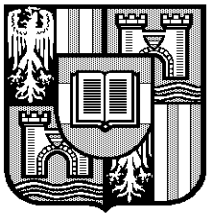

JOHANNES
KEPLER
UNIVERSITÄT
LINZ

Institut für Praktische Informatik
(Systemsoftware)

Adding Static Single Assignment Form and a Graph Coloring Register Allocator to the Java Hotspot™ Client Compiler

Hanspeter Mössenböck

Report 15
November 2000

Abstract

This report describes the work that I performed during my sabbatical in the Java Hotspot™ group at Sun Microsystems in Cupertino between June and August 2000. The goal was to find out how much effort it was to augment the Java Hotspot™ Client Compiler with Static Single Assignment (SSA) Form and with a Graph Coloring Register Allocator. SSA form not only simplifies register allocation but also helps performing optimizations such as common subexpression elimination, loop-invariant code motion or instruction scheduling.

I implemented a prototype compiler that creates a control flow graph (CFG), transforms the intermediate representation (IR) of the program into SSA form and does register allocation with graph coloring. The code generator is still missing. On the basis of this work, the Hotspot team should be able to decide whether or not it is worth adding SSA form and register allocation to the client compiler.

Throughout this report the term current compiler refers to the Hotspot™ Client Compiler [GrMi00] at the time when my work began. The term new compiler refers to our prototype compiler.

Contents

1. The Control Flow Graph (CFG)

The control flow graph is an intermediate representation of a program capturing its instructions and the control flow between them. Its nodes are basic blocks and its edges are jumps between basic blocks.

There was already a control flow graph in the current compiler, where a block was represented by a list of instructions starting with a *BlockBegin* instruction and ending with a *BlockEnd* instruction (e.g. a *Goto*) leading to the *BlockBegins* of other blocks.

In the new compiler, I added an explicit *Block* type, which makes it easier to traverse the CFG and also separates the concepts of blocks and instructions more clearly. Every block can have 0..*n* successors and 0..*m* predecessors. It has a pointer to the sequential list of instructions belonging to this block. It also has a unique block number (*block_id*) and the bytecode index (*bci*) of its first instruction. It finally has a *dom* field pointing to the immediate dominator of the block. A few more fields that are used for SSA form generation and register allocation will be introduced later. Fig. 1-1 shows the structure of blocks:

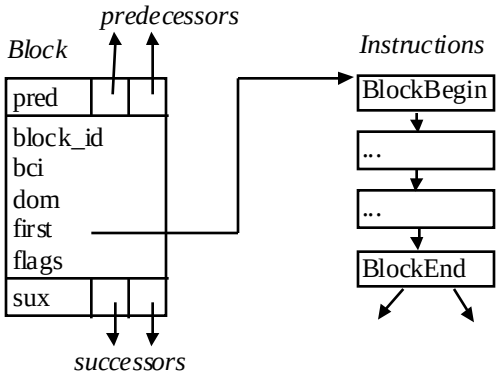

Fig. 1-1. Blocks in the CFG

In contrast to the current compiler, the new compiler stores not only the successors but also the predecessors of blocks. This is necessary because we sometimes have to generate register moves and load instructions in predecessor blocks. The predecessors are also necessary for constructing the dominator tree.

Currently, there is some redundant information in the data structures of the old and the new compiler. Both a *Block* and a *BlockBegin* have a *block_id*, a bytecode index (*bci*) and partly the same flags. A *BlockEnd* points to the (*BlockBegins*) of the successor blocks, whereas the *sux* fields of a *Block* point to the successor *Blocks*. This redundancy should eventually be removed. We left it in the prototype of the new compiler, because too many other parts of the compiler would have had to be changed otherwise.

1.1 Building the Control Flow Graph

The CFG is built from the Java bytecodes. As in the current compiler, the new compiler first finds the block leaders (i.e. the beginnings of blocks) by considering the targets and the successor instructions of jumps. However, it also remembers from where a jump came (the jump bci in the predecessor block). The CFG construction is implemented in *BlockListMaker::set_leaders* and in *GraphMaker::buildCFG* and proceeds as follows:

- Find block leaders, create blocks and remember predecessor bci's for every block.
- Sort blocks according to bci, so that the block containing a certain bci can be found by binary search.
- From the predecessor bci's compute the predecessor and successor edges of the CFG.
- Find loop headers.
- Remove dead blocks.
- Fill blocks with instructions and build the dominator tree.

All blocks are stored in an array *blocks*, which can be used for traversing blocks sequentially.

1.2 Finding Loop Headers

Various traversals of the CFG require that loop header blocks are treated in a special way. Thus they have to be found and marked. For finding the loop headers, the CFG is traversed using the following algorithm (in pseudocode):

```
visit(b):
  if (!b.visited) {
    b.visited = true; b.active = true;
    for (all successors s) visit(s);
    b.active = false;
  } else if (b.active) b.bwd_branches++;  // mark b as a loop header;
```

This algorithm is implemented in *GraphMaker::mark*. It counts the number of incoming backward branches. We know that a block *b* is a loop header if *b.bwd_branches* > 0. At the same time we also know the number of incoming forward branches, which is the difference between the number of predecessors and the number of incoming backward branches. The number of incoming forward branches is needed later (e.g. for the construction of the dominator tree).

1.3 Building the dominator tree

The dominator tree is needed for inserting blocks into empty branches (see Section 2.8). It is also used for optimizations. The dominator tree is built such that there is a *dom* pointer from every block to its parent block. This allows us to store an *n*-ary tree with just a single pointer per block.

In order to save graph traversals the generation of the dominator tree is combined with the process of filling blocks with instructions (*GraphMaker::fill_block*). The traversal algorithm is based on a DFS, where the sons of a block *b* are visited only when all forward branches to *b* have been traversed. Here is the algorithm in pseudocode:

```
visit(b, pred):
  b.ref--;
  if (b.dom == null) b.dom = pred; else b.dom = common_dom(b.dom, pred);
  if (b.ref == b.bwd_branches)
    for (all successors s) visit (s, b);
```

pred is the predecessor of *b*. *b.ref* is initialized with the number of predecessors in every block. It is decreased with every visit, and when it reaches the number of incoming backward branches, we know that all forward branches to this block have already been traversed, so it is safe to proceed to the sons of this block.

During the first visit of a block *b*, its *dom* pointer is set to its predecessor. At all further visits the *dom* pointer is recomputed as the nearest common dominator of *b.dom* and *pred*. This can easily be computed by traversing the *dom* chain starting at *b.dom* and marking every visited block. Then we traverse the *dom* chain starting at *pred*. The first marked block that we encounter on this way is the nearest common dominator (see *GraphMaker::common_dom*).

1.4 Eliminating Dead Blocks

The current version of javac may generate dead code. For example, when a jump leads to another jump, this indirection is transformed into a direct jump, possibly leaving the other jump as dead code. Dead blocks are eliminated by marking all used blocks (during loop header analysis) and then deleting all unmarked blocks.

2. Static Single Assignment Form (SSA Form)

SSA form is an intermediate representation of IR instructions, which simplifies many optimizations as well as register allocation. Its basic idea is that for every variable in a program there should be just one point where this variable is assigned a value. If there are multiple assignments to a variable in the original program, the program is transformed so that every assignment creates a new variable, and this value is used in all subsequent instructions up to the next assignment. As a consequence of this transformation, if two variables have the same name, we can be sure that they also contain the same value, because they have been assigned at the same place.

An algorithm for creating SSA form was given in *Cytr91*. We used a different algorithm based on the abstract interpretation of the bytecodes. This algorithm seems to be simpler although it may generate redundant phi functions which have to be eliminated later. It seems also better applicable to the situation that we are compiling from bytecodes and not from source code. A basic form of this algorithm was already implemented in the current compiler, but it was restricted to single blocks, while we apply it to the whole CFG.

2.1 State Arrays

In Java every local variable has an index which is its address on the stack. To keep track of the most recent assignment to every variable, we use a *state array*, where the *i*-th element points to the instruction where the variable with index *i* got its most recent value. Actually this is not a store instruction (since stores are eliminated in the IR) but the instruction that produced the value on the right-hand side of the assignment.

While the IR instructions are generated, a state array *locals* is carried along. If an instruction creates a value for variable *i*, a pointer to this instruction is stored in *locals*[*i*]. If an instruction uses a variable *j* as an operand, this operand is replaced by *locals*[*j*], i.e. by a pointer to the instruction where this value was created. Fig.2-1 shows a piece of source code, its bytecodes, the generated IR, and the contents of the state array after every IR instruction.

```
source code            bytecodes    generated IR                  locals

void f(int n,int m)    iload_0      i10: L0      // load n        (i10, ---, ---, ---)
{ int i, j;            iconst_1     i11: 1       // load 1        (i10, ---, ---, ---)
  i = n + 1;           iadd         i12: i10+i11 // i = n + 1     (i10, ---, i12, ---)
  j = m - n;           istore_2     i13: L1      // load m        (i10, i13, i12, ---)
}                      iload_1      i14: i13-i10 // j = m - n     (i10, i13, i12, i14)
                       iload_0
                       isub
                       istore_3
```

Fig. 2-1. IR for a simple statement sequence and the corresponding state array *locals*

At the end of the block, the state array contains pointers to the instructions where the most recent values of all variables were set (Fig.2-2).

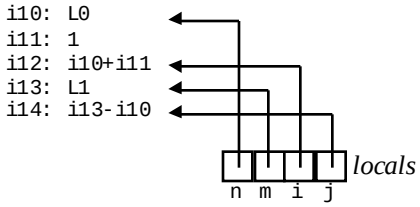

Fig.2-2. A state array points to the instructions where the current values of variables were set

A state array also keeps track of values that remain on the Java expression stack at the end of a block (e.g. in conditional expressions). Thus the state array consists actually of two arrays: *locals* and *stack*, but what we describe for the *locals* array in the following sections, applies also to the *stack* array. Both the *locals* array and the *stack* array are implemented in the class *ValueStack*, so state arrays are *ValueStack* objects.

