line exception throws

An array reference in Java must include code to check whether the subscript is within the bounds of the array. The array length is usually stored within the array object, allowing the JIT to inline bounds-checking code. Assuming that the address of the array is in `eax` and the subscript is in `ecx`, the naive code sequence for array bounds checking is as follows.

```
    cmp [eax + offset(length)], ecx
    ja  OK     ; fold two tests into one!
    ...        ; throw an exception
OK:            ; access the array element
```

The code for throwing an exception is infrequently executed (i.e., cold code) and the above implementation has two performance problems for the IA32 architecture:

1. Static branch prediction on Pentium® Pro and Pentium II processors predicts forward conditional branches not to be taken.
2. The exception-throwing portion of the code is likely to be loaded into the instruction cache even though it is unlikely to be executed.

The Intel JIT produces code optimized for the common case of the subscript being within array bounds, with the code after the `notOK` label appearing at the end of the method:

```
    cmp dword ptr [eax+offset(length)], ecx
    jbe notOK
    ...        ; access the array element
    ...

; cold code at end of method’s code space
notOK:
    ...        ; throw an exception
```

2.7 Example

Figure 3 shows a code sequence from the MPEG Player program, a Java applet for viewing MPEG files. We use this example to illustrate the lazy code selection algorithm, common subexpression elimination, out-of-line exception throwing, and register allocation. The first column shows a Java byte code sequence; each byte code is numbered with its index in the byte code stream. The second column shows the native code generated by the Intel JIT. This column also shows the point at which the code selector generates instructions; for instance, the code selector generates the first native instruction when it encounters the third byte code. The third column shows the expression tags held in the scratch registers at each point during code selection. In this example, global register allocation has assigned the `ebp` register to variable 0 (variables 5, 18, and 20 are in memory). The `ebp` register is typically the frame pointer, but for the method shown in this example, the Intel JIT eliminates the frame pointer allowing the global register allocator to allocate `ebp`; accesses to variables 5 and 20 are based off the stack pointer (`esp`). Note that the code for throwing the exception `ArrayIndexOutOfBoundsException` is moved out of line (the code that throws the exception is not shown).

	IA32 native code	eax	ecx	edx
483: aload_0				
484: getfield #34				
487: getfield #45	mov ecx, [ebp+04h]	---	**[483,4]**	---
490: getfield #44	mov edx, [ecx+190h]	---	[483,4]	**[483,7]**
493: lload 18				
495: l2i				
496: aaload	mov eax, [edx+18h]	**[483,10]**	[483,4]	[483,7]
	mov ecx, [esp+128h]	[483,10]	**[493,3]**	[483,7]
	cmp [eax+04h], ecx			
	jbe _throw			
497: getfield #16	mov edx, [eax+ecx*4+08h]	[483,10]	[493,3]	**[483,14]**
500: l2i				
501: istore 5	mov eax, [edx+04h]	**[483,18]**	[493,3]	[483,14]
	mov [esp+160h], eax			
503: aload_0				
504: getfield #34				
507: getfield #45				
510: getfield #44				
513: lload 18				
515: l2i				
516: aaload				
517: getfield #49				
520: i2l	mov ecx, [edx+0Ch]			
	mov edx, ecx			
	sar edx, 1Fh			
521: lstore 20	mov [esp+124h], edx			
	mov [esp+120h], ecx			

Figure 3: Example of lazy code selection and several optimizations in the Intel JIT, excerpted from the MPEG Player code.

The second column illustrates the laziness of our code selection approach. For instance, the code generation of the byte code at index 484 (`getfield #34`) is delayed until the byte code at index 487 (`getfield #45`) and the code generation of the byte code at index 490 (`getfield #44`) is delayed until the byte code at index 496 (`aaload`). In both cases, the code selector delays generating code for loading the object field until the field's value is needed.

