Efficient Register Mapping and Allocation in LaTTe, an Open-Source Java Just-in-Time Compiler

Byung-Sun Yang, Junpyo Lee, SeungIl Lee, Seongbae Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioğlu, *Senior Member*, *IEEE*, Erik Altman, *Member*, *IEEE*, and Soo-Mook Moon, *Member*, *IEEE*

Abstract—Java just-in-time (JIT) compilers improve the performance of a Java virtual machine (JVM) by translating Java bytecode into native machine code on demand. One important problem in Java JIT compilation is how to map stack entries and local variables to registers efficiently and quickly, since register-based computations are much faster than memory-based ones, while JIT compilation overhead is part of the whole running time. This paper introduces **LaTTe**, an open-source Java JIT compiler that performs fast generation of efficiently register-mapped RISC code. LaTTe first maps "all" local variables and stack entries into pseudoregisters, followed by real register allocation which also coalesces copies corresponding to pushes and pops between local variables and stack entries aggressively. Our experimental results indicate that LaTTe's sophisticated register mapping and allocation really pay off, achieving twice the performance of a naive JIT compiler that maps all local variables and stack entries to memory. It is also shown that LaTTe makes a reasonable trade-off between quality and speed of register mapping and allocation for the bytecode. We expect these results will also be beneficial to parallel and distributed Java computing 1) by enhancing single-thread Java performance and 2) by significantly reducing the number of memory accesses which the rest of the system must properly order to maintain coherence and keep threads synchronized.

Index Terms—Java virtual machine, just-in-time compilation, register mapping, register allocation, copy coalescing.

1 Introduction

Recently, Java became a prominent programming language for parallel and distributed computing, due to its support for multithreading, networking, CORBA, and remote method invocation [1]. Unfortunately, parallel and distributed Java still has the same performance issue as sequential Java, related to executing Java bytecode. Indeed these performance issues are magnified by the additional synchronization and coherence overhead required in multiprocessor environments. Efficient register allocation, as described in this paper, helps mitigate some of those problems by reducing the number of memory accesses which the rest of the system must properly order.

- *B.-S. Yang, J. Lee, and S. Lee are with Veloxsoft, Inc. and Seoul National University, Dae Rung Post Tower I, 7F, 212-8, Guro-Dong, Guro-Gu, Seoul, 152-050, Korea. E-mail: {sun.yang, walker, seungil}@veloxsoft.com.*
- *S. Park is with Google, Inc., 1600 Ampitheatre Parkway, Mountain View, CA 94043. E-mail: spark@google.com.*
- *Y.C. Chung is with the Information and Communication University, 119 Munjiro, Yuseong-gu, Daejeon 305-732, Korea. E-mail: chungyc@icu.ac.kr.*
- *S. Kim and E. Altman are with the IBM T.J. Watson Research Center, PO Box 218, Yorktown Heights, NY 10598. E-mail: kimsu@us.ibm.com, erik@watson.ibm.com.*
- *K. Ebcioğlu is with the Global Supercomputing Corporation, PO Box 603, Yorktown Heights, NY 10598. E-mail: kemal.ebcioglu@global-supercomputing.com.*
- *S.-M. Moon is with the School of Electrical Engineering, Seoul National University, San 56-1 ShinLim-Dong, KwanAk-Gu, Seoul 151-742, Korea. E-mail: smoon@altair.snu.ac.kr.*

Manuscript received 17 Dec. 2004; revised 15 Nov. 2005; accepted 27 Nov. 2005; published online 28 Nov. 2006.
Recommended for acceptance by R. Eigenmann.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0310-1204.

The Java Virtual Machine (JVM), a software layer to execute bytecode, while providing desirable features such as a "write-once, run anywhere" model for software developers, and security and portability for end-users, does not immediately lend itself to high performance. In order to circumvent the JVM overhead, a technique called Just-in-Time (JIT) compilation [2] is used to implement a JVM. Through JIT compilation, a bytecode method is translated into a native method on the fly, so as to remove the interpretation overhead.

The most important issue in Java JIT compilation is generating efficient code. A critical part of this is how to map and allocate stack entries and local variables into registers effectively. One constraint is that since the JIT compilation time is part of the whole running time, this job should be done quickly. This requires a trade-off between quality of the generated code and speed of mapping and allocating registers for the bytecode, which poses a challenging research and engineering problem beyond a simple register allocation problem.

LaTTe is a freely available JVM and JIT compiler. LaTTe aggressively maps registers for the bytecode, and performs fast register allocation. LaTTe first translates bytecode into pseudocode by mapping all stack entries and local variables to symbolic registers. There will be many copies corresponding to pushes and pops between local variables and the stack in the pseudocode. LaTTe removes most of these copies via efficient register allocation with a local lookahead.

The contribution of this paper is twofold. First, since LaTTe is a working, high-performance JIT compiler whose source code is publicly available, this paper, together with

1045-9219/07/$20.00 © 2007 IEEE Published by the IEEE Computer Society

the source code, can be helpful to readers interested in designing JIT compilers. Second, the present paper shows that LaTTe has made a reasonable trade-off between the quality and the speed of register mapping and allocation: the performance impact and the translation overhead of LaTTe's approach to register allocation are evaluated in detail in the paper. As already noted, these techniques contribute to improved performance in parallel environments by significantly reducing the number of memory accesses which the rest of the system must properly order. These techniques also contribute to parallel and distributed Java computing environments by improving the performance of individual threads.

The rest of the paper is organized as follows: Section 2 briefly reviews the Java VM and our target RISC machine, SPARC, focusing on calling conventions. Section 3 describes the register mapping and the translation of bytecode into pseudo SPARC code. Section 4 describes the real register allocation technique of LaTTe for the pseudocode. A comparison with previous JIT compilation techniques is given in Section 5. Section 6 briefly overviews the LaTTe JVM and its JIT compiler. Section 7 presents our experimental results. A summary follows in Section 8.

2 Java Virtual Machine and SPARC

The Java VM is a typed stack machine [3]. Each thread of execution has its own Java stack where a new activation record is pushed when a method is invoked and is popped when it returns. An activation record includes state information, *local variables*, and the *operand stack*. All computations are performed on the operand stack and temporary results are saved in local variables, so there are many pushes and pops between the local variables and the operand stack.

The calling conventions for a Java method are as follows: The actual parameters are pushed on the operand stack of the caller method before a call is made. In the case of a virtual method call `invokevirtual`, the `this` reference is also pushed as the first parameter. The JVM pops those parameters and moves them into local variables of the callee method in order, starting from local variable zero. When a (nonvoid) Java method returns, the return value is pushed on top of the caller's operand stack.

SPARC is a 32-bit RISC machine with a register-based instruction set [4]. A function has its own *register window* which consists of 24 consecutive integer registers: eight *in* registers (`%i0-%i7`), eight *local* registers (`%l0-%l7`), and eight *out* registers (`%o0-%o7`).[1] When a method is called, the register window is rotated, such that the callee gets a new register window, where the callee's *in* registers overlap the caller's *out* registers. This facilitates argument passing: the caller passes arguments in `%o0-%o5`, which can be retrieved by the callee in `%i0-%i5`. The callee saves the return value in `%i0` which can be retrieved by the caller in register `%o0` when the called method returns. In addition, each method has its own C stack frame in memory, with a reserved 64-byte register-window save area for saving the local registers when a trap is raised; LaTTe uses this for exception handling.

1. LaTTe uses 20 registers for allocation (excluding `%i6, %i7, %o6, %o7`). In the SPARC notation, the destination is the last operand, e.g., "`add %l1, %l2, %l3`" means "`%l3 = %l1 + %l2`" and "`mov %l1, %l2`" means a copy "`%l2 = %l1`."

3 Bytecode Translation with Aggressive Register Mapping

When a method is called for the first time, LaTTe translates its bytecode into SPARC code. In LaTTe, there are two issues in translating bytecode into register-based code. One is converting stack entries and local variables into symbolic registers, which we call *register mapping*. The other is assigning symbolic registers to real registers, which we call *register allocation*. This section deals with register mapping. We will first discuss some JIT compiler design issues pertaining to register mapping, and we will then show how each bytecode is translated.

3.1 Issues in Register Mapping for Bytecodes

There are a few JIT compiler design issues related to register mapping for bytecodes. The JIT compiler designer first needs to decide if registers will be used for stack entries only, or for local variables only, or for both. Obviously, mapping both the stack entries and local variables to registers would be better, but it would require a nontrivial but fast register allocation scheme, which must also be able to remove register copies corresponding to pushes and pops between stack entries and local variables. The JIT compiler designer also needs to decide whether to generate register-allocated code directly from the bytecode in a single pass, or to have a separate pass to generate pseudocode with symbolic registers, followed by real register allocation. The former approach would be faster, yet may constrain register allocation by preallocating fixed registers to some stack entries or local variables, to reduce allocation complexity. The latter would be more versatile in terms of allocating registers and eliminating copies, but it could be slower.

LaTTe uses registers for *all* stack entries and local variables. It also has a separate pass to generate pseudocode in order to allocate registers and remove copies in a highly flexible way. The translation process is composed of four stages. In the first stage, LaTTe identifies all control join points and subroutines (*finally* blocks) in the method's bytecode via a depth-first traversal. In the second stage, the bytecode is translated into a control flow graph (CFG) of pseudo SPARC instructions with symbolic registers. In the optional third stage, LaTTe optimizes the pseudocode. In the fourth stage, LaTTe performs fast register allocation, generating a CFG of real SPARC instructions, which is finally converted into SPARC code. In the remainder of this section, we focus on the second stage and the next section focuses on the fourth stage.