2.2 Merging State Arrays -- Phi Functions

When a block *b* has a single predecessor, the state array of this predecessor becomes the initial state array of *b*, and is propagated through the instructions of *b*. If *b* has two ore more predecessors, however, there are two or more state arrays flowing into the block, which have to be merged before a single state array can continue to flow through *b*.

If two distinct values *i1* and *i2* of a variable *i* flow together at a block, they have to be merged into a so-called ϕ-function (phi function) at the beginning of this block as shown in Fig. 2-3. The left-hand side of this phi function is the value of *i* which flows from there.

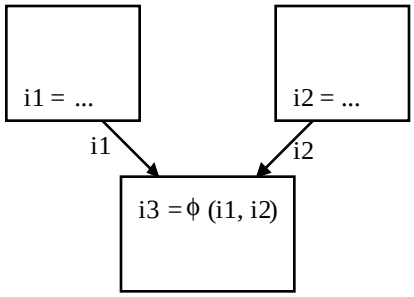

Fig. 2-3. Two values are merged into a phi function

A phi function has as many operands as there are incoming branches into the block to which it belongs. The meaning of a phi function *i3* = ϕ(*i1*, *i2*) (also written as *i3* = [*i1*, *i2*]) is: If we came over the first branch, the current value of *i* is *i1*; if we came over the second branch, it is *i2*. Phi functions are eliminated before code generation (see Section 3.3), since there is no machine instruction for them.

What is shown for a single value in Fig. 2-3 can be applied to a whole state array element by element. If two state arrays *a1* and *a2* flow together, a new state array *a3* is created, and a phi function is generated in *a3*[*i*] if *a1*[*i*] and *a2*[*i*] differ, otherwise *a3*[*i*] = *a1*[*i*] = *a2*[*i*]. Fig.2-4 shows all possible cases in which elements of a state array can be merged giving a new state array.

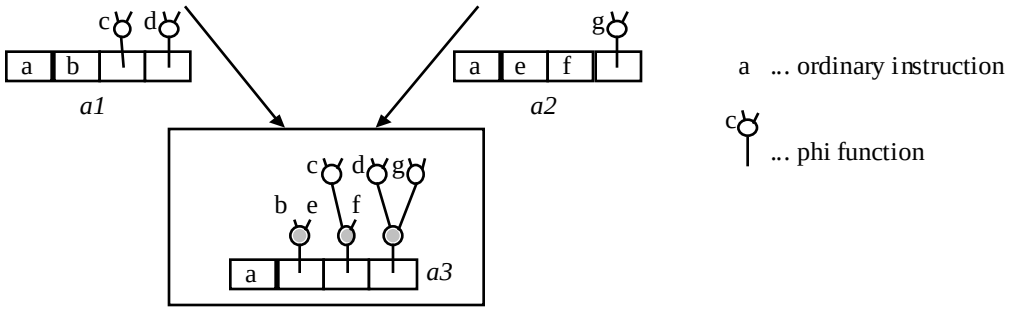

Fig.2-4. Merging state arrays that flow into a block

Fig. 2-5 shows an example of an if statement and its corresponding IR. There are four blocks B0 to B3. B2 is the join block of the if statement and holds a state array with three elements for the variables *n*, *i*, and *j*, respectively. Since different values of *i* and *j* flow together at B2, the state array contains phi functions for *i* and *j*, each having two operands, because B2 has two predecessors. The phi function i20, for example says that if we came from B1 the value of *i* is i14 (i.e. 1), if we came from B0, the value of *i* is i18 (i.e. 2). There is no phi function for *n* since the same value of *n* flows into B2 from both predecessors. Note that when *i* and *j* are used in instruction i22, the phi function values i20 and i21 are used for *i* and *j*, respectively.

```
source code          IR

int foo(int n) {     B3 i11  L0
  int i, j;             i12  0
  if (n > 0) {               if i11 <= i12 then B0 else B1
    i = 1; j = 2;
  } else {           B1 i14  1
    j = 1; i = 2;       i15  2
  }                          goto B2
 return i - j;
}                    B0 i17  1
                        i18  2
                             goto B2

                     B2 ------------------
                        0: i11              // n
                        1: i20 = [i14, i18] // i
                        2: i21 = [i15, i17] // j
                        ------------------
                        i22    i20 - i21
                               ireturn i22
```

Fig. 2-5. IR for an if statement

2.3 Phi Functions in Loop Headers

Loop headers are blocks that have at least 1 incoming backward branch and at least 2 predecessors so they need phi functions for variables that have different values in incoming branches. A special problem in loop headers is that the values flowing in from a backward branch are only known when the loop body has already been processed. If a phi function for some variable *i* has to be generated because of a value flowing in from a backward branch, all uses of *i* in the loop have to be renamed to point to this new phi function.

In order to avoid renaming of variables, we generate phi functions in loop headers for all variables in advance. If it turns out that a phi function was not needed, we eliminate it later. This seemed to be simpler than renaming variables and avoids having to maintain def-use lists.

Fig.2-6 shows a loop and its phi functions, assuming that there are only two local variables *x* and *y*. At the beginning of the loop header the phi functions *x2* and *y2* have been inserted. When *x* is used in the loop, it is the value *x2* that is used there. *x* is assigned a new value in the loop which flows back as the value *x3* to the phi function *x2*. As one can see, *y* does not get a new value in the loop, so the value flowing back into the phi function *y2* is *y2* again. This is an indication that the phi function for *y2* is redundant and can be eliminated.

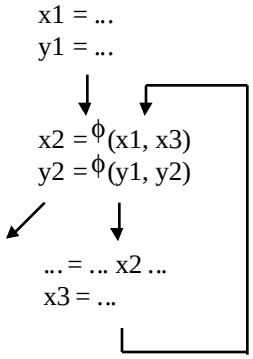

Fig. 2-6. Phi functions in a loop header

Fig.2-7 shows a more complete example of a while statement and its translation to IR. There are phi functions for all three variables (*n*, *i*, *j*) generated in the loop header prophylactically. Since *n* is never set in the loop, its phi function is redundant and can be replaced by the value that flew into the loop. But because

n was also not set before the loop, we have to generate a load instruction (L0) in the dominator of the loop header. Generating load instructions is described in more detail in Section 2.8. The question mark in the phi function i14 denotes a still undefined value (a *Local* value, as described in Section 2.6). Since the phi function i14 is redundant it is replaced by the inserted load instruction for *n* (alias i22).

```
source code              IR

int foo(int n) {         B3 i11  0                       // j = 0
  int i, j;                 i22  L0                      // load n
  j = 0;                         goto B0
  for (i=0; i<n; i++)
    j = j + i;           B1 i18  i16 + i15               // j = j + i
  return j;                  i19  1
}                            i20  i15 + i19              // i = i + 1
                                  goto B0

                         B0 ------------------
                            0: i14 = [?, i14] alias i22  // n
                            1: i15 = [i11, i20]          // i
                            2: i16 = [i11, i18]          // j
                            -----------------
                                  if i15 < i22 then B1 else B2

                         B2       ireturn i16            // return j
```

Fig. 2-7. IR for a while statement

2.4 Elimination of Redundant Phi Functions

It may turn out that a phi function was unnecessarily generated in which case it can be eliminated. There are two cases in which a phi function becomes redundant. Phi functions of the form

x = [y, x, x, ..., x]

(in which all but one operands equal the left-hand side) can be replaced by *y*. Such phi functions can appear in loop headers if a variable is not modified in the loop. The other form of redundant phi functions is

x = [x, x, ..., x]

(in which all operands are equal and are the same as the left-hand side). These phi functions can be replaced by *x*. They can appear in the join nodes of if and switch statements, if one or more operands were different from *x* initially but have been replaced by *x* due to eliminated phi functions further up in the CFG.

As soon as all branches into a node have been traversed, the phi functions of this node are *committed*, i.e. checked for the above two patterns. If one of them occurs, the phi function is virtually deleted by setting its field *subst* to the instruction that should substitute it. The compiler already uses this substitution mechanism for other purposes so that a single pass over all instructions will finally replace all references to virtually deleted instruction by their substitution. An example of a deleted phi function is i14 in Fig.2-7. This phi function is substituted by i22.

2.5 Implementation of Merging

For phi functions we introduced the new instruction kind *PhiFun*. The merging of state arrays is implemented in *GraphMaker::merge* which is called from *GraphMaker::fill_block*. It is based on the following ideas:

- If a block has only one predecessor, no merge is necessary and no state array is stored for this block.
- If a block with at least 2 predecessors is visited for the first time the incoming state array is stored in this block. If it is visited again, the new state array is compared with the stored state array element by element, and phi functions are generated or updated as described in Fig.2-4.
- The successors of a block are only visited after all forward branches to this block have been traversed. This makes sure that all necessary phi functions have already been generated before proceeding to the successors.

Care has been taken to generate as few state arrays as possible. If a block *b* has *n* successors, only *n*-1 state arrays are generated. The *n*-th successor receives the state array that was built up during the traversal of *b*.

When all incoming branches of a block have been traversed the phi functions of this block are committed (*GraphMaker::commit_phi*) and redundant phi functions are eliminated (*GraphMaker::simplify0*). At the time when a phi function *x* is committed, however, it may have operands which are still uncommitted phi functions. If those get eliminated later, the phi function *x* may also become deletable. Since this is not decidable here, all phi functions are rechecked after the whole CFG has been built (*GraphMaker::delete_redundant_phis* and *GraphMaker::simplify*). Nevertheless, it seems to be a good idea to try to eliminate redundant phi functions as soon as possible, because this will avoid unnecessary creation of other redundant phi functions.