At the time when the code selector generates the first native instruction (`mov ecx,[ebp+04h]`), it annotates `ecx` with the expression tag <483,4> to indicate that `ecx` holds the value of the expression formed by the first two byte codes. Notice that the code selector delays generating code for byte codes 490 (`getfield #44`), 493 (lload 18), and 495 (`l2i`), until it encounters byte code 496 (`aaload`). The values of `l2i` and `getfield #44` are kept in `ecx` and `eax`, respectively, and the expression tags of these registers are updated accordingly (<493,3> and <483,10>). As the value of byte code 496 (`aaload`) is loaded into `edx`, the expression tags <483,10> of `eax` and <493,3> of `ecx` are combined to form the expression tag <483,14> of `edx`. When the lazy code selection finishes generating code for byte code 501 (`istore 5`) and scans the next byte code (`aload_0` at index 503), three expressions, <483,18>, <493,3> and <483,14>, are being held in the three scratch registers. The selector attempts to match a CSE by searching the expression tags in the order of <483,18>, <483,14>, and <493,3>. The expression <503,14> matches <483,14> (i.e., a CSE is detected), and the code selector pushes `edx` onto the mimic stack. Then the selector skips byte codes 503 to 516, and continues generating code from byte code 517 (`getfield #49`).

3. GARBAGE COLLECTION SUPPORT

Java is a garbage collected (GC) language, moving the burden of memory management from the programmer to the system (i.e., the JVM). When the program runs low on heap space, the garbage collector determines the set of objects that the program may still access—the live objects—and frees the space used by dead objects. The garbage collector computes the set of live objects by starting with the set of references in global variables, in registers, and on the run-time stack (the root set), and locating all the references that can be reached from the root set by traversing the graph of reachable objects. Some GC algorithms move live objects to a new place in memory [14]. The Intel JIT is designed to work with moving as well as non-moving GC.

Computing the root set requires the cooperation of the JIT, because only the JIT is capable of precisely locating references held by local variables and by temporaries (either of which are assigned to stack locations or registers); the JVM keeps track of which classes have been loaded and thus can determine the set of global variables containing references, without any support from the JIT. The JIT keeps track of Java operand stack locations that contain references by computing a type bit vector for each GC site in the prepass. The type bit vector marks those stack locations that contain references at the GC site. Finding the set of variables containing references, however, is not trivial for the JIT because of "ambiguous types": the same variable may hold reference and non-reference values at different times during the execution of the method. At a GC site the JIT must be able to distinguish between variables containing references and those that do not contain references.

In our development we have experimented with different strategies for detecting variables containing references:

- Maintain a bit mask in a method's stack frame that is dynamically updated on every write to an ambiguously-typed variable. These tags are used at run-time to decide which variables hold references. This approach incurs an overhead of an extra memory reference for every store to an ambiguously-typed variable.
- Statically analyze ambiguously-typed variables using global data flow analysis, and record the information for each GC site. However, it is not possible in general to analyze all variables, because the specification of the JVM allows the same variable to hold either a reference or a primitive type value at the same byte code instruction [17, Section 4.9.6]. (The program may not reference such a variable at that point, but if it contains a reference, it must still be enumerated at a GC site.) These kinds of variables require other techniques, such as dynamic tagging, to maintain reference information (another solution involves *variable splitting* [1]).

The dynamic approach has a higher run-time cost but a lower JIT-time cost than the static approach, so it is unclear which approach is preferable. However, experiments with our benchmarks show that the incidence of ambiguously-typed variables is small and the choice of a solution to the enumeration problem will likely have a negligible impact on performance.

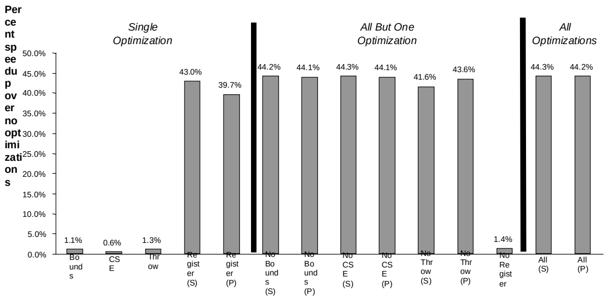
4. EXPERIMENTS

We used several Java benchmark programs to test the effectiveness of individual optimizations described in this paper, as well as combinations of the optimizations. Five of the benchmarks (Backprop from Spec92, and Compress, Go, JPEG, and Lisp from Spec95) were originally C programs, hand-translated into Java. One benchmark, Java Cup, is a parser originally written in Java. We measured the wall-clock time of each application, in seconds, averaged over several runs; the tests were measured on a 233MHz Pentium II with 64 MB of RAM, running Windows NT4.0, Service Pack 3. The Intel JIT is integrated with the Microsoft's JVM in both SDK 1.5.1 and SDK 2.0.