3.2 Translation of Bytecode into Pseudocode

This section describes the translation of key bytecode instructions into SPARC primitives with symbolic registers. The translation rule for each bytecode instruction is determined based solely on the operand types and the opcode of the instruction itself. When this independently generated SPARC code fragment for each bytecode is concatenated with others, the resulting code becomes

correct because consistent formats are used for symbolic registers, especially for those corresponding to stack elements; their format includes information on the current operand stack status, called `TOP` (explained shortly). A symbolic register in the pseudo SPARC code is composed of three parts:

- The first character indicates the type:
  `a` = address (object reference), `i` = integer, `f` = float, `l` = long, and `d` = double.
- The second character indicates the location:
  `s` = operand stack, `l` = local variable, `t` = temporaries generated by LaTTe for translation purposes.
- The remaining number further distinguishes the symbolic register.

For example, `al0` represents a local variable 0 whose type is an object reference. `is2` represents the second item of the operand stack whose type is an integer.

`TOP` is a translation-time variable used by LaTTe (not a value computed at runtime) which indicates the number of items on the operand stack just before translating the current bytecode instruction. For example, if the current value of `TOP` is 4, "`add is{TOP-1}, is{TOP}, is{TOP-1}`" means "`add is3, is4, is3`." There is another translation-time array, `type[1..TOP]` which indicates the type of each item (one of `a, i, f, l, d`) currently on the stack (required for translating `dup/pop`).

LaTTe traverses the bytecode of a method in depth-first order, starting at the beginning of the method with `TOP` set to zero. Following any path of the bytecode, when a bytecode instruction that pushes some item(s) on the stack is encountered, `TOP` is incremented by the number of pushed items. Similarly, when a bytecode instruction that pops some item(s) is encountered, `TOP` is decremented by the number of popped items. The type array `type[]` is appropriately updated by the type of pushed items. According to the JVM specification [3] paragraph 4.9.2, this translation-time computation of the operand stack status is justified, since if the number of items on the operand stack is N on one path from the beginning to a given point, the operand stack must have the same number of items N and the same types of items in the same order on any path arriving at the same point [3]. In fact, the JVM verifier checks if this property is violated during the class loading.

3.2.1 Stack/Local Variable Manipulation Instructions

Due to the stack computation model, bytecode instructions that push a local variable onto the stack or pop the stack top into a local variable are executed frequently. These are translated into symbolic register copies as follows (`$` means a translation-time action, not a runtime action).

```
iload n                 // stack: ... => ..., (local variable n)
  mov   il{n}, is{TOP+1} // means a copy "is{TOP+1} = il{n}"
  $TOP=TOP+1            // the stack now has one more item.
  $type[TOP]='I'        // the type of the new item is integer.

astore n                // stack: ..., (object reference) => ...
  mov   as{TOP}, al{n}
  $TOP=TOP-1            // the stack now has one less item.
```

It should be noted that these symbolic register copy instructions do not really generate code because they will be coalesced during the register allocation phase.

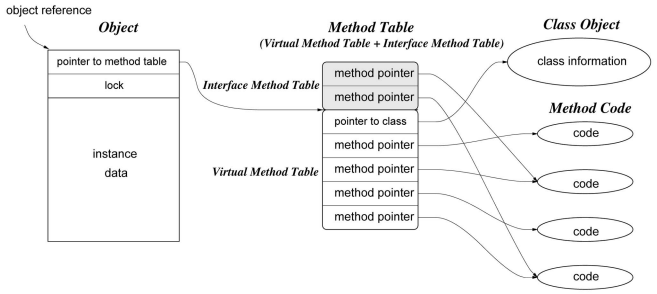


Fig. 1. The object model of **LaTTe**.

3.2.2 Arithmetic/Logical/Shift Instructions

The arithmetic/logical/shift bytecode instructions that operate on the top items of the operand stack can be directly mapped to one or two pseudo instructions.

```
iadd                    // stack: ..., x, y => ..., (x+y)
  add   is{TOP-1}, is{TOP}, is{TOP-1}
                        // means "is{TOP-1} = is{TOP-1} + is{TOP}"
  $TOP=TOP-1            // the stack now has one less item.
```

3.2.3 Object Access Instructions

Fig. 1 depicts the object model of LaTTe. An object includes two fields before the instance data: a pointer to the virtual/interface method table and a 32-bit lock, which are for method invocation and for thread synchronization, respectively. The instance data can be accessed by a single memory access, compared to two accesses used in some implementations of the JDK [3]. Here is an example pseudocode for accessing the integer field `foo` of an object.

```
getfield <x.foo>    // stack: ..., (object ref) => ..., (integer)
  ld     [as{TOP} + foo_offset], is{TOP}
                    // "is{TOP} = load @[as{TOP}+foo_offset]"
  $type[TOP]='I'    // foo_offset is a constant

putfield <x.foo>    // stack: ..., (object ref), (integer) => ...
  st     is{TOP}, [as{TOP-1} + foo_offset]
  $TOP=TOP-2
```

The JVM is required to throw a `NullPointerException` if the object reference is NULL. LaTTe does not generate such check code here because if the object reference is NULL, a SIGSEGV or SIGBUS signal will be raised by the operating system during the execution of the load/store; the LaTTe JVM includes a signal handler where the `NullPointerException` is thrown.

3.2.4 Method Invocation Instructions

The LaTTe JVM maintains a *virtual method table* for each loaded class. The table contains the start address of each method defined in the class or inherited from the superclass. Due to the single inheritance property of Java, if the start address of a method is placed at offset n in the virtual method table of a class, it can also be placed at offset n in the virtual method tables of all subclasses of the class. Consequently, the offset n is a translation-time constant. Since each object includes a pointer to the method table of its corresponding class as shown in Fig. 1, a virtual method invocation can be translated into an indirect function call after two loads, as follows:

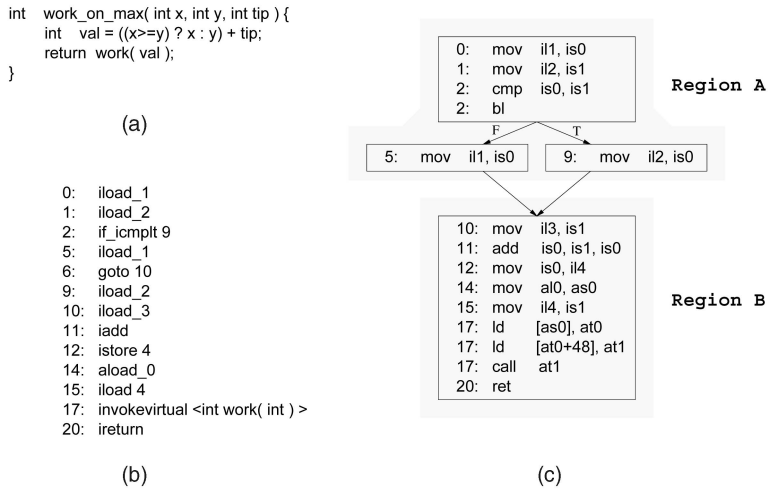


Fig. 2. A translation example from bytecode into pseudo SPARC code. (a) Java source, (b) bytecode, and (c) CFG of pseudo SPARC code.

```
invokevirtual  x.func // assume func takes two integer arguments
                      // and returns an integer
                      // stack: ..., (object ref), (int), (int)
                      //        => ..., (int)
  ld     [as{TOP-2}], at0
  // pointer of the table is located at offset 0 in the object
  ld     [at0 + func_offset], at1
  // pointer of func is located at func_offset in the table
  call   at1
  $TOP=TOP-2          // the stack now has two less items
  $type[TOP]='I'
```

In the above example, the virtual method `x.func` is assumed to have two integer arguments and to return an integer value. At `call at1`, these two arguments and the implicit `this` argument are mapped to symbolic registers `is{TOP}`, `is{TOP-1}`, and `as{TOP-2}`, respectively. Also, the return value is mapped to a symbolic register `is{TOP-2}` when the method returns.

It is desirable to allocate these symbolic registers following the SPARC calling conventions. In our example, the argument registers, `as{TOP-2}`, `is{TOP-1}`, and `is{TOP}`, are preferably allocated into `%o0`, `%o1`, and `%o2`, respectively; otherwise, we should insert copies before the call instruction. Similarly, the return value register `is{TOP-2}` after the call should be allocated into `%o0`.

The calling conventions should also be followed at the callee side. At the beginning of `x.func`, the `this` argument and the two integer arguments are mapped to local symbolic registers `al0`, `il1`, and `il2`, respectively. These registers must be allocated into `%i0`, `%i1`, and `%i2`, respectively. The return value symbolic register, `is0` at the end of the method, must be allocated to `%i0`. Section 4 describes how LaTTe can allocate registers following the calling conventions.

The LaTTe JVM also maintains an *interface method table* for each class which lists the start address of each method implementing an interface method. Each interface method is assigned a globally unique offset so that `invokeinterface` is also translated into an indirect function call after two loads. This is faster than searching the virtual method table although it incurs some space overhead. We have currently seen a maximum of 150 entries in an interface method table.