2.6 HiWord and Local Instructions

Most variables in Java occupy a single word, except variables of type *long* and *double*, which take 2 words. If a variable with index *i* takes 2 words, it also takes 2 slots in the state array: slot *i* pointing to the instruction where this value was set, and slot *i*+1 containing a so-called *HiWord* instruction, which is mainly used for making sure that such slots are not inadvertently used for other purposes. If the variable *i* has not yet been assigned to, slot *i* of the state array holds a pointer to a so-called *Local* instruction, denoting an uninitialized local variable (or parameter).

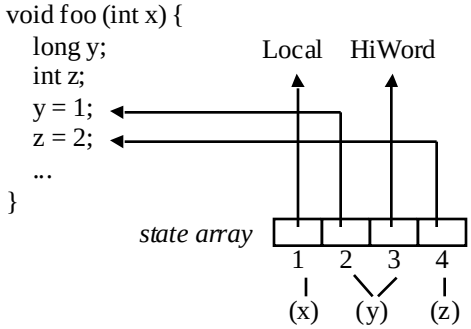

Fig. 2-8. Typical contents of a state array

Fig. 2-8 shows the contents of the state array after the assignment to *z*. The parameter *x* has not been assigned yet, therefore its array slot 1 points to a *Local* instruction. Slot 2 points to the instruction where *y* got its value. Since *y* is of type long, it occupies two slots, and slot 3 points to a *HiWord*. Finally, slot 4 points to the instruction where *z* got its value.

2.7 Type Analysis

When the compilation of a method is started, we only know the number of words needed for its local variables (*method.max_locals*), but we do not know the types of the individual variables. Yet this knowledge is necessary, in order to initialize the state array appropriately, i.e. to reserve 2 slots for *long* and *double* variables, and to set the right types in the *Local* instructions, with which all slots except *HiWords* are initialized.

As a solution to this problem, we do type analysis during the parsing of the bytecodes, i.e. when we split the bytecode stream into basic blocks. Since every variable is eventually loaded or stored, and since load and store instructions are typed in Java, we can derive the variable types from the load and store instructions.

Another unpleasant problem is, that there may be variables with the same index but with different types in Java. Since variables may be declared in inner blocks, and since parallel blocks may declare variables of different types, the program

```
void foo() {
  int x;
  if (...) {
    long y;
    ...
  } else {
    int z;
    ...
  }
}
```

has both *y* and *z* at index 1, but *z* has type *int* and needs just one slot, while *y* has type *long* and needs 2 slots. We solve this problem by assigning distinct indexes to overlapping variables. In the above method, *x* stays at index 0, *y* gets index 1 (occupying 2 slots) and *z* gets index 3. Thus, there is only one variable at every index in the state array. Of course this can cause the state array to grow.

For reassigning indexes to variables we use a matrix *loc* in addition to the state array. *loc* has one line for every possible variable type (*int*, *long*, *float*, *double*, *object*). If a variable is of type *t* and has originally index *i*, *loc*[*t*, *i*] contains the new index of this variable. Fig.2-9 shows the contents of the state array and of *loc* for the above method *foo*.

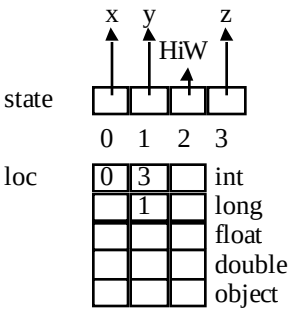

Fig. 2-9. Data structures for reassigning variable indexes

The variable *y* is of type *long* and has originally index 1, so its new index is indicated by *loc*[*long*, 1], which is again 1. The variable *z* is of type *int* and has originally also index 1. Its new index is *loc*[*int*, 1], which is 3.

The mapping is implemented in class *LocalMap*, which has essentially 2 methods, *put* and *get*. *put*(*t*, *i*) is invoked for every load or store bytecode for a variable with type *t* and index *i* (*BlockListmaker:: set_leaders*). If this was the first load or store instruction for index *i* (*state*[*i*] == null), *put* creates a new *Local* instruction of type *t*, stores it in *state*[*i*], and lets *loc*[*t*, *i*] point to *i*. If there was already a load or store instruction for index *i* (*state*[*i*] != null), *put* obtains a new index *j* at the end of the state array, creates a *Local* instruction of type *t* for *state*[*j*], and lets *loc*[*t*, *i*] point to *j*.

The method *get* is called when the IR instructions are generated from the bytecodes. Every time a load or store instruction for a variable of type *t* and index *i* is encountered (*GraphMaker::load_local* and *GraphMaker::store_local*), *get*(*t*, *i*) provides the new index by returning *loc*[*t*, *i*].

All slots in the state array that were not assigned *Local* or *Hiword* instructions are finally assigned a *Local* instruction of type *unknownType*. Such slots may occur, when a variable or parameter was declared but not used. Most Java compilers will report this as an error, but the program should also be compilable if they do not.

After type analysis, a state array has been built, which contains a *Local* instruction with the appropriate type at the appropriate index for every variable of the method. This array is used as the initial state array in the construction of SSA form, and is passed to *GraphMaker::build_CFG*.

2.8 Generating Loads for Parameters

Every local variable has to be assigned before it is used, so the using instructions refer to the instruction where the most recent value of this variable was set. For parameters, however, things are different. Parameter values are passed by the caller, so the using instructions cannot reference a value-assigning instruction in the same method. Therefore we insert *LoadLocal* instructions for parameters immediately before they are used for the first time in a method (*GraphMaker::load_local*). The using instructions can then reference this *LoadLocal* instruction. Whether a parameter is still unassigned can be recognized by the fact if its element in the state array points to a *Local* instruction (see Section 2.6).

During code generation, a *LoadLocal* instruction has to be translated into a load of the respective parameter into the register denoted by the *LoadLocal*. If parameters are already passed in registers, register allocation should try to map the *LoadLocal* instruction to the register in which the parameter is passed; if this is possible, code generation can ignore the *LoadLocal*, otherwise a register move has to occur.

If the *i*-th operand of a phi function in a block *b* points to a *Local* instruction, we have to generate a *LoadLocal* at the end of the *i*-th predecessor of *b*. This can be seen in Fig.2-10, where the second phi function in block B0 would have had a *Local* as its first operand. Therefore a *LoadLocal* was generated in B5, and the phi function i19 references it.

```
source code                IR

int foo(int n, int m) {    B2 i12  L0
  if (n < 0) {                i13  0
    n = 0; m = 0;              14  if i12 >= i13 then B5 else B1
  }
  return n + m;            B5 i25  L1              // generated LoadLocal
}                              24  goto B0

                           B1 i15  0
                               17  goto B0

                           B0 ------------------
                              0: i18 = [i12, i15]  // n
                              1: i19 = [i25, i15]  // m (originally i19 = [?, i15])
                              ------------------
                              i20  i19 + i18
                              i22  ireturn i20
```

Fig. 2-10. Insertion of a *LoadLocal* instruction in B5

Note that B5 (the else branch of the if statement) had to be generated just to place the *LoadLocal* instruction in it. In general, if the phi function is in block *b* and the *LoadLocal* instruction should go into a predecessor block *pred*, a block has to be inserted between *b* and *pred* if

- *pred* = *b.dom* (the immediate dominator of *b*) and
- there is some other predecessor *pred'* ≠ *pred* of *b* that is not dominated by *b* (i.e. the edge between *pred'* and *b* is not a backward branch). See Fig.2-11.

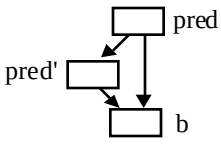

Fig.2-11. pred = *b.dom* and *b* does not dominate *pred'*

3. Register Allocation

Register allocation tries to assign registers to all variables and intermediate values in a method. Constraints are the number of available registers and the fact that two values that are live at the same time cannot reside in the same register. We also have to take into account that certain instructions require their operands to be in special registers.

We start by assuming that we have an unlimited number of *virtual registers.* Every instruction that produces a value is assigned a new virtual register that holds this value (a virtual register corresponds to the instruction number, e.g. register 10 for i10 in Fig.2-1). Next, we determine the *live ranges* of all values, i.e. the IR part between the point where a value was created and the point where it was used for the last time. Then we build the *Register Interference Graph* (RIG), whose nodes are the live ranges, and there is an edge between two nodes if the corresponding live ranges overlap. Finally, the RIG is colored such that adjacent nodes get different colors. The node color corresponds to the (physical) register number. If we have n colors (registers) available but the graph can only be colored with $m > n$ colors, then the colors 0 .. n-1 correspond to physical registers, while the colors n .. m-1 correspond to memory locations. Our register allocator does not generate spill instructions but assumes that every value is either always in a register or always in memory (from where it has to be loaded when needed). This simplifies the register allocator and avoids repetitive passes made necessary by spill code generation.

3.1 Live Range Analysis and the Register Interference Graph

3.2.1 The Basic Algorithm

Live ranges are computed by traversing the instructions in reverse order. For every instruction the following situation occurs

```
          ---------- live'  = live - {x} + {y, z}
x = y op z
          ---------- live
```

where *y* and *z* are operands (there can be any number of them), *x* is the value produced by this instruction and *op* is some operation. We start with a set *live* of values that are live at the end of the instruction and propagate it backwards. Obviously, *x* is not live before the instruction, because it was just set here, so we exclude it from *live.* However, *y* and *z* must have been live before the instruction, otherwise we could not have used them there. So the set of live values at the beginning of the instruction is *live* - {*x*} + {*y, z*}. This data flow equation is repeatedly applied to all instructions of a basic block from the last one to the first one.

Note that SSA form considerably simplifies the analysis. If we had not transformed the program to SSA form, *x* might also be alive before the instruction, because it was possibly set by a previous assignment. In SSA form, every assignment creates a new variable so we can be sure that *x* is not alive before its assignment.