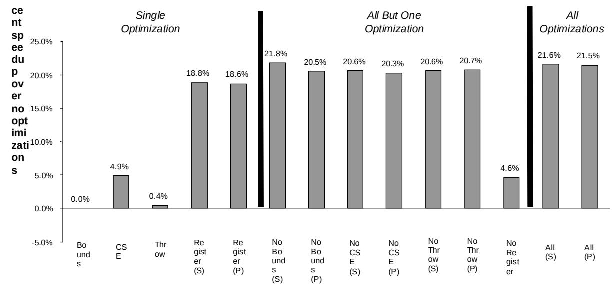
The table below compares the running times of Microsoft's JIT, Intel's JIT with all optimizations disabled, Intel's JIT with all optimizations enabled (using the simple global register allocator described in Section 2.4.2), and Intel's JIT with all optimizations enabled (using the priority-based register allocator described in Section 2.4.2). The optimizations include array bounds check elimination, common subexpression elimination, out-of-line exception throwing, and global register allocation. This table shows that the Intel JIT's performance is comparable with that of the Microsoft JIT.

	SDK 1.5.1				SDK 2.0			
	Microsoft	Intel (no opts)	Intel (all opts, simple allocator	Intel (all opts, priority allocator	Microsoft	Intel (no opts)	Intel (all opts, simple allocator	Intel (all opts, priority allocator
Backprop	40.94	74.52	41.51	41.56	31.78	73.01	41.72	41.39
Compress	3.33	3.46	2.72	2.72	3.00	3.61	2.94	2.96
Java Cup	1.29	1.20	1.20	1.21	0.94	1.01	1.00	1.08
Go	14.62	15.33	11.14	11.33	9.35	11.66	8.05	8.06
Jpeg	12.03	12.81	11.84	11.75	6.61	8.04	7.12	7.12
Lisp	39.73	44.10	40.52	40.84	14.42	18.72	18.23	17.97

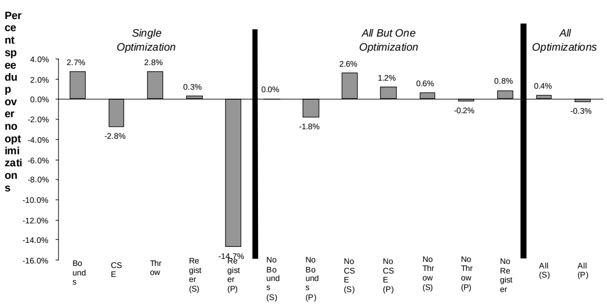
Figures 4 and 5 show the performance improvement from the various optimizations. All values are given as percent improvements over the no-optimization case. Each figure is divided into three categories (separated by dark lines): the effect of a single optimization, the effect of all but one optimization, and the effect of all optimizations. When register allocation is considered, we use "(S)" and "(P)" to denote the simple and priority-based methods, respectively.