3.2.5 Array Access Instructions

Arrays in Java are objects. The layout of a LaTTe array object starts with the same two fields as in Fig. 1, followed by the array length and the array data. The JVM is supposed to check array bounds for all array accesses. LaTTe inserts the bound check code based on a trap, as opposed to branches around calls to error routines, in order to simplify control flow. The signal handler takes care of throwing the exception. The check of a NULL array reference is handled by SIGBUS as previously. The translation of `iaload`, for example, is as follows:

```
iaload       // stack: ..., (array ref), (index) => ..., array[index]
    // array bound checking code
    ld     [as{TOP-1} + offset_of_array_length_field], it0
    subcc is{TOP}, it0, g0 // g0 is a global register
                           // that contains zero
    tcc    0x10            // traps if is{TOP} < 0
                           //       or is{TOP} >= array_length
    // load the array element
    // can be removed if we arrange the first element at offset 0
    add    as{TOP-1}, offset_of_array_data, it0
    sll    is{TOP}, 2, it1  // since the sizeof(int) is 4 bytes
    ld     [it0 + it1], is{TOP-1}
    $TOP = TOP -1
    $type[TOP] = 'I'
```

3.2.6 A Translation Example

Fig. 2 shows a simple translation example. The instance method `work_on_max()` in Fig. 2a simply takes the maximum of two values, adds a tip value, and calls another instance method `work()` (whose offset in the method table is 48). Starting from the first bytecode in Fig. 2b with `TOP` = 0, translation of each bytecode will generate the pseudocode in Fig. 2c.

4 Fast Register Allocation

The translation rules described above indicate that it is simple to convert the bytecode into SPARC code with symbolic registers. We now describe our fast register allocator which effectively coalesces copies and conserves registers. The technique is based on the *left-edge* greedy interval coloring algorithm [5], extended to a larger region of code called the *tree region*.

4.1 Tree Regions

The CFG of pseudocode is partitioned into tree regions which are single-entry, multiple-exit subgraphs shaped like trees. Tree regions start at the beginning of the program or at control join points and end at the end of the program or at other join points. For example, the CFG in Fig. 3,

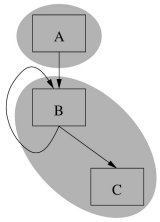


Fig. 3. A CFG of basic blocks and tree regions.

composed of three basic blocks (A, B, C), has two tree regions depicted by shaded areas.

A tree region is a unit of optimizations in LaTTe, such as redundancy elimination, common subexpression elimination, constant propagation, loop invariant code motion, as well as register allocation. Tree regions can be enlarged by code duplication techniques such as loop unrolling to increase the opportunity for optimization in frequently executed parts of code. By working on tree regions, LaTTe trades off quality and speed of optimization.

After regions are constructed for a method, last uses of each symbolic register are computed. Stack symbolic registers are supposed to be dead once they are used, and the live range of temporary symbolic registers cannot span beyond the translated code sequence for a bytecode instruction. Consequently, last uses of these symbolic registers can be readily identified. For local symbolic registers, however, liveness is computed "approximately" via a single postorder traversal [6] of the regions such that every local symbolic register is assumed to be live on a backward edge of the CFG. This gives a conservative, yet fast, estimation of live variables in each region. Based on the liveness information, we can identify the last use of each local symbolic register. When a local symbolic register is dead on one path of a conditional branch while it is live on the other path, we mark the path where it is dead with the last use for the register.

The regions are then register-allocated one by one in a *reverse postorder* traversal of the regions, such that a region is allocated before its descendents are allocated, in a depth-first spanning tree of regions [6]. In *each* region during the traversal, the tree is traversed twice, first by postorder which is called the *backward sweep*, followed by preorder which is called the *forward sweep*. The backward sweep collects information on the preferred destination registers for instructions, which works as a local lookahead. The forward sweep performs real register allocation using that information. During each traversal, a map which is a set of (symbolic, real) register pairs is collected and propagated following the traversal direction. The map is called *p_map* in the backward sweep which describes preferred assignments for destination symbolic registers, and *h_map* in the forward sweep which describes the current register allocation result of symbolic registers.

4.2 Backward Sweep and Forward Sweep

The backward_sweep() algorithm in Fig. 4 is called with the root of the region as an argument. The purpose of the backward sweep is computing the *p_map*, based on the required register assignment at the end of the region, or at method calls/returns according to the calling conventions. For example, if a symbolic register il2 is to be allocated to a real register r at a method call due to the calling conventions, and we have an operation sequence "add is1, is2, is1; mov is1, il2" just before the call, then the destination register is1 of the add is preferably allocated to r to avoid a copy. This preference can be known if the *p_map* propagated through the add includes (is1, r).

Copies are important in computing the *p_map*. If the *p_map* includes (x, r) under a copy "mov y, x," then the *p_map* above the copy includes (y, r). At a conditional branch, the *p_map* of both paths are unioned, yet if there are two different *p_map* for a symbolic register, an arbitrary one is taken. If the destination register of an instruction is included in the *p_map*, its preferred assignment is set to its mapped real register in the *p_map*. Fig. 4 shows this process in detail.

```
backward_sweep (instr)
  returns p_map // set of (symbolic,real) register pairs
{
  p = NULL;
  if (instr is the header of a next region) { // reached a region boundary
      if (there is an old h_map, h_old, saved at instr) // the region boundary has been visited
         p = h_old;  // use the result of a prior forward sweep in other regions
      return p;
  }
  for (each successor s of instr) {
      p = merge_map(p, backward_sweep(s));
  }
  // Now, compute the p_map above instr
  if (instr has a target symbolic register x AND there is a preferred assignment (x,r) in p) {
      preferred_assignment(instr) = r; // if instr is a copy, preferred assignment is not used
      if (instr is a copy x=y ) { insert (y,r) into p; }
      delete (x,r) from p; // (x,r) is not propagated above instr since r is already defined at instr
  } else if (instr is a call) {
      p -= (returned value symbolic register, *);
      p += (argument symbolic register, out register) pairs;
  } else if (instr is a return) {
      p += (returned value symbolic register, return register);
  }
  return p;
}
```

Fig. 4. The backward sweep algorithm.

```
forward_sweep (instr, h, refcount, freereg)
{
  if (instr is the header of the next region) {
     if (instr does not have h_old assigned yet) {
         h_old(instr) = h;
         incremental_backward_sweep(instr); // if other paths reaching instr from the same region exist
     } else Reconcile_h_map(h, h_old(instr)); // reconcile by inserting copies
     return;
  }
  // parse the instruction as "z = x + y"
  real_rhs = "h[x] + h[y]";
  if (x is a last use) {
     if ((refcount[h[x]]--) == 0) freereg += {h[x]};
     delete (x,h[x]) from h;
  }
  if (there is a second operand y != x AND y is a last use) {
     if ((refcount[h[y]]--) == 0) freereg += {h[y]};
     delete (y,h[y]) from h;
  }
  if (instr has a destination register z) { // Now, determine the target register of instr
     if (instr is a copy z = x) real_lhs = real_rhs;
     else if (instr has a preferred assignment r for z AND r is in freereg) real_lhs = r;
     else if (freereg != NULL) real_lhs = first available one in freereg;
     else real_lhs = spill_a_real_register();

     h[z] = real_lhs; refcount[real_lhs]++;
     if (real_lhs != real_rhs) Generate("real_lhs = real_rhs");
  } else Generate("real_rhs");

  for (each successor s of instr) {
     kill_dead_variables(s, h, refcount, freereg);
     forward_sweep(s, h, refcount, freereg);
  }
}
```

Fig. 5. The forward sweep algorithm.

After the preferred assignments for instructions are computed, the forward sweep is performed to allocate real registers. The `forward_sweep()` algorithm in Fig. 5 is called at the root of the region with `h`, an *h_map* that is saved at the root. Other arguments include `refcount` that shows how many symbolic registers are mapped to each real register and `freereg` which indicates the set of real registers to which no symbolic registers map, as determined from `h`. For the starting region of a method, `h` is initialized by the map of parameters. For example, `h` for the method `x.func` in Section 3.2.4 is initialized by `{(al0, %i0), (il1, %i1), (il2, %i2)}`. As regions are allocated in reverse postorder, `h` at the end of a region is propagated to the root of the next region and saved there.

The allocation is performed with a preorder traversal of the tree from the root. When an instruction $z = x + y$ is encountered, the real code is generated as follows. First, the right-hand-side is generated as $h[x] + h[y]$. If the `x` use is the last use of `x`, the `refcount` of the real register `h[x]` is decremented by one, and `h[x]` is added to the `freereg` if the `refcount` becomes zero, and `(x, h[x])` is deleted from `h`, meaning that `x` is now dead. The same is done for `y`. For the target register `z`, if the instruction is a copy $z = x$ and `x` was mapped to a real register `r`, then `z` is also allocated into `r`, meaning that the copy is coalesced. For noncopy instructions, if there is a preferred assignment for the instruction (a real register that `z` will eventually be mapped into) and if it is in `freereg`, we choose the register. Otherwise, we choose the first free register in `freereg`. If `freereg` is empty, we need to spill, which will be described shortly. Now, the pair (`z`, the chosen real register) is inserted into `h`. After the `forward_sweep()` passes through a conditional branch, if some symbolic register `x` is dead on a path, `(x, h[x])` is deleted from `h`, and `refcount` and `freereg` are also updated.

Starting from the root of a region, all instructions are register-allocated as described above. When the root of the next region is encountered, we save the current *h_map* at that root so that the forward sweep at the next region can start with this as an initial *h_map*. Since the root is a join point, more than one forward sweep may reach the same root. If some *h_map* is already saved at the root when the current forward sweep reaches it, we need to reconcile the current *h_map* and the old one that has already been there by inserting some copies, as described below.

4.3 Reconciling *h_map* at Region Join Points

Let us call the old *h_map* and the new *h_map* `h_old` and `h_new`, respectively. Assume $h\_old[x] = h\_old[y] = r$. If $h\_new[x] = h\_new[y] = r'$, we need to insert a copy $r = r'$ on the new incoming edge as shown in Fig. 6. This conserves the old mapping, namely, $h[x] = h[y] = r$.

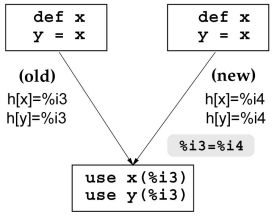


Fig. 6. Reconciling register allocation.