While we compute the live ranges we can also build the RIG. The live ranges of two values overlap if one value is alive where the other one is set. Thus we only have to look at the places where values are created. The steps for processing an instruction *x* = *y* op *z* are thus:

(*live* is the set of variables alive after the instruction)
1. *live* = *live* - {*x*}
2. generate edges between *x* and all values in *live*
3. *live* = *live* + {*y, z*}

(*live* is the set of variables alive before the instruction)

Fig.3-1 shows an example of how the live ranges and the RIG are computed for the instruction sequence of Fig.2-1, assuming that no values are alive at the end of the sequence. The example has to be read from bottom to top.

```
                    live = {}
i10  L0                             edges:
                    live = {i10}
i11  1                              edges: i11-i10
                    live = {i10, i11}
i12  i10 + i11                      edges: i12-i10
                    live = {i10}
i13  L1                             edges: i13-i10
                    live = {i10, i13}
i14  i13 - i10                      edges:
                    live = {}
```

Fig. 3-1. Computing the live ranges and the RIG for a statement sequence

The RIG resulting from Fig.3-1 is shown in Fig.3-2. It is easy to see that the graph can be colored with two colors, 0 and 1.

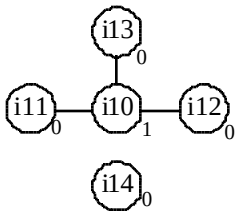

Fig.3-2. The RIG resulting from Fig.3-1

The instruction sequence can thus be written as

```
r1 = L0
r0 = 1
r0 = r1 + r0
r0 = L1
r0 = r0 - r1
```

3.2.2 Live Range Analysis in the Presence of Branches

When a CFG contains branches, the live ranges are computed for every block individually starting with the last block and proceeding to the first block in a postorder traversal. The live set at the end of a branch node is the union of the live sets at the beginning of its successors, as is shown in Fig.3-3. The live sets within the basic blocks as well as the edges of the RIG are computed in the same way as shown in Fig.3-1.

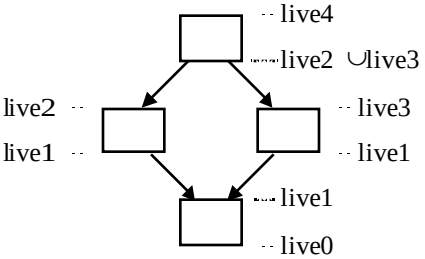

Fig.3-3. Live set propagation in the presence of branches

In order to avoid the recomputation of live sets, every block stores the set of values that is live at its beginning.

3.2.3 Live Range Analysis in the Presence of Loops

The live sets in a loop cannot be computed in a single pass. Looking at Fig.3-4, the values that are live at the last block of a loop (*live0*) are the values that are live at the beginning of its successor, i.e. the loop header (*live2*). However, *live2* has not yet been computed (it is still empty) when it is needed for the first time, so *live0* starts with the empty set and a preliminary version of *live2* is computed in a first pass. Then a second pass of live range analysis is performed, where *live2* is not empty any more but contains all values that were used and not killed in the loop. After this second pass, all live sets in the loop are complete.

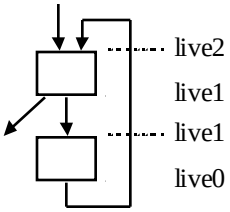

Fig.3-4. Live set propagation in the presence of loops

Unfortunately, the number of passes necessary to compute the live sets depends on the nesting level of loops. A simple loop needs 2 passes. If a loop contains another loop, we need 3 passes. In general, if the maximum nesting level of loops in a method is *n*, we need *n*+2 passes to compute the live sets.

There is a way to restrict the number of passes to 2 irrespective of the loop nesting level. In the first pass all values that are used and not killed in loops are propagated to the headers of all outer loops. Before we start pass 2, we can add the live values at a loop header to the live values at all inner loop headers. This is justified by the observation, that if a value is life in a loop, it is also live in all inner loops. Unfortunately, this technique cannot be easily applied in the Hotspot compiler, because we have to take into account that the control flow in a method can be completely unstructured (e.g. when the bytecodes were hand-crafted) and thus the nesting structure of loops would be difficult to find. Although it could probably be done, it is so complicated that we refrained from doing it in this version of the compiler. Rather, we iteratively perform live range analysis until the live sets in the blocks do not change any more. This is safe and can perhaps be optimized later.

3.2 Special Issues in Live Range Analysis

3.2.1 Handling Phi Functions

A phi function reflects the fact that the control flow came to a block via different branches. When we perform live range analysis, we have to treat these branches separately. If we have the following sequence of phi functions

```
x3 = [x1, x2]
y3 = [y1, y2]
z3 = [z1, z2]
```

and if *live* is the set of values alive at the end of this sequence, then the set of values alive at the beginning of the sequence is either

```
live - {x3, y3, z3} + {x1, y1, z1}
```

or

```
live - {x3, y3, z3} + {x2, y2, z2}
```

depending on whether we plan to continue to the first or to the second predecessor. The live sets stored in the blocks are the values that are alive after the phi functions and before the other instruction of this block.

When the live set of block *b* is requested by its *i*-th predecessor, the set *b.live* is propagated through the phi functions backwards, removing their left-hand sides and adding their *i*-th operands. At the same time, the edges of the RIG are computed.

3.2.2 Handling Constant Operands

Many machine instructions are able to take a constant as an immediate operand. It would be a waste of registers to load constants into registers if they can also be processed as immediate operands. Therefore, if an IR instruction

```
x = y op z
```

can be translated into a machine instruction with *z* as an immediate operand, and if *z* is a constant, the data flow equation is modified to

```
live' = live - {x} + {y}
```

Also, the instruction in which the constant *z* is defined should not produce edges in the RIG. Of course, constant operands are handled in the same way wherever the occur. For example, an array store instruction

```
storeIndexed  array, index, value
```

can have both *index* and *value* as constants, excluding them from live range analysis if the actual values are constants.

3.2.3 Putting Values into Specific Registers

Some machines have instructions that require operands to be in specific registers. For example, on the Intel i486 the first operand of a division must be in eax; the second operand of a shift instruction must be in ecx, etc. There are also instructions that produce values in specific registers. For example, the i486 places the remainder of a division into edx. But even on more regular machines there are cases where a value has to be in a specific register. For example, the parameters of a method have to go into the first *k* registers. We therefore have to find a way to tell the register allocator to assign specific registers to certain values.

Values are produced by instructions. Therefore every instruction has a field *reg* that indicates the register in which the value of this instruction should be stored. If *reg* is initialized with -1, the allocator will assign it an arbitrary free register. If *reg* is initialized with a value ≥ 0, however, the allocator will not modify it, but will regard it as the register in which the value of this instruction should be stored. We have to distinguish two cases:

- *Instructions that require an operand in a specific register*. If an instruction

  ```
  x = y op z
  ```

  requires *y* to be in a specific register *r*, it is not a good idea to simply set *y.reg* to *r*, because this could cause conflicts with other instructions that need *r* as well. Therefore, we introduce a register move immediately before the instruction so that it becomes

  $y_r = y$
  $x = y_r$ op z

  and set y_r*.reg* to *r*. If we do that with all operands that have to be in specific registers, none of them will block the register for its whole life. Later we will try to join the registers of *y* and y_r (see Section 3.3) so that in many cases the register move can be eliminated again. For register moves we introduce a new instruction kind *RegMove*.

- *Instructions that leave their results in specific registers*. If an instruction

  x = y op z

  should leave its result in a specific register *r*, we set *x.reg* to *r* but introduce a register move immediately after the instruction so that it becomes

  x_r = y op z
  x = x_r

  This register move makes the specific register free again immediately after it was set. Later we try to join the registers of *x* and x_r, and in may cases the register move can be eliminated again.

Handling Register Moves

Register moves have to be treated in a special way during live range analysis. For example, if we consider the following Java program

```
i = x << d;
j = x >> d;
```

and its IR

```
i1  Lx
i2  Ld
i3₁ := i2          // register move because shift distance must be in register 1
i4  i1 << i3₁
i5₁ := i2          // register move because shift distance must be in register 1
i6  i1 >> i5₁
```

live range analysis would look as follows (to be read from bottom to top):

```
i1  Lx
                live = {i1}
i2  Ld                              edges: i2-i1
                live = {i1, i2}
i3₁ := i2                           edges: i3-i1, i3-i2
                live = {i1, i2, i3}
i4  i1 << i3₁                       edges: i4-i1, i4-i2
                live = {i1, i2}
i5₁ := i2                           edges: i5-i1
                live = {i1, i5}
i6  i1 >> i5₁
                live = {}
```

producing the RIG shown in Fig.3-5.

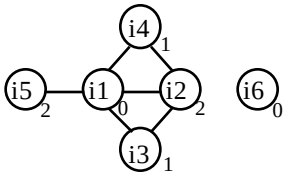

Fig.3-5. RIG with an interference between i2 and i3

As we can see, there is an interference between i2 and i3, which prevents us from putting these two values into the same register. Therefore the register move i3 := i2 cannot be eliminated.

The reason for this problem is that i2 is still alive after the register move i3 := i2. However, we should take our initial intention into account. We introduced the move i3 := i2 just for the purpose that if some *other* value needed to go into register 1 and if this *other* value interfered with i2, i2 should be stored in a register different from 1. We did not want to introduce an interference between i3 and i2. In other words, we should not make the left-hand side of a register move interfere with its right-hand side. For a register move $x := y$ we should rather take the following steps:

(*live* is the set of variables alive after the instruction)
1. $live = live - \{x, y\}$
2. generate edges between x and all values in *live*
3. $live = live + \{y\}$

(*live* is the set of variables alive before the instruction)

This avoids the edge between i2 and i3 and allows us to put i2, i3 and i5 into register 1, making both register moves unnecessary.

3.2.4 Handling Instructions Needing Several Registers

Some IR instructions such as *NewInstance* or *CheckCast* are translated to multiple machine instructions and need several temporary registers. These registers should be reserved by the register allocator. Since every register corresponds to an instruction that provides the value of this register, we have to introduce a new pseudo instruction *Affect*, the only purpose of which is to indicate to the allocator, that it should reserve a specific register ($reg \geq 0$) or arbitrary register ($reg = -1$) at this point.