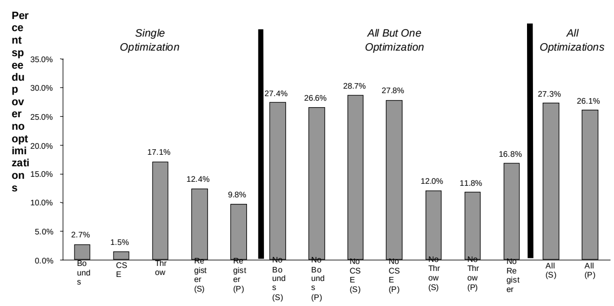
(a) Backprop



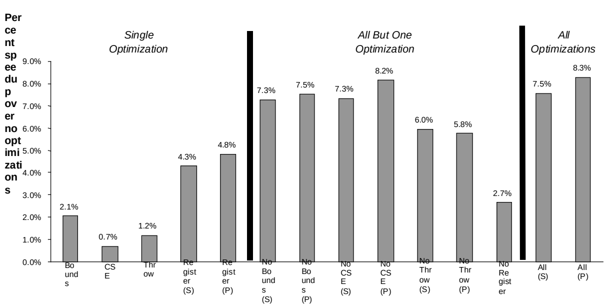
(b) Compress



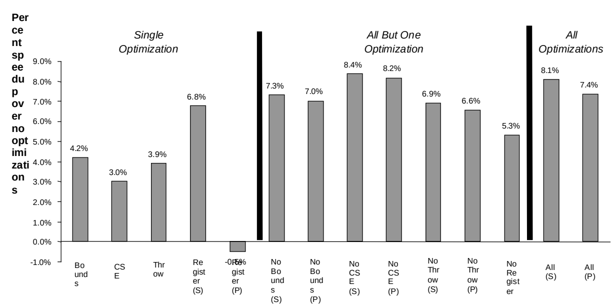
(c) Java Cup



(d) Go



(e) Jpeg



(f) Lisp

Figure 4: Effect of optimizations under SDK 1.5.1.

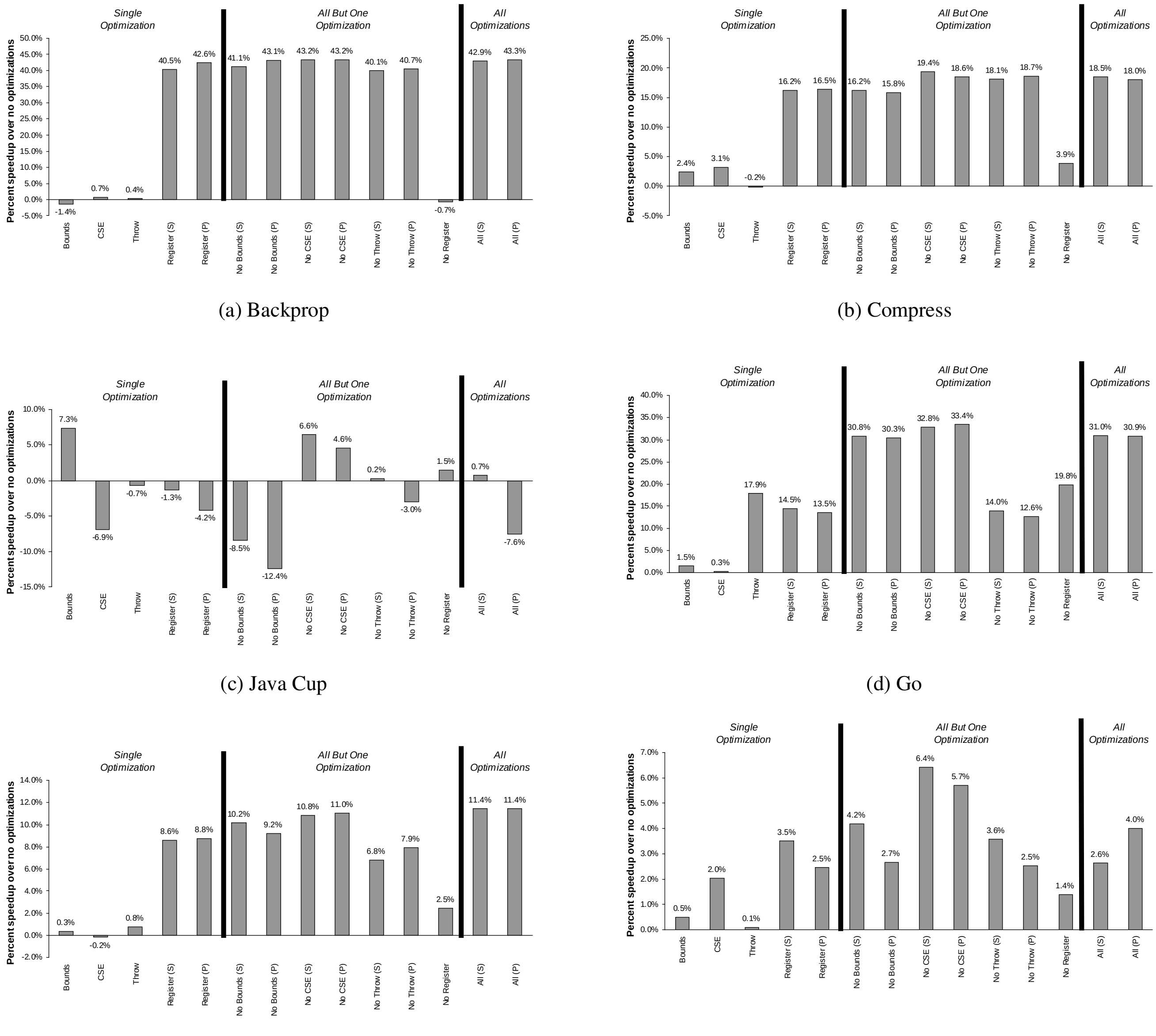
Figures 4(a) and 5(a) show the performance of Backprop. The key feature here is that register allocation is the most important optimization, and that the other optimizations have little effect on the overall performance. Both register allocation algorithms perform roughly equally well. Note that register allocation accounts for a huge gain in overall performance.