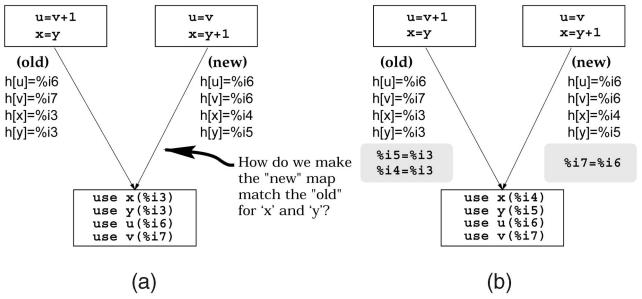


Fig. 7. Reconciling register allocation. (a) Problem and (b) solution.

If h_new[x] is different from h_new[y], however, there is a problem. Suppose in the new mapping, $\texttt{h\_new[x]} = \texttt{r}'$ but $\texttt{h\_new[y]} = \texttt{r}''$. This can happen if there is $\texttt{x} = \texttt{y} + 1$ on the new incoming edge which makes y unequal to x while there is $\texttt{x} = \texttt{y}$ in the old incoming edge, making y equal to x. Fig. 7a depicts this situation. It also shows the opposite case, i.e., $\texttt{u} = \texttt{v} + 1$ is on the old incoming edge while there is $\texttt{u} = \texttt{v}$ on the new incoming edge, which is easier to handle.

As shown in Fig. 7b, it is still possible to reconcile the mapping by inserting copies in the old incoming edge. One issue is that if the region has already been allocated using h_old before h_new reaches the region, we might need to reallocate the region and probably its successor regions, which will be expensive. Fortunately, since we traverse regions in reverse postorder, this can occur only at a loop entry region; when a loop entry region is encountered following the back edge, it would have already been allocated using *h_map* propagated through the loop entry edge.

In order to handle this, when a loop entry region is encountered for the first time, we force each pseudoregister to be mapped to a separate real register by inserting copies (e.g., in Fig. 7a, we insert a copy `mov %i3, %i4` at the old incoming edge and update $\texttt{h[y]} = \texttt{\%i4}$). In this way, when the loop entry is encountered again through the back edge, we do not have to update the previous h of the region nor reallocate the region; we just add copies at the back edge if required.

Reconcilation overhead is, in practice, small due to the backward sweep. Let us assume that region A and C are predecessors of region B, and A is allocated first. The forward sweep at region A will save its *h_map* at the root of region B. Then, the backward sweep at region C will take the saved *h_map* as an initial value of its *p_map* and propagate across region C. So, the forward sweep at region C will generate an *h_map* more compatible with A's, which can reduce reconcilation.

Our algorithm also handles a case when there is more than one edge from region A to region B. In Fig. 2c, for example, when the forward sweep at region A reaches the root of region B for the first time following the false path, we save the current *h_map* at the root. We know the true path from A also reaches the same root but has not yet been forward swept. At this point, we perform an *incremental* backward sweep for the true path to give preferred assignments based on the saved *h_map* from the false path. This will also reduce reconcilation when the forward sweep on the true path reaches the root of region B. Fig. 5 includes the consideration for this case.

The reconciling problem, in fact, is similar to replacing SSA ϕ nodes by a set of equivalent move operations [7] and we can use the same solution to minimize copies.

4.4 Register Spill

When no free registers are available at some instruction I during the forward sweep, we heuristically choose a real register r to spill. Let us assume that r is mapped only to pseudoregisters x and y at that point ($\texttt{h[x]} = \texttt{h[y]} = \texttt{r}$). We insert a store instruction to a spill location "`st x, SPILL0`" just before I and mark x and y last uses there. We then register allocate the inserted store, generating "`st r, SPILL0`" (since $\texttt{h[x]} = \texttt{r}$). We now map the symbolic registers x and y to SPILL0 (i.e., $\texttt{h[x]} = \texttt{h[y]} = \texttt{SPILL0}$) and r is moved back to freereg with its refcount zero. In this way, the forward sweep can continue at I with a new available register r. When a spilled register is used later by an instruction, say "`add x, 2, w,`" we replace the instruction by a new sequence of instructions, [`ld SPILL0, x; mov x, y; add x, 2, w;`] (the copy is needed since both x and y had the same value when spilled), and continue the register allocation. When the load and the copy are register allocated, x and y might be allocated to a different register this time, say r′. Both x and y are mapped to r′, and the refcount of r′ is set appropriately.

At a region boundary, reconciling copies may occasionally include spill locations (e.g., $\texttt{SPILL0} = \texttt{r3}$, $\texttt{r3} = \texttt{SPILL1}$, or $\texttt{SPILL0} = \texttt{SPILL1}$) as well as normal register copies. We handle them appropriately.

4.5 A Register Allocation Example

Fig. 8 describes the register allocation process for the example in Fig. 2. There were two regions in Fig. 2c. The backward sweep and the forward sweep for the region A and the region B are described in Figs. 8a and 8b, respectively.[2] The final register allocation result is shown in Fig. 8c, where only the essential code is generated.

The difference and novelty of our register allocation algorithm compared to the original left-edge interval coloring algorithm [5] are as follows: Our algorithm uses aggressive copy elimination to avoid generating code for copy operations. It maps multiple symbolic registers to the same real register when they are equal, and uses clever heuristics to match physical register assignments across tree region boundaries, in order to avoid introducing copy operations in such boundaries.

5 Comparison with Previous JIT Compilation Techniques

It is highly desirable to be able generate high-performance native code for a bytecode instruction, while keeping the translation process fast.The quantity:

$$\begin{aligned}&(\text{total compilation time for the bytecode})\\ &+ (\text{number of executions of the bytecode})\\ &* (\text{average execution time of the translated bytecode})\end{aligned}$$

2. The incremental backward sweep is not shown because it does not affect the allocation result in this example.

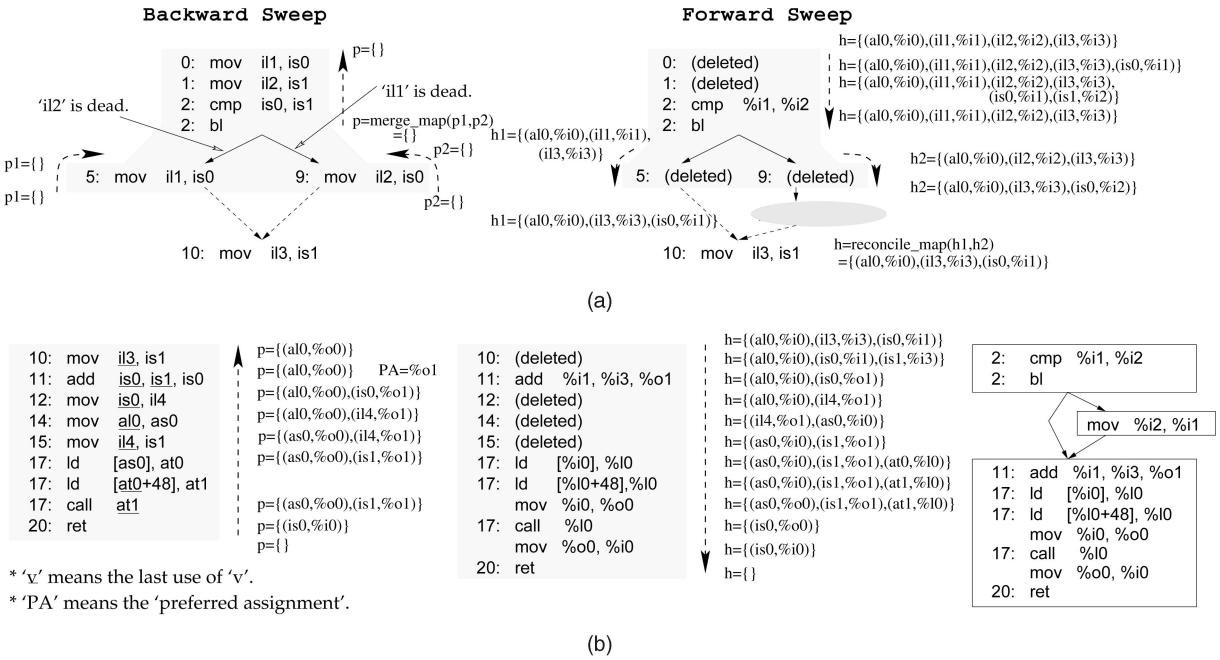


Fig. 8. Register allocation process for the example in Fig. 2. (a) Register allocation of Region A and (b) register allocation of Region B.

must be minimized, in order to reduce the contribution of a bytecode instruction to the total execution time. Hence, finding the right trade-off between translation time and execution time can be very important.

Modern adaptive JIT compilers selectively resort to traditional compiler optimizations which can consume a lot of time, but only for "hot-spot" methods, while interpreting or performing only moderate compiler optimizations on the less frequently executed parts of the program. Indeed, compile time pressure goes away when true hot-spots with very high re-use rates exist. However, continuously detecting the hot-spots accurately and with low overhead can itself be difficult; also, some programs do not have code fragments that are hot enough and worthy of a time-consuming optimization effort. Hence, a base compilation technique similar to LaTTe's, that can already quickly generate high performance native code from the start (along with hardware and OS assistance for accurate profiling), could be helpful for all JIT compilers, including those following a profile-directed adaptive strategy for hot-spots.

In this section, we compare LaTTe's JIT compilation technique with some of those earlier JIT compilation techniques that translate all executed methods, including *Kaffe* [8], *VTune* [9], and *CACAO* [10], focusing on quality and speed of register allocation. We then describe register allocation techniques employed by adaptive compilation techniques.