In the current version of our compiler, we place *Affect* instructions immediately after the instruction x that has a need for temporary registers. It would be cleaner, however, to have the *Affect* instructions as "arguments" of x. This would make it easier to move x during instruction scheduling. However, since the number of temporary registers needed by high-level instructions differs on various machines, the IR would become machine-dependent. One possible solution for this problem is to allow a variable length list of arguments for all instructions, and to have the back end attach the *Affect* instructions to the other instructions.

The *Affect* instructions have to be treated in a special way during live range analysis, too. If an instruction has a sequence of *Affect* instructions, the registers assigned to these *Affects* should be different from each other. Therefore we have to introduce interference edges between all instructions of an *Affect* sequence.

Another example of using the *Affect* instruction is when we indicate to the register allocator that an instruction such as division on the i486 not only places the quotient into eax but also affects edx by placing the remainder there. It furthermore requires the dividend to be in [edx, eax]. The division $x = y / z$ is therefore translated into the following IR:

```
i1  Ly
i2  Lz
i3₀ := i1     // register move: dividend has to be in register 0 (eax)
i4₂ affect    // division also affects register 2 (edx)
i5₀ i3 / i2   // division leaves result in register 0 (eax)
i6  := i5₀    // register move makes eax free again
```

Appendix B shows a table of registers required by the various bytecode instructions.

3.2.5 Handling Values of Type long

Values of type *long* need 2 words, i.e. 2 registers. Instructions producing a *long* result therefore have to be colored with 2 registers. However, if an instruction corresponds to a node in the RIG, there would be nodes that had 2 colors, which would complicate the coloring algorithm. Therefore we preferred to stay with the simple rule that every instruction corresponds to just a single register. For instructions producing *long* values we add a *HiWord* instruction accounting for the second register.

An instruction *x* producing a *long* value and its corresponding *HiWord* instruction *y* are linked to each other by

```
x.next == y         (i.e., y immediately follows x)
y.lo_word == x
```

Requiring that *y* immediately follows *x* makes it difficult to move *x* during instruction scheduling. One could solve that problem by using a different pointer instead of *next*, which we did not do in our current version of the compiler.

There are two kinds of instructions which are not in the regular instruction stream and therefore have their *next* fields unused: the *PhiFun* and the *Local* instructions. Both of them are stored in state arrays though and have already a corresponding *HiWord* as an element of the state array (see Section 2.6). We just have to link these instructions with their *HiWords* in the above mentioned way in order to tie them together.

Fig.3-6 shows a method using *long* values and its translation into IR. One can see that instructions l9, l13, l17, l20 and l23 are followed by *HiWord* instructions. At the end of the method there is a register move of the result into the required registers (r0, r2). These register moves can possibly be eliminated later by joining the left-hand and right-hand sides (see Section 3.3).

```
source code                  IR

long foo(long n, long m) {   B0 l9   L0         // load n
  return (n + m) / 2;           l10  hi(l9)
}                               l13  L2         // load m
                                l14  hi(l13)
                                l17  l9 + l13   // n + m
                                l18  hi(l17)
                                l20  2L         // load 2
                                l21  hi(l20)
                                l23  l17 / l20  // (n + m) / 2
                                l24  hi(l23)
                                r0   := l23     // register move to r0,r2
                                r2   := l24
                                l28  lreturn r0
```

Fig.3-6. Operations with *long* values and their translation to IR

Long values have to be treated in a special way during live range analysis. If we consider the long value operation

a = b op c
d = hi(a)

we have to do the following steps:

(*live* is the set of variables alive after the *HiWord* instruction)
1. *live* = *live* - {*d*}
2. generate edges between *d* and all values in *live*
3. if *b* is a long value then *live* = *live* + {*b.next*}
   if *c* is a long value then *live* = *live* + {*c.next*}
4. live = live - {a}
5. generate edges between *a* and all values in *live*
6. *live* = *live* + {*b, c*}

(*live* is the set of variables alive before the instruction)

Phi functions are not in the regular instruction stream, so if they produce *long* values they cannot be handled by the above sequence of steps. Rather, a phi function *a* = [..., *b*, ...] has to be treated as follows:

(*live* is the set of variables alive after the phi function)

1. if *a* is a long value then
   live = *live* - {*a.next*}
   generate edges between *a.next* and all values in *live*
2. if *b* is a long value then *live* = *live* + {*b.next*}
3. *live* = *live* - {*a*}
4. generate edges between *a* and all values in *live*
5. *live* = *live* + {*b*}

(*live* is the set of variables alive before the instruction)

3.2.6 Handling Values of Type float and double

Values of type *float* or *double* have to be stored in floating point registers. Since these are different from integer registers, we would have to build a separate RIG for floating point values and color it independently.

The current version of our compiler does not do that. Instead, we assume that for floating point values a simple on demand register allocation will be done by the code generator as it is done in the current compiler.

3.3 Joining Values in the Register Interference Graph

There are places in a program, where it is desirable to put two or more values into the same register. For example:

- The left-hand side and the right-hand side of register moves should go into the same register so that the move can be eliminated.
- All operands of a phi function as well as its left-hand side should go into the same register so that the phi function can be eliminated.
- On two-address machines such as the Intel i486, an operation *x = y op z* should have *x* and *y* in the same register.

Putting two values into the same register can be achieved by joining their nodes in the RIG into a single node (this is also called *coalescing*). This will reduce the size of the RIG and will assign a single color to the joined node, i.e. to the two values for which the node stands. Two nodes *x* and *y* can only be joined if

- There is no edge between them in the RIG.
- If *x.reg* ≥ 0 and *y.reg* ≥ 0 then *x.reg* and *y.reg* must be the same.
- After joining the nodes, adjacent nodes must still have different colors (i.e. if their *reg* values are ≥ 0 they must be different).

If a value *b* is joined with a value *a*, the field *b.subst* is set to *a* (*a* becomes a substitute for *b*), *b* and all its edges are removed from the RIG, and the former edges of *b* are added to the edges of *a*. After coloring the RIG, all values *i* with *i.subst* != *null* get *i.reg* set to *i.subst.reg* (actually the substitution chain can also be longer in which case the last node in the chain is the one that was colored). In Fig.3-7 *b* and *c* were joined with *a*, and *d* was joined with *c*. Only *a* remains in the graph and gets colored. *b*, *c* and *d* get their color from *a*.

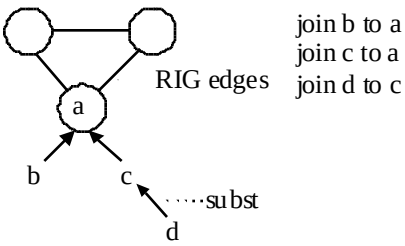

Fig. 3-7. Joining four values where *a* is the value that is going to be colored

3.3.1 Joining Register Moves

If $x := y$ is a register move and x and y can be joined, the code generator can eliminate the move. The only thing that we have to take care of is that joins of register moves should happen after other joins (i.e. after joins in phi functions and two-address instructions). The reason is that one join operation can make other joins impossible, because every join operation makes the RIG denser, i.e. the average number of edges per node increases and so does the probability that two nodes that are to be joined are connected by an edge. Joining the operands of register moves is less important than joining other values so we want to give them lower priority.

During live range analysis we collect pairs of values that are to be joined. After the RIG has been built we traverse the list and try to join every pair. The list is built such that join pairs of two-address instructions come before join pairs of register moves. This gives them priority over register moves.

3.3.2 Joining Source and Destination in Two-Address Instructions

If $x = y$ op z is an instruction on a two-address machine, x and y have to be in the same register. Therefore we add the pair (x, y) to the list of join pairs. If it turns out that x and y cannot be joined we try to swap y and z if *op* is commutative and x and z can be joined. If that does not work either, we try to put x and z into *different* registers by introducing an egde between them in the RIG. The code generator can then produce the following code:

```
x = y
x = x op z
```

If it is not possible to put x and z into different registers, the code generator will have to use a scratch register s and produce the following code:

```
s = y
s = s op z
x = s
```

3.3.3 Joining Phi Operands

If $x = [y, z]$ is a phi function and x can be joined with both y and z, the code generator can eliminate the phi function, since the incoming values y and z are in the same register as the outgoing value x.

Phi operands are not entered into the join list, however. Instead we iterate over the state arrays of all blocks to find the phi functions and join their operands immediately.

If a phi operand for branch i cannot be put into the same register as the left-hand side of the phi function, a register move has to be generated at the end of the i-th predecessor block. This will move the value that could not be joined into the correct register, and the phi function can be eliminated again. Fig.3-8 illustrates this process.

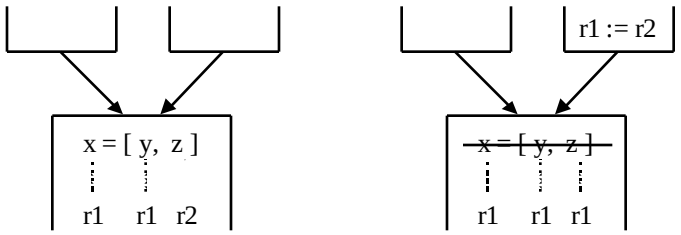

Fig.3-8. A register move is generated for every phi operand that is not joinable

In Fig.3-9 we have the situation that a register move has to be generated for the phi operand *z* but the corresponding predecessor block is missing. In this case we have to insert an empty block and place the register move there. This situation is the same as the one described in Section 2.8 in which *LoadLocal* instruction had to be generated for phi operands of kind *Local*.