Figures 4(b) and 5(b) show the performance of Compress. While CSE and out-of-line exception throwing each contribute to the overall performance, once again it is register allocation that is the most important optimization.

Figures 4(c) and 3(c) show the performance of Java Cup. Note that the performance differences are fairly small, and recall from the table above that the total execution time is only around 2 seconds. In this example, it appears that the costs of executing the CSE and register allocation algorithms (the priority-based method in particular) do not make up for the resulting performance gains in this application.

Figures 4(d) and 5(d) show the performance of Go. Here, bounds check elimination and CSE are ineffective, while register allocation is important. In addition, out-of-line exception throwing is even more important, due to the amount of array bounds checking that must be performed.

Figures 4(e) and 5(e) show the performance of JPEG. As before, register allocation is an important optimization. In addition, CSE and out-of-line exception throwing produce noticeable improvements, while bounds check elimination makes virtually no difference.



(a) Backprop

(b) Compress

(c) Java Cup

(d) Go

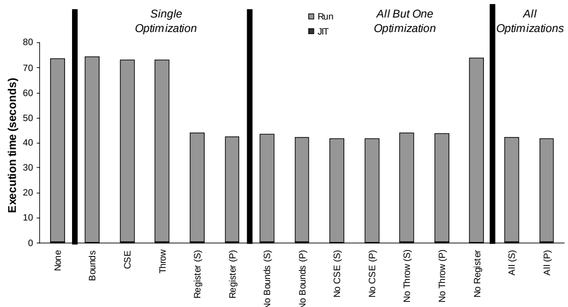
(e) Jpeg

(f) Lisp

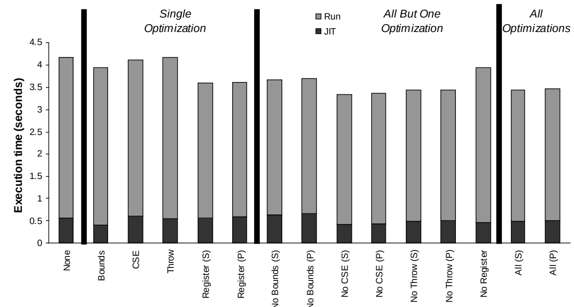
Figure 5: Effect of optimizations under SDK 2.0.

Figure 4(f) and 5(f) show the performance of Lisp. We note that bounds check elimination and out-of-line execption throwing have little effect, whereas CSE costs more to execute than it gains in run-time performance. Particularly interesting is the fact that priority-based register allocation is noticeably more effective than simple register allocation; the reason is that priority-based assigns more variables to fewer registers in at least two frequently-executed methods.