Kaffe is a public-domain JVM with a relatively simple JIT compiler. Kaffe detects basic blocks and performs single-pass code generation with register allocation (i.e., it generates no pseudocode). For all local variables and operand stack slots, there are corresponding entries in the C stack of the translated method. If a variable or a stack slot is used in a basic block, a register is used to load it from the C stack. At the end of a basic block, registers corresponding to locals or stack slots that have been defined in the basic block are spilled back to the C stack. Consequently, there are many loads/stores in the translated code.

Intel's VTune includes a JIT compiler for its x86 platform, yet the technique itself is applicable to RISC machines as well. All local variables are globally preallocated before the translation starts. Then, single-pass code generation is performed with local register allocation for stack slots and temporaries. A *mimic stack* is computed during the translation to trace the current operand stack which contains registers and the C stack addresses corresponding to local variables and temporaries. Lazy code generation with the mimic stack avoids many copies corresponding to `xload`, yet copies from the operand stack to local variables corresponding to `xstore` are generated. When the mimic stack is not empty at the end of a basic block, all stack entries are spilled to the C stack.[3] Fig. 9b shows the translation process by VTune for our previous example in Fig. 9a. The VTune code can be compared with the LaTTe code shown in Fig. 9d.

CACAO is a JIT compiler targeting the Alpha platform. Each local variable is also preallocated as in VTune, yet for operand stack slots which are live beyond a basic block, *interface* pseudoregisters are allocated instead of spill locations in the C stack. CACAO first converts the bytecode into an intermediate form and analyzes the operand stack to build a *static stack* for each instruction which contains local variables and interface registers (i.e., not real registers). Delayed code generation using the static stack also avoids many copies corresponding to `xload`, yet CACAO can also avoid some copies corresponding to `xstore` if its target local variable can be used as a destination for the computation result at the stack top (e.g., [`iload a; iload b; iadd; istore c;`] can be translated into "`add a, b, c`"). This is possible because CACAO performs more elaborate analysis on the intermediate code. Fig. 9c shows the translation process of CACAO.

The approach of VTune/CACAO based on a simulated operand stack, has two types of inefficiencies compared to

3. Another version of VTune uses priority-based coloring, yet for most benchmarks, it gives worse results [9].

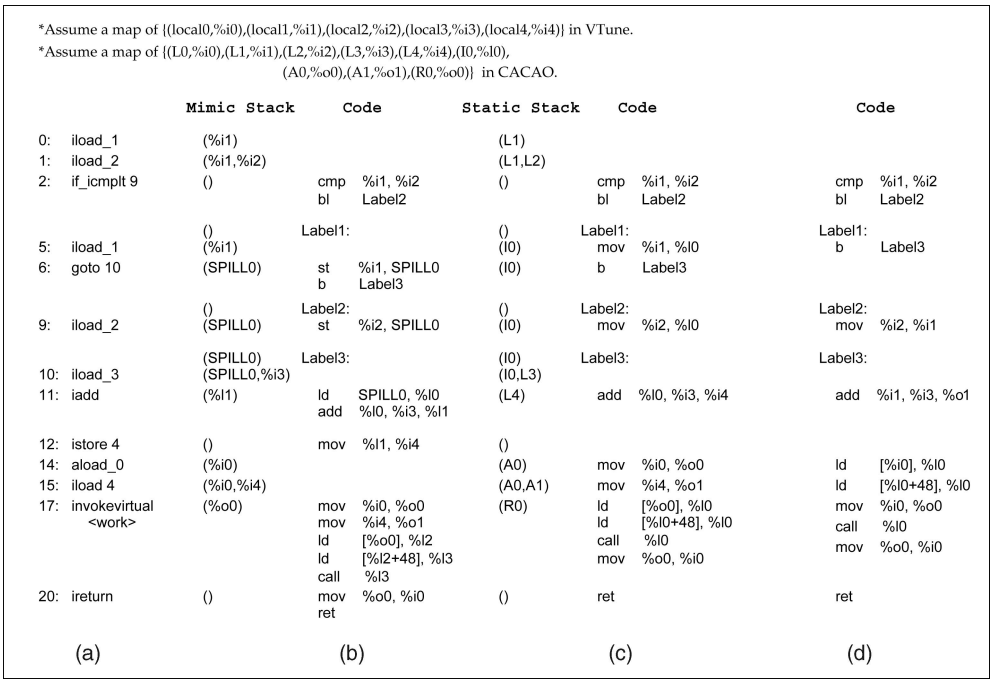


Fig. 9. Translation by VTune and CACAO. (a) Bytecode, (b) VTune, (c) CACAO, and (d) LaTTE.

LaTTe. First, the fixed preallocation of local variables generates inefficient code. In LaTTe, if one local variable is copied into another variable (e.g., through the *x*`load`-*x*`store` sequence), they can be allocated to the same register. This means that LaTTe can conserve registers better and can eliminate more copies than VTune/CACAO. LaTTe can also allocate different registers to different live ranges of a variable, if required. This is hard to achieve in VTune/CACAO because of the fixed preallocation, which might even cause some difficulty in code generation. For example, if there is an update of a variable while its previous value resides in the static/mimic stack due to a previous *x*`load`, the copy for the *x*`load` cannot be avoided. Fig. 10 shows an example for a Java statement `a = b++` where a copy for `iload` cannot be avoided. A copy for `istore` cannot be avoided in VTune, and mostly in CACAO.[4]

Another inefficiency is that VTune/CACAO gives up coalescing at join points. When the mimic/static stack is not empty at a join point, all stack entries are mapped to the C stack/interface registers, always generating spills/copies, respectively. On the other hand, LaTTe resolves join conflicts, coalescing the copy between the stack and the local variable at least for one path. A typical example is the Java condition statement[5] `a = (b > c)?b : c`, in Figs. 8 and 9. For this example, VTune and CACAO generated four and two more operations than LaTTe, respectively.

Many recent JVM JIT compilers employ more elaborate register allocation algorithms due to their adaptive compilation framework. The HotSpot JVM uses interpretation to detect hot spots and then uses a JIT compiler to compile and optimize such hot spots [11], [12]. The JIT compiler uses a global graph coloring allocator based on Briggs' and Chaitin's algorithm with some refinements for allocation speed and code quality.

The Jalapeño JVM [13] and its enhanced open-source version called Jikes RVM [14] employ compile-only adaptive compilation. Each method is compiled by a quick compiler when it is first executed, and then is recompiled by an optimization compiler if it is computationally intensive. The optimizing compiler uses a linear-scan register allocation (LSRA) algorithm [15].

The major differences between LSRA and LaTTe are as follows: First, LaTTe coalesces copies aggressively during register allocation while LSRA does not and focuses on fast register allocation itself. Second, LaTTe employs backward sweep in order to reduce more copies, especially from those caused by calling conventions, yet LSRA does not have such a phase. Finally, the unit for register allocation is tree region in LaTTe, but it is a sequence of instructions in LSRA.

The IBM JIT compiler also uses interpreter-based adaptive compilation, yet its register allocation algorithm is simpler [16]. Frequently used local variables are allocated to physical registers first, and then the remaining registers are used for stack variables. When spilling is needed, the least recently used register is spilled to avoid any complex computation to search spill candidates.

6 The LaTTe JVM and JIT Compiler

The register mapping and allocation techniques comprise the basis of the LaTTe JIT compiler. It also includes other

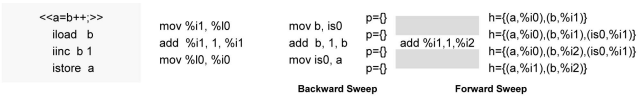


Fig. 10. An inefficient translation example.

4. CACAO's copy elimination for *x*`store` is impossible if preallocation causes nontrue data dependences, e.g., for [`iadd; iload a; istore b; istore a;`], we cannot remove the copy corresponding to "`istore a`" due to "`iload a`."

5. In Java standard class libraries, there are many source files that include a call to `Math.min` or `Math.max`. We found that the corresponding bytecode is not a static method call, rather it is an inlined sequence of bytecode for this conditional form of Java code. Therefore, the conditional form occurs rather frequently.

optimization techniques and is well-coordinated with other JVM components. In this section, we briefly overview the LaTTe JIT compiler and its other JVM components.

There are two versions of the LaTTe JIT compiler: a base version (`-Obase`) and an optimized version (`-Oopt`). The base version performs only the fast register allocation described in Sections 3 and 4 without any other optimizations. The optimized version performs two additional optimizations: "traditional" optimizations and limited object-oriented (OO) optimizations.

For traditional optimization, LaTTe performs common subexpression elimination (CSE), redundancy elimination (RE), loop invariant code motion (LICM), and inlining of static, private, and final methods. Many of these optimizations are performed on a unit of tree region.

LaTTe's OO optimization is primarily for reducing the virtual call overhead of load-load-jump. LaTTe performs two such optimizations: *customization* [17] and *dynamic inline patching* [18], [19]. Customization creates a "specialized" version of a method based on the actual receiver type of a virtual call. With dynamic inline patching, both the inlined version and the load-load-jump sequence are generated, but the inlined version is executed until the target method is overridden.

LaTTe delays the translation of exception handlers until an exception actually occurs [20]. Since an exception would be an "exceptional" event, this reduces the translation overhead and, more importantly, it allows full optimizations when translating the normal flow, without interference from constraints caused by the exception flows (in contrast, many JIT compilers seem to turn off optimization if a method has an exception handler). LaTTe preserves the consistency of register allocation between the exception-causing point and the exception handler during translation.