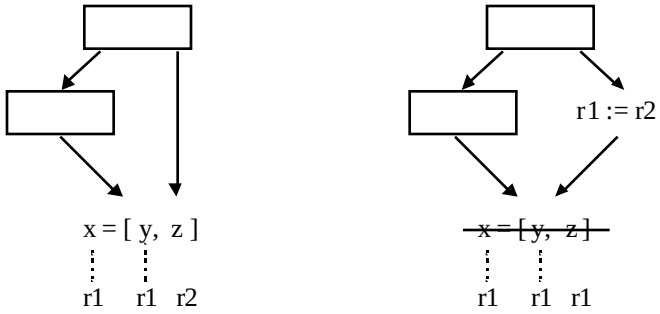

Fig.3-9. A block has to be inserted in order to place the register move

3.4 Graph Coloring

3.4.1 The Basic Algorithm

For coloring the RIG we use the standard coloring algorithm [e.g. *Much97*] with some adaptations for handling values that have to go into specific registers. The general idea is as follows: if we have *n* colors available, we search for a node with less than *n* neighbors. If we find such a node, we remove it together with its edges and color the remaining graph recursively. Then we add the node and its edges again and assign it a color that is different from the colors of its neighbors. Since there are less than *n* neighbors there will always be a color available for this node.

If we cannot find a node with less than *n* neighbors, we choose the node which has minimal "weight" (see below), remove it, and color the remaining graph recursively. When we add this node again, all of the *n* colors may already be used by its neighbors. In this case we assign it the smallest color greater than *n*. If a node has a color greater than *n* this means that it resides in memory instead of in a register. In general, if we have *n* registers but need $m > n$ colors to color the RIG, then the colors $0..n-1$ correspond to real registers, while the colors $n..m-1$ correspond to memory locations. A stack frame would then consist of *m* slots (plus the slots for floating point values), where the first *n* slots are used as a save area for registers and the remaining slots are the variables that cannot be kept in registers.

Our simple coloring algorithm does not create spill instructions, i.e. instructions that move values from registers to memory and back in order to decrease the register pressure. Instead, we divide the set of values into those which *always* reside in registers and those which *always* reside in memory. This may not be optimal but much simpler and faster than a spilling register allocator. It allows us to do the coloring in a single pass. On RISC machines, where many registers are available, it should produce good results in most

of the cases. The register moves introduced for values that have to go into special purpose registers (see Section 3.2) provide some way of live range splitting and reduce the register pressure.

On a CISC machine we keep one register as a scratch register that we do not use for coloring. Most CISC instructions allow at least one operand to be in memory. The other operand can be loaded into the scratch register if necessary. On a RISC machine we have to reserve 2 scratch registers for this purpose.

Fig.3-10 shows an example of a RIG that is to be colored with 3 colors. First, nodes are removed and after the remaining graph has been colored they are added again, and a color is assigned to them.

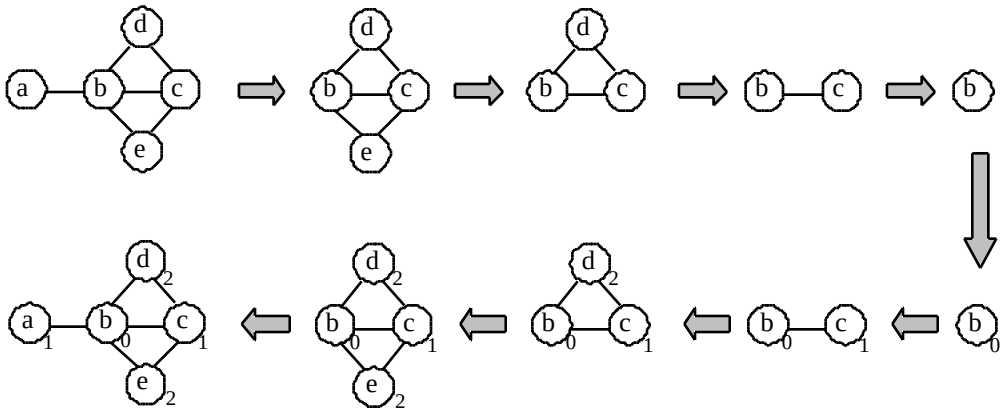

Fig. 3-10. Coloring a graph with 3 colors

3.4.2 Putting values into specific registers

Before the graph is colored, its nodes are either uncolored (*reg* = -1) or precolored (*reg* ≥ 0). We can be sure that no two precolored nodes have equal color if they are adjacent. Precolored nodes should retain their color, thus we can simply exclude them from coloring and just have to make sure that the color of their neighbors will not conflict with them.

We do that be removing only uncolored nodes and coloring the remaining graph recursively. When the graph only contains precolored nodes we stop the recursion and begin adding nodes again. Fig.3-11 shows the coloring of the same graph as in Fig.3-10, but this time nodes *a* and *e* should go into register 1, i.e. they are precolored with 1.

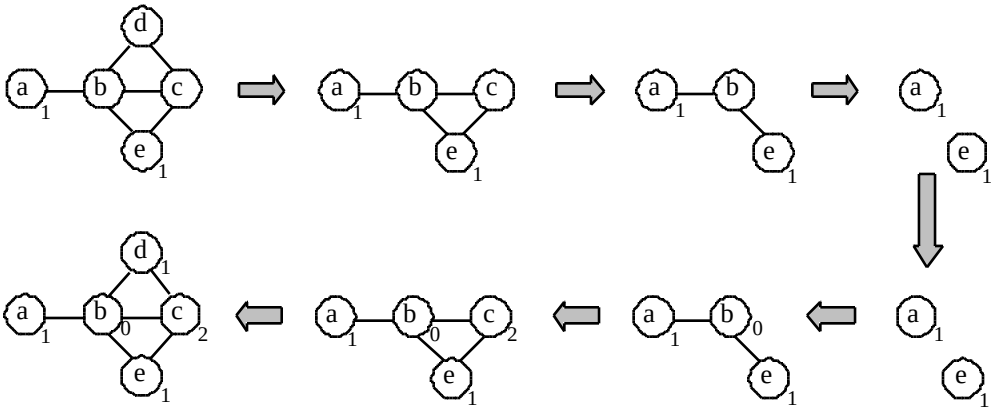

Fig.3-11. Coloring a graph where *a* and *e* have to get color 1

Here is a pseudocode version of the coloring algorithm which also handles values that have to go into specific registers. *n* is the number of available registers.

```
color():
  x = any uncolored node with less than n neighbors;
  if (there is such a node) {
    remove x and its neighbors from RIG;
    color();
    add x and its neighbors to RIG;
    x.reg = smallest register which is not used by neighbors of x;
  }
```

3.4.3 Computing the Weights of Nodes

If the coloring algorithm cannot find a node with less than *n* neighbors, it should remove the node with minimal weight. The weight should be a function of the costs that it takes to access this value if it resides in memory instead of in a register. A common cost function is a weighted use count of this value, where every use outside of a loop increases the count by 1, while every use within a loop is multiplied by a factor of 10.

If we remove the nodes with minimal access costs first, these nodes are more likely to receive register numbers greater than *n* and thus go into memory, whereas the higher cost values receive lower numbers and are therefore allocated in registers. This makes sense because we want the values that are accessed more frequently to be in registers.

Because of time restrictions, we did not implement the weight computation so far. When we cannot find a node with less than *n* neighbors we currently take any of the yet uncolored nodes. The only difficult issue in the weight computation will be how to determine the nesting level of a use, especially in the case of unstructured programs.

3.5 Implementation of Register Allocation

Our register allocator is implemented in four classes:

RIG represents the Register Interference Graph with operations to add and remove nodes, to set, remove and check for edges between nodes, to join nodes, and finally to color the graph. In addition to the actual graph, which is stored as a lower triangular bit matrix, we also maintain an array *val*, which maps node numbers (i.e. instruction numbers) to instructions. Fig.3-12 sketches the data structures of the *RIG* for a given graph.

RIGNode represents the nodes of the *RIG*. This is mainly a data class.

RIGVisitor traverses the IR instructions in reverse order, performs live range analysis, and builds the RIG. It holds the set of currently live values as well as the list of value pairs that are to be joined later. When it traverses phi functions the *RIGVisitor* knows in which branch it is, i.e. which phi operand it should consider.

RAlloc is the proper register allocator. Its main method is *build_RIG* which is called from *IRScope::build_graph.* The method *life_at_block*(*b*) computes the values that are live at the beginning of block *b*. This method calls itself recursively for the successors of *b*, starts the *RIGVisitor* on the instructions of the blocks and stores the live sets in the blocks. *RAlloc* also has methods to join nodes of the graph and to generate register moves for phi operands that cannot be joined with the left-hand side of their phi functions.

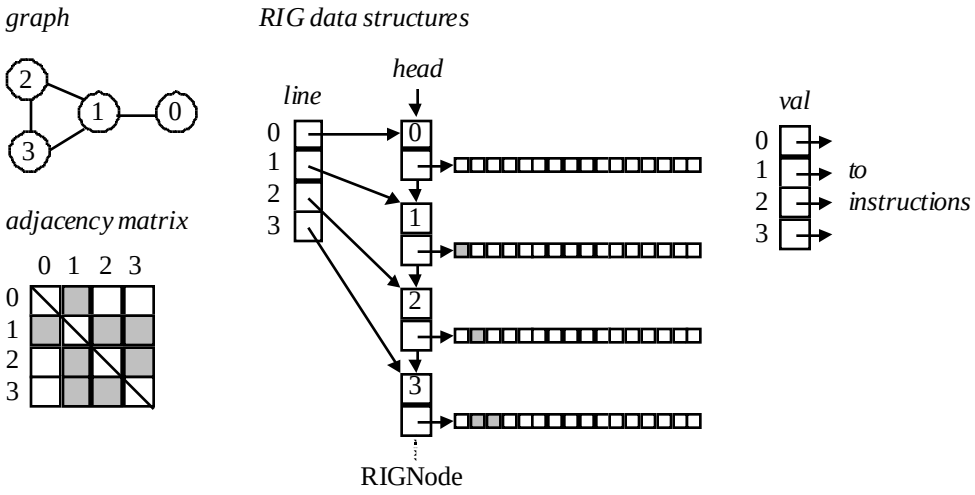

Fig.3-12. Data structures of the RIG for a given graph

As can be seen from Fig.3-12 a RIG consists of several lines. The *line* array provides a mapping from node numbers to *RIGNode*s. A *RIGNode* points to the set of this node's neighbors. The *RIGNode*s are also linked into a linear list which makes it easy to remove and add nodes and still find out quickly which nodes are currently in the graph without having to traverse the whole line array. Note that only the lower triangle of the adjacency matrix is stored in the neighbor sets. The sets are implemented as objects of class *Bitset*, which offers bit sets of arbitrary length stored as word arrays. Since all sets are of equal length (the number of instructions in the method), the length is not stored in the sets but is implemented as a static variable of class *Bitset*.