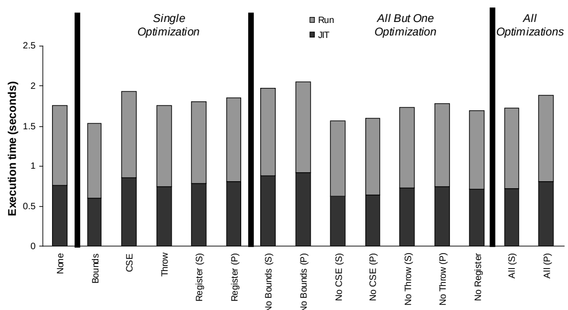
Figure 6 shows the separation of the JIT time and execution time of each program under SDK 2.0 (the results for SDK 1.5.1 are similar). The CSE optimization is more expensive than other optimizations in terms of the JIT time. One noticeable aspect from the figure is that the JIT time of non-optimization is more than the JIT time of turning on bound checking elimination. The reasoning is that our bound checking elimination is fast and avoids generating the internal data structures for the eliminated bound checking instructions that would be needed otherwise. For computation extensive programs (Backprop and Lisp), the JIT time is negligible relative to the running time. For those programs, the JIT compiler can afford to apply aggressive global optimizations, e.g. bound elimination, code hosting, inlining and code scheduling.



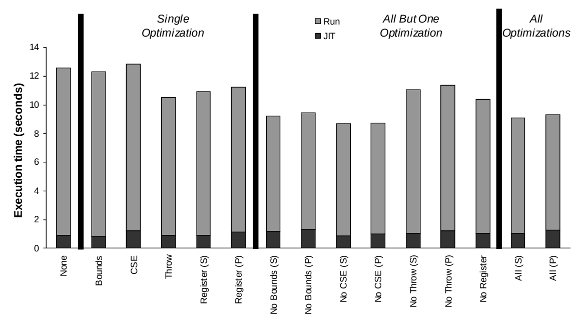
(a) Backprop



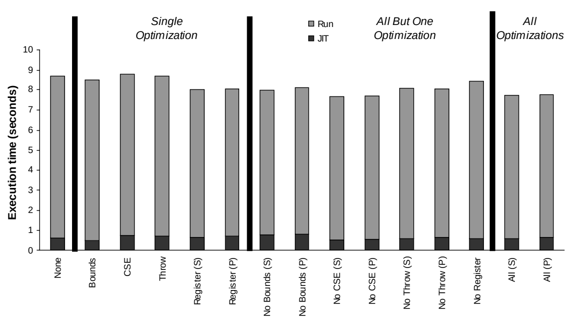
(b) Compress



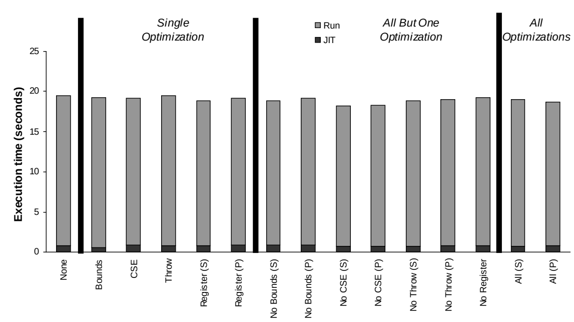
(c) Java Cup



(d) Go



(e) Jpeg



(f) Lisp

Figure 6: JIT time and execution time under SDK 2.0.

5. CONCLUDING REMARKS

In this paper, we have presented the design and implementation of a Just-In-Time Java compiler tailored for the Intel Architecture. We have described *lazy code selection*, the basic code generation strategy used in this JIT. We have shown lazy code selection to be a lightweight, effective way to fold Java stack operands and instructions into addressing modes of IA32 instructions, thus reducing register pressure and allowing more intermediate values to be kept in scratch registers. Lazy code selection is fast (i.e., linear-time), and generates IA32 code directly from Java byte codes.