Java supports monitors, a language-level synchronization construct for multithreading. The LaTTe JVM includes an efficient user-level monitor implementation, called the *lightweight monitor* [21]. A 32-bit word dedicated to representing a *lock* is embedded in each object for efficient lock access (see Fig. 1). The lock manipulation code is highly optimized and is inlined by LaTTe.

Memory management is also crucial to JVM's performance. LaTTe allocates small objects using *lazy worst fit* [22], which usually allocates objects using pointer increments, and uses worst fit to find a new free memory chunk if pointer-incrementing allocation does not work.

LaTTe employs a *partially conservative* mark and sweep garbage collector, in the sense that the runtime stack is scanned conservatively for pointers while all objects located in the heap are handled in a type accurate manner. For the sweep phase, we use *selective sweeping* [23], which sorts all live objects by address and then frees each gap between live objects in constant time.

7 Experimental Results

In this section, we perform an evaluation of LaTTe's JIT compilation technique. In order to evaluate whether LaTTe's sophisticated register mapping and allocation really pays off, we compare the performance of LaTTe's JIT compiler with that of Kaffe's, by implementing both JIT compilers on the same LaTTe JVM. Then, we evaluate how LaTTe allocates registers.

TABLE 1
LaTTe JVM Running Time (Seconds) with LaTTe JIT and Kaffe JIT

Benchmark	Translated Bytes	LaTTe (**Kaffe**) TR	LaTTe (**Kaffe**) TOT	LaTTe (**Base**) TR	LaTTe (**Base**) TOT	Ratio TR (base/Kaffe)	Ratio TOT (Kaffe/base)
SPECjvm98							
_201_compress	24315	0.28	144.06	0.81	69.71	2.89	2.07
_202_jess	45230	0.38	94.57	1.27	41.68	3.34	2.27
_209_db	26414	0.29	138.20	0.88	61.50	3.03	2.25
_213_javac	91963	0.63	150.77	2.38	52.79	3.78	2.86
_222_mpegaudio	38768	0.66	197.68	1.48	79.99	2.24	2.47
_227_mtrt	33998	0.34	106.97	1.07	50.71	3.15	2.11
_228_jack	51208	0.48	88.82	1.68	48.80	3.50	1.82
GeoMean (SPEC)						**3.10**	**2.24**
Java Grande							
Series	8162	0.14	80.11	0.37	57.35	2.64	1.40
LUFact	9393	0.15	13.74	0.40	7.69	2.67	1.79
SOR	8139	0.14	46.62	0.37	28.51	2.64	1.64
HeapSort	8087	0.14	13.68	0.37	8.99	2.64	1.52
Crypt	9264	0.15	24.36	0.39	17.73	2.60	1.37
FFT	8583	0.14	107.80	0.39	63.82	2.79	1.69
SparseMatMult	8148	0.14	78.55	0.37	69.80	2.64	1.13
Search	10505	0.15	121.37	0.43	63.17	2.87	1.92
Euler	22613	0.23	431.32	0.78	192.36	3.39	2.24
MolDyn	24105	0.36	431.85	2.03	44.60	5.64	9.68
MonteCarlo	14487	0.16	123.21	0.52	47.92	3.25	2.57
RayTracer	11020	0.15	230.33	0.41	75.32	2.73	3.06
GeoMean (GRANDE)						**2.96**	**2.04**
Others							
richard_g	7673	0.14	38.27	0.36	17.34	2.57	2.21
richard_gf	7760	0.14	38.71	0.36	16.16	2.57	2.40
richard_gns	7768	0.14	42.26	0.36	21.84	2.57	1.93
richard_dna	8026	0.15	90.74	0.36	33.01	2.40	2.75
richard_dav	8186	0.15	138.04	0.36	79.63	2.40	1.73
richard_daf	8186	0.14	133.76	0.36	47.34	2.57	2.83
richard_dai	8314	0.15	233.30	0.37	78.16	2.47	2.98
richard_all	17189	0.17	662.93	0.52	290.82	3.06	2.28
spell	4505	0.12	27.28	0.23	21.18	1.92	1.29
javacc	125698	0.96	160.98	3.78	42.53	3.94	3.79
deltablue	9429	0.15	7.92	0.40	2.88	2.67	2.75
jjtree	62594	0.39	5.09	1.61	4.04	4.13	1.26
jlex	25785	0.22	3.16	0.75	1.76	3.41	1.80
hashjava	24437	0.23	2.18	0.74	2.51	3.22	0.87
GeoMean (Others)						**2.79**	**2.06**
GeoMean (all)						**2.92**	**2.09**

7.1 Experimental Environment

Our benchmarks are composed of seven SPECjvm98 benchmarks [24], 12 Java Grande benchmarks [25], and 14 nontrivial Java programs we found from the public domain (listed in Table 1 with the translated bytecode size). They are a good mix of integer and floating-point programs.

Our test machine is a SUN Ultra5 270 MHz with 256 MB of memory, running Solaris 2.6, tested in a single-user mode. We ran each benchmark five times and took the minimum running time. In fact, there was little variance in those five running times.

7.2 Evaluation of LaTTe's JIT Compilation Techniques

We modified the LaTTe JVM to use Kaffe's JIT compiler as an execution engine, and compared its performance with that of the **base** version of the LaTTe JIT compiler. Since neither JIT compilers perform any serious optimizations other than the code translation with register allocation, this experiment can evaluate the effectiveness of LaTTe's sophisticated register mapping and allocation, compared against a naive one that maps local variables and stack slots to memory.

Table 1 shows the total running time (TOT)[6] of each JIT configuration with the translation overhead (TR); TR is part of TOT. The table shows that the TOT with LaTTe's JIT is about half of the TOT with Kaffe's JIT. As for the translation overhead, LaTTe's TR is three times larger than Kaffe's TR on average, yet both TRs take a tiny portion of the TOTs.

We also checked the relationship between the translation overhead and the translated bytecode size. Fig. 11 depicts for each benchmark the TR of both JIT compilers and the translated bytecode size shown in Table 1. We can see that LaTTe's TR grows much faster than Kaffe's since LaTTe requires more compilation passes with elaborate analysis.

6. TOT means the total elapsed time, which is not comparable with a SPECjvm98 metric.

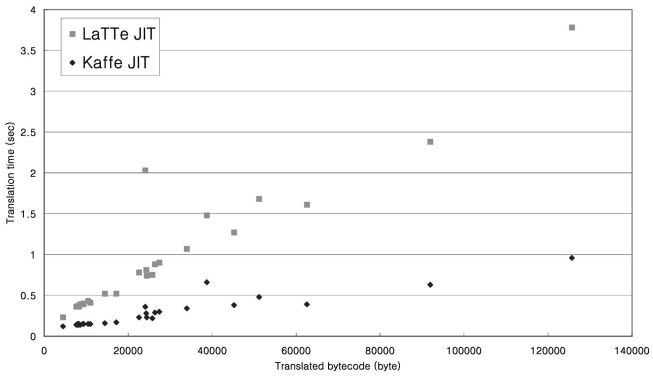


Fig. 11. Translation overheads and translated bytes of LaTTe JIT and Kaffe JIT.

However, LaTTe's TR still increases almost linearly[7] to the translated amount of bytecode, as Kaffe's TR does.

The results in this section indicate that LaTTe's sophisticated register mapping and allocation really pay off without causing a big translation overhead. In order to complete the evaluation, however, it would be desirable to compare with register-mapping JIT compilers such as CACAO or VTune. Unfortunately, it would be extremely difficult to implement and tune them completely on the same framework and to make a fair comparison.

7.3 Speed and Quality of LaTTe's Register Mapping and Allocation

Although LaTTe's JIT compilation overhead is higher than that of Kaffe's, Table 1 indicates that LaTTe's translation overhead is reasonable since it takes a tiny portion of the total running time; for all benchmarks except for `javacc`, TR consistently takes only one or two seconds when TOT takes several tens of seconds. Since register mapping and allocation is the main contributor to TR in the base LaTTe (our experiments show that it takes 67 percent of TR, on average), this means LaTTe's register mapping and allocation is reasonably fast.

In order to examine how LaTTe allocates registers, we measured for each method the "peak" number of real registers (including spill locations) used during its register allocation. This is measured by tracing the number of live real registers mapped to some symbolic registers in the *h_map* during the forward sweep. Comparing this number with worst-case register requirements or minimum requirements when preallocating local variables will be helpful in evaluating LaTTe.

For the base LaTTe, Table 2 shows the peak number (denoted by `M`) for the top five methods with the highest bytecode execution counts in each benchmark (which comprises 57 percent of the total bytecode execution counts on average), along with the number of local variables (denoted by `L`). The table also includes the number of stack entries used (denoted by `S`) and the number of temporary registers used (denoted by `T`), at some point in the method where $\mathtt{S}+\mathtt{T}$ is maximum. The sum $\mathtt{L}+\mathtt{S}+\mathtt{T}$ thus is the worst-case register requirement. If local variables are preallocated as in CACAO, $\mathtt{L}+\mathtt{T}$ is the minimum number of real registers required (for VTune, some nonoverlapping local variables can be allocated to the same register via limited live range analysis).