3.6 Interfacing the Code Generator to the Register Allocator

The code generator of the current compiler uses a visitor to traverse the instructions, allocate registers, create items and emit code. If our register allocator is used, the code generator can work in the same way, except that register allocation does not have to be done for most values. If the code generator visits an instruction *x*, the following situations may occur:

- If *x* is a *Constant* and *x.reg* < 0, this constant will be used as an immediate operand in a subsequent instruction. No register has to be allocated.
- If *x* is of type *int* or *object*, *x.reg* holds the register that is to be used for the value of this instruction.
- If *x* is of type *long*, *x.reg* and *x.next.reg* hold the register pair that is to be used for the value of this instruction. The instruction *x.next* is a *HiWord* and should be ignored.
- If *x* is of type *float* or *double*, the code generator should allocate a floating point register as in the current version of the compiler.
- If *x* is a *RegMove*, then if *x.reg = x.value.reg* the instruction should be ignored, otherwise a register move should be emitted. It may be necessary to update the items that hold *x.value.reg* so that they now hold *x.reg*.
- If *x* is an *Affect* instruction, *x.reg* holds the register that has been reserved as a temporary. *Affect* instructions have to be ignored otherwise.

Before code for an instruction is emitted, the code generator should check if the registers that are used for the instruction are real registers or memory locations. If they are memory locations, it should generate load or store instructions to load the value into the scratch register or store it from there.

At a method call point the registers in use must be saved and after the call they must be restored. Currently there is no information about which registers are in use at a method call point. This could, however, be easily provided by our register allocator. During live range analysis we know which values are

alive at a method call point. After the graph has been colored we also know which registers are used by these values. We could simply store these sets for all method call points.

4. Conclusions

Building on the implementation of the current Java Hotspot™ Client Compiler we have added static single assignment form for IR instructions as well as a graph coloring register allocator. We have used simple algorithms that do not always produce optimal results but are fast. Although detailed measurements have not yet been performed since the code generation phase is still missing, we received the impression that compilation is not slowed down substantially by our modifications. Due to time restrictions in this sabbatical a number of issues are still missing or could be improved with hindsight.

Issues that are still missing

- SSA form is currently not generated for methods having exception handlers or jsr instructions. In these situations we simply "bail out", i.e. the method is not compiled but continues to be interpreted.
- Currently we do not compute the weights of values, that should be used to determine which values go into registers and which go to memory. The weights should be computed based on weighted use counts, which should not be very difficult.
- Currently we do not generate information about the locations of pointers at safepoints (i.e. the points where exceptions and garbage collection can occur safely). This could be easily added, however, since we know the live values at these points as well as their locations and just have to bring this knowledge into the desired form.
- Currently we do not produce information about which registers are in use at method call points. However, this information could be easily provided based on the knowledge about live values and their registers at method call points.

Issues that could be improved

- The idea to generate phi functions in loop headers for all variables in advance should be reconsidered. The advantage of this idea is that we do not have to rename the use of values after a phi function was inserted. The disadvantage is that we have to eliminate redundant phi functions later, which turned out to be more complicated than expected. One should try out to create phi functions in loop headers only when necessary and to rename all uses of the corresponding values with def-use lists. The simpler of the two techniques should then be retained.
- Our compiler inserts *RegMove*, *Affect*, and *HiWord* instructions into the IR. In most cases this is made necessary by the architecture of the target machine (e.g. in order to force certain values into specific registers or to reserve temporary registers). Currently these modifications are made in class *GraphMaker*, which belongs to the front end of the compiler. Since they are machine specific, however, they should be made in the back end. In order to do that we would need a pass over all instructions before live range analysis. In this pass we could insert the necessary instructions.
- The *RegMove*, *Affect* and *HiWord* instructions are currently position dependent, i.e. they have to occur before or after a certain instruction in the IR. This may be a problem when instructions are to be moved (e.g. for instruction scheduling). As a remedy, one could make these instructions explicit arguments of the instructions to which they belong. One could add a variable length argument list to all instructions so that any number of auxiliary instructions can be attached to them. On the other hand, if register allocation is the last pass before code generation and if auxiliary instructions are only inserted before register allocation, their position dependent nature may not be a big problem.
- The existing practice of "pinning" instructions (i.e. marking those instructions that have to be generated in a fixed order) might not be correct yet. We did not have time to look at this aspect in detail.
- The memory used for live sets is currently larger than necessary. The set size depends on the maximum set element, i.e. on the largest instruction number. About 40% of the instructions do not generate a value, however, and thus will never occur as an element of a live set. One should use a denser

numbering scheme for instructions, so that only instructions that produce values get a number and so the live sets become smaller.

Acknowledgements

I would like to thank Robert Griesemer and Srdjan Mitrovic, the architects of the Java Hotspot™ Client Compiler. It was a pleasure to work with them and to study their compiler. Its clean and simple structure made it easy to add my enhancements. I am also grateful for the time that they took to discuss various aspects of my work and contributing many ideas. I would also like to thank all the colleagues in the Hotspot team and their managers, particularly Tricia Jordan, who made it possible for me to spend my sabbatical at Sun Microsystems, and who provided a comfortable working and living environment for me.

References

Cytr91 Ron Cytron, Jeane Ferrante, Barry K. Rosen, Mark N. Wegman: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM TOPLAS 13(4):451-490, October 1991.

GrMi00 Robert Griesemer, Srdjan Mitrovic: A Compiler for the Java HotSpot Virtual Machine. In László Böszörmenyi, Jürg Gutknecht, Gustav Pomberger (eds.): The School of Niklaus Wirth—The Art of Simplicity. dpunkt.verlag 2000

Much97 Steven S. Muchnick: Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997

Appendix A: Files and Classes

The following list shows the files and classes that were added or modified during this work (*name.** means *name.hpp* and *name.cpp*).

New

Files	***Classes***	***Description***
c1_Bitset.*	Bitset	Light weight bit sets for live sets and for marking.
c1_Block.*	Block	The basic blocks of the CFG.
c1_GraphMaker.*	LocalMap	Used during type analysis to make overlapping variable indexes unique.
	BlockListMaker	Corresponds to BlockListBuilder. Finds block leaders.
	GraphMaker	Corresponds to GraphBuilder. Builds the CFG and transforms the instructions into SSA form.
c1_RAlloc.*	RIGNode	A node of the RIG.
	RIG	The Register Interference Graph.
	RIGVisitor	Visits all instructions, performs live range analysis and finds the edges in the RIG.
	RAlloc	Starts the RIGVisitor on all blocks, tries to join nodes and starts the coloring of the RIG.

Modified

Files	***Classes***	***Description***
c1_Canonicalizer.*	Canonicalizer	Visitor functionality for Affect, Local, PhiFun, RegMove.
c1_CodeGenerator_i486.cpp	ValueGen	Visitor functionality for Affect, Local, PhiFun, RegMove.
c1_Inliner.cpp	Inliner	Visitor functionality for Affect, Local, PhiFun, RegMove.
c1_Instruction.hpp	Instruction	Affect, Local, PhiFun and RegMove added. HiWord is now also considered in visitors. Instruction got a reg field and a vid() method (preparation for a denser numbering scheme for instructions).
c1_InstructionPrinter.*	InstructionPrinter	Visitor functionality for Affect, Local, PhiFun, RegMove.
c1_IR.cpp	IRScope	Method build_graph now directs the building of the CFG, the gneration of SSA form and register allocation.
c1_ValueStack.*	ValueStack	Modifications to load_local and store_local. New methods store_local_grow, finish_locals, full_copy, dump.
c1_ValueType.*	UnknownType	added

Appendix B: Instructions Requiring Special Registers

The following bytecode instructions are currently implemented to need special registers on the i486:

Bytecode	*Register*	*Before operation*	*After operation*
imul	eax edx	multiplicand	low word of product high word or product
idiv irem	eax edx	low word of dividend high word or dividend	quotient remainder
ishl ishr iushr	ecx	shift distance	
ireturn areturn	eax	return value	
lreturn	eax, edx	return value	
invokevirtual invokeinterface	ecx eax (edx)	receiver auxiliary	 result if a function result if a long function
invokespecial	ecx	receiver	
new	eax ebx, ecx, edx, esi	 auxiliaries	allocated object
newarray	eax ebx, ecx, edx, esi, edi	 auxiliaries	allocated object
anewarray	eax ebx, ecx, edx, esi, edi	 auxiliaries	allocated object
athrow	eax	exception argument	
checkcast	ebx, ecx, edx	auxiliaries	
instanceof	ebx, ecx, edx	auxiliaries	
monitorenter	eax esi, edi	lock auxiliaries	
monitorexit	eax esi, edi	lock auxiliaries	

Appendix C: Examples

Swapping Variables

```
static int diff(int n, int m) {
  int h;
  if (m > n) {h = n; n = m; m = h;}
  return n - m;
}
```