We have also described lightweight implementations of several standard optimizations, including common subexpression elimination, priority-based global register allocation, and array bounds check elimination. Our optimizations use memory sparingly because they do not use an interim intermediate representation; all optimizations operate directly on the byte codes plus additional data structures that are managed on the fly. Using several benchmark programs, we have shown our optimizations to have various degrees of effectiveness, ranging from small (e.g., CSE) to large (e.g., register allocation). The Intel JIT serves as a framework for the design and evaluation of these and other lightweight optimizations for Java programs.

A Just-In-Time compiler is a critical component of a high-performance Java Virtual Machine implementation. To achieve high performance, not only must a JIT compiler generate high quality code, but it must also be fast because it executes as part of the Java application run-time. The complexity constraints on a JIT compiler are therefore much stricter than a traditional static compiler—a JIT compiler must sometimes trade off the quality of the generated code to achieve better compilation time and smaller memory space requirements. The Intel JIT is representative of this tradeoff: it achieves high performance by being fast and by generating good quality IA32 code.

While the Intel JIT generates good quality code, it also serves as a testbed for implementing and evaluating lightweight optimizations for Java programs. Using this framework, we are able to study the pros and cons of a variety of optimizations for implementation in the next generation JIT compiler.

REFERENCES

[1] O. Agesen and D. Detlefs. Finding References in Java Stacks. Presented at the *OOPSLA'97 Workshop on Garbage Collection and Memory Management*, Atlanta, October 1997.

[2] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.

[3] D. Bernstein, D. Q. Goldin, M.C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R.Y. Pinter. Spill code minimization techniques for optimizing compilers. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 258-263. ACM, July 1989.

[4] P. Briggs, K.D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 275-284. ACM, July 1989.

[5] D. Callanhan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 192-203. ACM, June 1991.

[6] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47-57, January 1981.

[7] F. Chow. *A Portable, Machine-Independent Global Optimizer—Design and Measurements*. PhD thesis, Stanford University, 1984.

[8] F. C. Chow and J. L. Hennessy. A priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12:501-535, Oct. 1990.

[9] D. W. Goodwin and K. D. Wilken. Optimal and Near-Optimal Global Register Allocation Using 0-1 Integer Programming. *Software–Practice and Experience*, 26:930-965, Aug. 1996.

[10] J. Gosling, B. Joy and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[11] Intel Corp. Intel Architecture Software Developer's Manual, order number 243192. 1997.

[12] Intel Corp. Pentium Pro Family Developer's Manual, order number 000900-001. 1996.

[13] Intel Corp. VTune: Visual Tuning Environment. Available at http://developer.intel.com/design/perftool/vtune

[14] R. Jones and R. Lins. *Garbage Collection*. John Wiley & Sons, 1996.

[15] J. Knoop, O. Ruthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 224-234. ACM, June 1992.

[16] A. Krall and R. Grafl. CACAO—A 64-bit Java VM Just-in-Time Compiler. In *Proceedings of the ACM PPoPP'97 Workshop on Java for Science and Engineering Computation*.

[17] T. Lindholm and F. Yellin *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[18] G. Lueh and T. Gross. Call-cost directed register allocation. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 296-307. ACM, June 1997.

[19] G. Lueh, T. Gross, and A. Adl-Tabatabai. Global register allocation based on graph fusion. In *Proceedings of the '96 Workshop on Languages and Compilers for Parallel Computing*, pages 246-265. Aug. 1996. Springer-Verlag.

[20] Microsoft Corp. MS SDK 1.5.1. Available at http://www.microsoft.com/java

[21] Microsoft Corp. MS SDK 1.5.1 JIT Structure. Available at http://www.microsoft.com/java/sdk/151/vendor/vm015.htm

[22] C. Norris and L. L. Pollock. Register allocation over the program dependence graph. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 266-277. ACM, June 1994.

[23] M. Poletto, D.R. Engler and M.F. Kaashoek. tcc: A System for Fast, Flexible, and High-Level Dynamic Code Generation. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 109-121. ACM, June 1997.