TABLE 2
Register Mapping and Allocation Quality of the Base LaTTe for Top Five Frequent Methods

Top Five Frequent	1st method				2nd method				3rd method				4th method				5th method			
	M	L	S	T	M	L	S	T	M	L	S	T	M	L	S	T	M	L	S	T
_200_check	<	8	7	3	1	<	16	13	5	0	<	4	2	2	1	<	12	10	5	0
_201_compress	<	11	10	6	0	<	10	7	5	0	<	10	5	6	1	<	4	1	5	0
_202_jess	<	18	16	5	0	<	7	4	3	1	<	4	3	4	0	<	6	4	3	1
_209_db	<	12	11	2	1	<	14	9	3	3	<	6	2	5	0	<	6	4	5	0
_213_javac		7	4	3	0	<	8	7	3	1	<	5	1	5	0	<	11	8	3	1
_222_mpegaudio	<	15	13	4	1	#	20	31	5	1	<	10	7	5	0	<	9	6	4	0
_227_mtrt	<	13	9	4	1	<	6	4	3	1	<	7	5	4	0	<	1	1	1	0
_228_jack	<	4	1	4	0	<	16	13	5	0	<	18	14	6	1	<	8	7	3	1
Series	<	17	15	8	0	<	6	6	6	0	<	13	4	12	0	<	8	7	3	1
LUFact	<	18	13	9	0	<	15	10	6	1	<	15	9	8	1	<	18	15	10	0
SOR	<	22	18	8	1	<	8	4	6	0	<	5	3	5	0	<	11	6	6	0
HeapSort	<	8	5	4	1	<	9	5	4	1	<	8	4	6	0	<	7	3	4	1
Crypt	<	18	15	4	1	<	7	3	4	1	<	7	4	3	1	<	8	7	3	1
FFT	<	39	35	6	0	<	15	11	4	1	<	8	4	6	0	<	11	7	6	0
Sparse	<	16	9	8	1	<	8	4	6	0	<	7	2	5	1	<	5	3	5	0
Search	<	7	6	3	0	<	20	20	5	0	<	10	5	5	1	<	8	5	4	1
Euler	<	17	9	10	0	<	26	19	8	1	<	12	5	9	1	<	23	14	11	0
MD	*	55	52	6	0	<	8	1	8	0	<	7	3	5	0	<	9	5	6	0
MC	<	8	4	6	0	<	10	2	9	0	<	12	9	6	0	<	7	2	7	0
RayTracer		8	2	6	0	<	15	9	7	0	<	7	3	5	0		11	7	4	0
richards_g		4	2	0	2	<	5	3	3	0	<	6	5	3	1	<	5	2	3	1
richards_gf		4	2	0	2	<	5	3	3	0	<	6	5	3	1	<	5	2	3	1
richards_gns		2	1	0	1	<	3	2	4	0	<	6	5	3	1	<	5	3	3	0
richards_dna		2	1	1	0	<	3	2	3	0		2	1	0	1	#	6	7	3	1
richards_dav		2	1	1	0	<	3	2	3	0		2	1	0	1	#	6	7	3	0
richards_daf		2	1	1	0	<	3	2	3	0		2	1	0	1	#	6	7	1	2
richards_dai		2	1	1	0	<	3	2	3	0		2	1	0	1	#	6	7	3	0
richards_all		4	2	0	2		4	2	0	2		2	1	0	1	<	3	2	4	0
spell	<	12	11	2	1	<	16	13	5	0	<	8	7	3	1	<	6	3	5	0
javacc	<	8	7	2	0	<	11	8	3	1	<	8	7	3	1	<	6	2	5	0
delta	<	6	2	5	0	<	3	3	2	0		3	1	1	1		3	1	1	1
jjtre	<	16	13	5	0	<	6	5	6	0	<	7	4	5	0	<	4	3	3	0
jlex	<	10	10	3	0	<	6	2	5	0	#	7	4	0	4		8	6	2	0
hash	<	8	7	3	1	<	10	7	6	0	<	11	8	3	1	<	5	1	5	0

We can find from the table that `M` is smaller than $\mathtt{L}+\mathtt{S}+\mathtt{T}$ in many cases (marked by $<$). For some methods, `M` is even smaller than $\mathtt{L}+\mathtt{T}$ (marked by `#`) or even than `L` itself (`_222_mpegaudio` and four `richards` benchmarks). This is possible because LaTTe can coalesce copies between local variables generated by the *x*`load`-*x*`store` bytecode sequences, and can allocate the same register into nonoverlapping local variables through its conservative live variable analysis. This flexibility is due to LaTTe's aggressive register mapping with pseudocode generation as well as LaTTe's efficient register allocation, which obviates preallocating local variables as in CACAO or VTune.

In this table, we can also find there are only two methods that spill (marked by $*$). Since spills are related to the register pressure of the translated code as well as to the quality of register allocation, we also need to check those cases where register pressure would be higher.

We examined the top five methods with the largest number of local variables as shown in Table 3. Although the register pressure is much higher, we see spills only in five methods.[8] (In this table, `M` is still smaller than $\mathtt{L}+\mathtt{S}+\mathtt{T}$ and smaller than $\mathtt{L}+\mathtt{T}$ or `L` in even more methods).

We have also measured the same data for the optimized version of LaTTe for the same methods in Table 2 and Table 3 where the register pressure is higher due to inlining and other optimizations. In particular, `L` tends to be increased due to inlining. Also, there are many cases when `S` is reduced while `T` is increased. This is due to CSE which replaces many stack variables by temporary variables. We found that even with this higher register pressure, LaTTe rarely spills registers.

7. In fact, all phases in the LaTTe JIT compilation are linear in the bytecode size except for the register allocation phase. The backward sweep and the forward sweep are linear, but the reconciliation at join points is quadratic, however, we found in practice that the reconciliation time is negligible in most cases.

8. The first method in many benchmarks ($\mathtt{M}=35, \mathtt{L}=37, \mathtt{S}=7, \mathtt{T}=1$) that causes the spill is the same one in the JDK class library called `dtoa()` which converts double numbers into strings.

TABLE 3
Register Allocation Quality of the Base LaTTe for Top Five Largest-Locals Methods

Top Five Largest-Local	1st method				2nd method				3rd method				4th method				5th method			
	M	L	S	T	M	L	S	T	M	L	S	T	M	L	S	T	M	L	S	T
_200_check	*	35	37	7	1	22	14	8	0	<	16	13	5	0	<	13	13	13	0	<	16	13	3	1
_201_compress	*	35	37	7	1	22	14	8	0	<	16	13	5	0	<	13	13	13	0	<	16	13	3	1
_202_jess	*	35	37	7	1	*	21	16	10	1	<	18	16	5	0	#	14	15	5	0		22	14	8	0
_209_db	*	35	37	7	1	22	14	8	0	<	16	13	5	0	<	13	13	13	0	<	16	13	3	1
_213_javac	*	35	37	7	1	<	23	22	8	0	<	26	22	10	1	<	24	22	4	1	<	23	21	8	0
_222_mpegaudio		22	14	8	0	<	16	14	3	0	<	16	13	5	0	<	13	13	13	0	<	16	13	3	1
_227_mtrt	*	35	37	7	1	<	27	23	7	0	<	23	21	7	0	<	21	19	8	1	<	21	16	8	1
_228_jack	*	35	37	7	1		22	14	8	0	#	12	14	7	1	<	18	14	6	1	<	16	13	5	0
Series	*	35	37	7	1	<	17	15	8	0	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0
LUFact	*	35	37	7	1	<	18	15	10	0	<	17	14	12	0	<	16	13	5	0	<	18	13	9	0
SOR	*	35	37	7	1	<	22	18	8	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0
HeapSort	*	35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3
Crypt	*	35	37	7	1	<	18	15	4	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0
FFT	*	35	37	7	1	<	39	35	6	0	<	16	13	5	0	<	15	11	4	1	<	13	10	9	0
Sparse	*	35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	16	9	8	1
Search	*	35	37	7	1	<	20	20	5	0	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0
Euler	*	35	37	7	1	<	38	33	13	0	<	32	21	18	0	<	26	19	8	1	<	20	16	12	1
MD	*	55	52	6	0	*	35	37	7	1	*	31	30	4	1	<	19	19	3	0	<	16	13	5	0
MC	*	35	37	7	1	<	19	15	4	1	<	19	15	8	0	#	14	15	5	0	<	16	13	5	0
RayTracer	*	35	37	7	1	<	24	21	7	0	<	20	19	6	0	<	25	17	12	0	#	14	16	10	0
richards_g	*	35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3
richards_gf	*	35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3
richards_gns	*	35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3
richards_dna	*	35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3
richards_dav	*	35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3
richards_daf	*	35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3
richards_dai	*	35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3
richards_all	*	35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3
spell	<	16	13	5	0	<	12	11	2	1	<	13	10	9	0	<	12	10	5	0	<	12	9	3	3
javacc	#	14	21	7	0	*	22	20	6	1	#	21	19	3	3	<	18	16	4	1	<	19	16	3	1
delta	*	35	37	7	1	<	16	13	5	0	<	13	10	9	0	<	12	10	5	0	<	13	10	5	1
jjtre	#	8	20	3	1	#	15	18	3	1	#	6	15	3	1	#	12	14	5	0	<	16	13	5	0
jlex		18	14	4	0	<	16	13	5	0	<	14	12	3	1	<	14	11	5	0	<	13	10	9	0
hash	#	27	25	3	3	<	22	21	5	1	<	17	15	4	1	<	16	14	3	0	<	20	13	17	1

These results indicate that even with LaTTe's aggressive mapping of registers and copy coalescing, the register pressure of the translated code would rarely be too high, which makes LaTTe's fast, region-based register allocation with local lookahead effective enough to avoid spills.[9]

In conclusion, LaTTe generates efficient code via aggressive register mapping and efficient register allocation. On the other hand, it is unlikely for a JIT compiler that can generate code as efficient as LaTTe's to be much faster than LaTTe since LaTTe's JIT compilation overhead is already small enough. Therefore, we believe LaTTe made a reasonable trade-off between speed and quality of JIT compilation.

8 Summary

In this paper, we described the design and implementation of LaTTe, a Java JIT compiler with fast and efficient register mapping and allocation. Our aggressive register mapping with a separate pass for real register allocation that coalesces copies with a local lookahead is an elaborate engineering solution that trades off the quality of generated code and the speed of JIT compilation. This trade-off was confirmed empirically by measuring translation overhead and performance impact.