IR before register allocation

Note how the swap assignments (*h* = *n*; *n* = *m*; *m* = *h*;) are encoded in the phi functions of B0 (i14 := i10 and i15 := i11). Block B1 does not have any instructions left.

```
B2(0) --> B1 B0   (dom: B4)
  0    0    i10    L1                              load m
  1    0    i11    L0                              load n
. 2    0     12    if i10 <= i11 then B0 else B1   if m <= n then B0 else B1

B1(5) --> B0   (dom: B2)
. 10   0     13    goto B0

B0(11) -->   (dom: B2)
  ------------------
  0: i14 = [i11, i10]  // n
  1: i15 = [i10, i11]  // m
  2: i16 = [?, i11]    // h (the ? denotes an unloaded Local)
  ------------------
  13   0    i17    i14 - i15                       n - m
  14   0    r0     := i17
. 14   0    i19    ireturn r0
```

IR after register allocation

Note how the phi functions are translated back into register moves in order to get rid of them. Block B5 was inserted in order to be able to place a register move there.

```
B2(0) --> B1 B5   (dom: B4)
  0    0    r2     L1                              edx := m
  1    0    r1     L0                              ecx := n
. 2    0     12    if r2 <= r1 then B5 else B1     if edx <= ecx then B5 else B1

B1(5) --> B0   (dom: B2)
  5    0    r0     := r2                           eax := edx
  5    0    r2     := r1                           edx := ecx
. 10   0     13    goto B0

B5(11) --> B0   (dom: B2)
  11   0    r0     := r1                           eax := ecx
. 11   0     21    goto B0

B0(11) -->   (dom: B2)
  13   0    r0     r0 - r2                         eax := eax - edx
  14   0    r0     := r0
. 14   0    i19    ireturn r0
```

Greatest common divisor

```
static int ggt(int n, int m) {
  int r;

  r = n % m;
  while (r != 0) {
    n = m;
    m = r;
    r = n % m;
  }
  return m;
}
```

IR before register allocation

Note how the assignments *n* = *m*; and *m* = *r*; are encoded in the phi functions of B0 (i18 := i19; and i19 := i20;).

```
B3(0) --> B0   (dom: B5)
  0    0    i11    L0                                load n
  1    0    i12    L1                                load m
  2    0    r0     := i11
  2    0    r2     affect
. 2    0    r2     r0 % i12                          r := n % m
  2    0    i16    := r2
. 4    0     17    goto B0

B1(7) --> B0   (dom: B0)
  13   0    r0     := i19
  13   0    r2     affect
. 13   0    r2     r0 % i20                          r := m % r  (actually n % m)
  13   0    i26    := r2
. 14   0     27    goto B0

B0(15) --> B1 B2   (dom: B3) - loop header
  ------------------
  0: i18 = [i11, i19]   // n
  1: i19 = [i12, i20]   // m
  2: i20 = [i16, i26]   // r
  ------------------
  16   0    i21    0
. 16   0     22    if i20 != i21 then B1 else B2   if r != 0 then B1 else B2

B2(19) -->   (dom: B0)
  20   0    r0     := i19
. 20   0    i29    ireturn r0                        return n
```

IR after register allocation

```
B3(0) --> B0   (dom: B5)
  0    0    r0     L0                            eax := n
  1    0    r3     L1                            ebx := m
  2    0    r0     := r0
  2    0    r2     affect
. 2    0    r2     r0 % r3                       edx := eax % ebx
  2    0    r1     := r2                         ecx := edx
  2    0    r0     := r3                         eax := ebx
. 4    0     17    goto B0

B1(7) --> B0   (dom: B0)
  13   0    r0     := r0
  13   0    r2     affect
. 13   0    r2     r0 % r1                       edx := eax % ecx
  13   0    r2     := r2
  13   0    r0     := r1                         eax := ecx
  13   0    r1     := r2                         ecx := edx
. 14   0     27    goto B0

B0(15) --> B1 B2   (dom: B3) - loop header
  16   0    i21    0
. 16   0     22    if r1 != i21 then B1 else B2  if ecx != 0 then B1 else B2

B2(19) -->   (dom: B0)
  20   0    r0     := r0
. 20   0    i29    ireturn r0       return eax
```

Sieve of Erathostenes

```
  static void sieve(int max) {
    boolean prime[] = new boolean[max];
    int p, i;
    for (i = 0; i < max; i++) prime[i] = true;
    p = 2;
    while (p < max) {
      System.out.println(p);
      for (i = p; i < max; i += p) prime[i] = false;
      do { p++; } while (p < max && !prime[p]);
    }
  }
```

IR before register allocation

```
B10(0) --> B0   (dom: B12)
  0    0    i19    L0                              i19 = max
. 1    0    r0     new boolean array [i19]         prime[] = new boolean[max]
  1    0    r3     affect
  1    0    r1     affect
  1    0    r2     affect
  1    0    r6     affect
  1    0    r7     affect
  1    0    a26    := r0                           a26 = prime
  4    0    i27    0                               i = 0
. 6    0     28    goto B0

B1(9) --> B0   (dom: B0)
  11   0    i34    1
. 12   0    i35    a26[i32] := i34                 prime[i] = true
  13   0    i37    i32 + i34                       i = i + 1
. 13   0     38    goto B0

B0(16) --> B1 B2   (dom: B10) - loop header
  ------------------
  0: i29 = [i19, i29] alias i19  // max
  1: a30 = [a26, a30] alias a26  // prime
  2: i31 = [?, i31] alias ?      // p
  3: i32 = [i27, i37]            // i
  ------------------
. 18   0     33    if i32 < i19 then B1 else B2    if i < max then B1 else B2

B2(21) --> B3   (dom: B0)
  21   0    i39    2                               p = 2
. 23   0     40    goto B3

B4(26) --> B5   (dom: B3)
  26   0    a46    <unloaded class>                System.out.println(p);
. 26   0    a47    a46._-1
  30   0    r1     := a47
  30   0    r0     affect
. 30   0    v50    r1.invokevirtual(i43)
. 35   0     51    goto B5

B6(38) --> B5   (dom: B5)
  40   0    i57    0
. 41   0    i58    a26[i55] := i57                 prime[i] = false
  44   0    i59    i55 + i43                       i = i + p
. 45   0     60    goto B5

B5(46) --> B6 B7   (dom: B4) - loop header
  ------------------
  0: i52 = [i41, i52] alias i19  // max
  1: a53 = [a42, a53] alias a26  // prime
  2: i54 = [i43, i54] alias i43  // p
  3: i55 = [i43, i59]            // i
```

```
  ------------------
. 48   0     56    if i55 < i19 then B6 else B7   if i < max then B6 else B7

B7(51) --> B8 B3   (dom: B5) - loop header
  ------------------
  0: i61 = [i41, i61] alias i19  // max
  1: a62 = [a42, a62] alias a26  // prime
  2: i63 = [i43, i66]            // p
  3: i64 = [i55, i64] alias i55  // i
  ------------------
  51   0    i65    1
  51   0    i66    i63 + i65                      p = p + 1
. 56   0     67    if i66 >= i19 then B3 else B8  if p >= max then B3 else B8

B8(59) --> B7 B3   (dom: B7)
  61   0    i68    a26[i66]
  62   0    i69    0
. 62   0     70    if i68 == i69 then B7 else B3  if prime[p] == false then B7 else B3

B3(65) --> B4 B9   (dom: B2) - loop header
  ------------------
  0: i41 = [i19, i41, i41] alias i19  // max
  1: a42 = [a26, a42, a42] alias a26  // prime
  2: i43 = [i39, i66, i66]            // p
  3: i44 = [i32, i55, i55]            // i
  ------------------
. 67   0     45    if i43 < i19 then B4 else B9   if p < max then B4 else B9

B9(70) -->   (dom: B3)
. 70   0    v71    return
```

IR after register allocation

```
B10(0) --> B0   (dom: B12)                        r8 := max
  0    0    r8     L0
. 1    0    r0     new boolean array [r8]        eax := new boolean[r8]
  1    0    r3     affect
  1    0    r1     affect
  1    0    r2     affect
  1    0    r6     affect
  1    0    r7     affect
  1    0    r3     := r0                         ebx := eax
  4    0    r1     0                             ecx := 0
. 6    0     28    goto B0

B1(9) --> B0   (dom: B0)
  11   0    i34    1
. 12   0    i35    r3[r1] := i34                 ebx[ecx] := 1
  13   0    r1     r1 + i34                      ecx = ecx + 1
. 13   0     38    goto B0

B0(16) --> B1 B2   (dom: B10) - loop header
. 18   0     33    if r1 < r8 then B1 else B2    if ecx < r8 then B1 else B2

B2(21) --> B3   (dom: B0)
  21   0    r2     2                             edx := 2
. 23   0     40    goto B3

B4(26) --> B5   (dom: B3)
  26   0    r0     <unloaded class>              System.out.println(p)
. 26   0    r1     r0._-1
  30   0    r1     := r1
  30   0    r0     affect
. 30   0    v50    r1.invokevirtual(r2)
  30   0    r1     := r2                         ecx := edx
. 35   0     51    goto B5
```

```
B6(38) --> B5   (dom: B5)
  40   0    i57    0
. 41   0    i58    r3[r1] := i57                  ebx[ecx] := 0
  44   0    r1     r1 + r2                        ecx := ecx + edx
. 45   0     60    goto B5

B5(46) --> B6 B7   (dom: B4) - loop header
. 48   0     56    if r1 < r8 then B6 else B7     if ecx < r8 then B6 else B7

B7(51) --> B8 B3   (dom: B5) - loop header
  51   0    i65    1
  51   0    r2     r2 + i65                       edx := edx + 1
. 56   0     67    if r2 >= r8 then B3 else B8    if edx >= r8 then B3 else B8

B8(59) --> B7 B3   (dom: B7)
  61   0    r0     r3[r1]
  62   0    i69    0
. 62   0     70    if r0 == i69 then B7 else B3   if ebx[edx] == 0 then B7 else B3

B3(65) --> B4 B9   (dom: B2) - loop header
. 67   0     45    if r2 < r8 then B4 else B9     if edx < r8 then B4 else B9

B9(70) -->   (dom: B3)
. 70   0    v71    return
```