Acknowledgments

The source code of LaTTe can be downloaded from its Web site http://latte.snu.ac.kr, and more than 4,200 copies have been downloaded as of November 2005. This paper revises and expands—with results and details not originally presented—a paper published in the *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach, California, pages 196-204, October 1999. That work was supported by a grant from the IBM T.J. Watson Research Center. Kemal Ebcioğlu performed the work described in this paper while at the IBM T.J. Watson Research Center. He is currently at Global Supercomputing Corporation, New York.

9. Actually, even some of those spills are due to the SPARC calling conventions, not due to high register pressure.

References

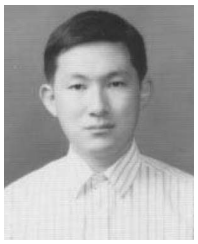
[1] J. Farley, *Java Distributed Computing.* O'Reilly, 1998.
[2] J. Aycock, "A Brief Bistory of Just-in-Time," *ACM Computing Surveys,* vol. 35, no. 2, June 2003.
[3] F. Yellin and T. Lindholm, *The Java Virtual Machine Specification.* Addison-Wesley, 1996.
[4] D.L. Weaver and T. Germond, *The SPARC Architecture Manual Version 9,* 1994.
[5] A. Tucker, "Coloring a Family of Circular Arcs," *SIAM J. Applied Math.,* vol. 29, no. 3, pp. 493-502, Nov. 1975.
[6] S. Muchnick, *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.
[7] P. Briggs, K. Cooper, T. Harvey, and L. Simpson, "Practical Improvement to the Construction and Destriuction of Static Single Assignment Form," *Software Practice and Experience,* vol. 28, no. 8, July 1998.
[8] Kaffe Home Page, http://www.transvirtual.com/, 1998.
[9] A.-R. Adl-Tabatabai, M. Ciernak, G.-Y. Lueh, V.M. Parikh, and J.M. Stichnoth, "Fast, Effective Code Generation in a Just-in-Time Java Compiler," *Proc. ACM SIGPLAN '98 Conf. Programming Language Design and Implementation,* http://orp.sourceforge.net, June 1998.
[10] A. Krall, "Efficient JavaVM Just-in-Time Compilation," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* http://www.cacaojvm.org, 1998.
[11] *The Java HotSpot Virtual Machine,* vol. 1.4.1, http://java.sun.com/products/hotspot/, Sept. 2002.
[12] M. Paleczny, C. Viek, and C. Click, "The Java HotSpot Server Compiler," *Proc. Java Virtual Machine Research and Technology Symp. (JVM '01),* pp. 1-12, Apr. 2001.
[13] B. Alpern et al., "The Jalapeno Virtual Machine," *IBM System J.,* vol. 39, no. 1, Feb. 2000.
[14] Jikes RVM homepage, http://jikesrvm.sourceforge.net, 2000.
[15] M. Poletto and V. Sarkar, "Linear Scan Register Allocation," *ACM Trans. Programming Languages and Systems,* vol. 21, no. 5, Sept. 1999.
[16] T. Suganuma et al., "A Dynamic Optimization Framework for a Java Just-in-Time Compiler," *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications,* Oct. 2001.
[17] C. Chambers and D. Ungar, "Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language," *Proc. SIGPLAN '89 Conf. Programming Language Design and Implementation,* 1989.
[18] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Kamatsu, and T. Nakatani, "Design, Implementation, and Evaluation of Optimizations in a Just-in-Time Compiler," *Proc. ACM 1999 Conf. Java Grande,* 1999.
[19] M. Cierniak, G.-Y. Lueh, and J.M. Stichnoth, "Practicing JUDO: Java under Dynamic Optimizations," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 13-26, June 2000.
[20] S. Lee, B.-S. Yang, and S.-M. Moon, "Efficient Java Exception Handling in Just-in-Time Compilation," *Software Practice and Experience,* vol. 34, no. 15, Dec. 2004.
[21] B.-S. Yang, S.-M. Moon, and K. Ebcioğlu, "Lightweight Monitors for the Java Virtual Machine," *Software Practice and Experience,* vol. 35, no. 3, Mar. 2005.
[22] H.-K. Choi, Y.C. Chung, and S.-M. Moon, "Java Memory Allocation with Lazy Worst Fits for Small Objects," *The Computer J.,* vol. 48, no.4, July 2005.
[23] Y.C. Chung, S.-M. Moon, K. Ebcioğlu, and D. Sahlin, "Reducing Sweep Time for a Nearly Empty Heap," *Proc. 27th Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '00),* pp. 378-389, Jan. 2000.
[24] SPEC JVM98 Benchmarks, http://www.spec.org/osg/jvm98, 1998.
[25] The Java Grande Forum Benchmark Suite, http://www.epcc.ed.ac.uk/javagrande/, 1998.

Byung-Sun Yang received the BEE and MSEE degrees in electrical engineering and computer sciences from the Seoul National University, Korea, in 1997 and 1999, respectively, and is currently working toward the PhD degree.

Junpyo Lee received the BEE and MSEE degrees in electrical engineering and computer sciences from the Seoul National University, Korea, in 1998 and 2000, respectively, and is currently working toward the PhD degree.

SeungIl Lee received the BEE and MSEE degrees in electrical engineering and computer sciences from the Seoul National University, Korea, in 1998 and 2000, respectively, and is currently working toward the PhD degree. He had participated in developing open source Java Virtual Machine, LaTTe, and is currently working on Java Virtual Machine on embedded systems. His research interests include performance optimization such as just-in-time compilation and ahead-of-time compilation for Java Virtual Machines especially on embedded systems.

Seongbae Park received the BEE and MSEE degrees in electrical engineering and computer sciences from the Seoul National University, Korea, in 1997 and 1999, respectively, and was a staff engineer of the SPARC code generator group at Sun Microsystems from 1999-2006. He is now a staff engineer at Google Inc. His research interests are in the areas of the microprocessor architecture and the optimizing compiler.



Yoo C. Chung received the BEE and MSEE degrees in electrical engineering and computer sciences from the Seoul National University, Korea, in 1999 and 2001, respectively, and is currently enrolled in the PhD program at the Information and Communications University.

Suhyun Kim received the PhD degree in electrical engineering and computer sciences from the Seoul National University, Korea, in 2005, and is currently a postdoctorate at the IBM T.J. Watson Research Center. His current research interests are in the field of optimizing compilers, with special interests in the following topics: instruction-level parallelism, embedded systems, dynamic optimization, and virtual machines.

Kemal Ebcioğlu received the PhD degree in computer science from the State University of New York at Buffalo in 1986. He conducted research on compilers, architectures, and languages for fine-grain parallelism at the IBM T.J. Watson Research Center, from 1986 to 2005. Dr. Ebcioglu led numerous IBM Research projects on fine-grain parallelism (including VLIW and DAISY). His last position at IBM was coleader, Programming Model and Tools, HPCS/PERCS Project, which is IBM's DARPA-funded supercomputer design effort. In 2006, he retired from IBM and founded Global Supercomputing Corporation. Dr. Ebcioglu has more than 70 technical publications and nine US patents. He has served as IFIP Working Group 10.3 (Concurrent Systems) chair in the period 2001-2006, and as ACM SIGMICRO Chair in the period 1999-2005. He has served as general chair, program chair, and steering committee chair for various conferences related to fine grain parallelism. He is an associate editor of *ACM Transactions on Architecture and Code Optimization*, and is a senior member of the IEEE and the IEEE Computer Society. His present research interests include high productivity peta-scale systems, overcoming the memory wall barrier, parallel utility computing, and dynamic binary translation and optimization.

Erik Altman is a research staff member at the IBM T.J. Watson Research Center. His research interests include binary translation and optimization, compilers, architecture, and microarchitecture. He has authored or coauthored more than 30 conference and journal papers. He was one of the original architects of the IBM DAISY project, that allowed VLIW architectures to achieve 100 percent binary compatibility with the PowerPC architectures, while also providing excellent performance. He was also one of the original architects of the Cell processor chip that is to appear in the forthcoming Sony Playstation 3 game consoles. He has been the program chair and general of several conferences, such as the International Conference on Parallel Architectures and Compilation Techniques (PACT) and the P=ac2 (Power/Performance = Architecture x Circuits x Compilers). He has served on numerous program committees, and has also served as guest editor of *IEEE Computer*, the *ACM Journal of Instruction Level Parallelism*, and the *IBM Journal of Research and Development*. He is currently the vice-chair of the ACM Special Interest Group on Microarchitecture (SIGMICRO). He is a member of the IEEE and the IEEE Computer Society.

Soo-Mook Moon received the PhD degree at the University of Maryland, College Park, in 1993. From 1992-1993, he worked at the IBM T.J. Watson Research Center where he developed the IBM VLIW compiler. From 1993-1994, he was a software design engineer at the Hewlett-Packard Company in the California Language Lab where he contributed to the development of an optimizing compiler for the PA-RISC CPUs. Since 1994, he has been with the faculty of the Seoul National University in the School of Electrical Engineering where he is now a professor. He now leads the Microprocessor Architecture and System Software (MASS) Laboratory, which is researching advanced compilation techniques for ILP machines, embedded RISC CPUs, and Java virtual machines in the context of just-in-time (JIT) compilation. Professor Moon visited the IBM T.J. Watson research center as a visiting scientist during the summer of 1997. He was with Sun Microsystems during 2002 as a visiting professor. He was a receipient of IBM Faculty Award in 2000-2001. He is a member of the IEEE and the IEEE Computer Society.