Linear Scan Register Allocation in the Context of SSA Form and Register Constraints[1]

Hanspeter Mössenböck and Michael Pfeiffer

University of Linz, Institute of Practical Computer Science
{moessenboeck,pfeiffer}@ssw.uni-linz.ac.at

Abstract. Linear scan register allocation is an efficient alternative to the widely used graph coloring approach. We show how this algorithm can be applied to register-constrained architectures like the Intel x86. Our allocator relies on static single assignment form, which simplifies data flow analysis and tends to produce short live intervals. It makes use of lifetime holes and instruction weights to improve the quality of the allocation. Our measurements confirm that linear scan is several times faster than graph coloring for medium-sized to large programs.

1 Introduction

Register allocation is the task of assigning registers to the variables and temporaries of a program. It is crucial for the efficiency of the compiled code. The standard algorithm for register allocation is based on *graph coloring* [4, 3]: it builds an interference graph, in which the nodes represent the values in a program. An edge is drawn between two values if they are live at the same time. The graph is then colored such that adjacent nodes get different colors. If colors are viewed as registers we get a register allocation in which two values are kept in different registers if they are live at the same time.

There are situations, however, in which graph coloring is too slow, for example in a just in time (JIT) compiler that translates an intermediate program representation to machine code at load time or even at run time. JIT compilers must do their job in almost no time but should still produce high quality code. This conflict has led to a new register allocation technique that is called *Linear Scan* [10,11,13]. It assigns registers to values in a single linear scan over the live intervals of all values in a program. A live interval of a value *v* is the range of instructions starting at the defining instruction and ending at the instruction where *v* was used for the last time. If the live intervals of two values overlap, the values cannot reside in the same register. Although graph coloring leads to a slightly better register allocation than linear scan, the latter runs several times faster and is therefore an attractive register allocation technique in JIT compilers.

This paper describes an implementation of the linear scan register allocation technique making two contributions: Firstly, and in contrast to [10,11,13,8], we base

[1] This work was supported by Sun Microsystems, California.

R. N. Horspool (Ed.): CC 2002, LNCS 2304, pp. 229-246, 2002.

our allocator on programs in *static single assignment form* (SSA form). This simplifies data flow analysis and tends to produce shorter live intervals but requires modifications to the original linear scan algorithm. Secondly, we show how linear scan can be applied to register-constrained architectures such as the Intel x86. While [10,11,13 8] describe the algorithm for RISC architectures, a CISC machine like the Intel x86 requires modifications to the basic algorithm because of its two-address instructions and the fact that some operations expect or deliver values in specific registers.

The work described in this paper was done in a joint project with Sun Microsystems, in which their Java HotSpot™ client compiler [7] was extended with SSA form, register allocation and various optimizations. The HotSpot client compiler is a JIT compiler that is invoked for frequently called methods. Our modified compiler builds a control flow graph from the bytecodes of the method, translates the bytecodes to intermediate instructions of a register machine, brings them in SSA form (eliminating loads and stores for local variables), performs global common subexpression elimination and register allocation, and finally generates code for the Intel x86. The first version of our compiler used a graph coloring register allocator. Since this was not fast enough, we reimplemented the allocator using the linear scan technique.

Section 2 of this paper describes the original linear scan algorithm both in its simple form and in a refined form in which lifetime holes are exploited to fill them with other live intervals. We also explain how SSA form affects the computation of live intervals. Section 3 explains the data structures on which our algorithm relies and Section 4 describes how the intermediate code is prepared for register allocation. In Section 5 we explain our linear scan technique taking the peculiarities of the Intel architecture into account. Section 6 evaluates the complexity of our algorithm, compares it with related approaches and shows some measurements. Finally, Section 7 summarizes the results.

2 Linear Scan Register Allocation

Linear scan was introduced by Poletto et al. [10, 11] as an alternative to graph coloring allocation. It computes the live intervals of values in a program and scans them sequentially to find overlaps. Non-overlapping intervals can be assigned the same register. Since the live interval of a value *v* may contain holes in which *v* is not live, a refined version of this algorithm (called *second-chance binpacking*) was described by Traub et al. [13]. Although more complicated, this algorithm results in a better usage of registers. It also splits live intervals so that a value may reside in different registers during its lifetime. Both algorithms, however, do not take into account, that many optimizing compilers keep the intermediate program representation in SSA form. Therefore Section 2.3 describes how SSA form affects the linear scan allocation technique.

2.1 Basic Algorithm

The live interval of a value *v* is the range of instruction numbers [*i*, *j*[such that *i* is the instruction where *v* starts to live and *j* is the instruction where it ends living. The value

v may still be used at *j* but it does not interfere with another value defined at *j*. Thus the interval is open on the right-hand side. The instructions are numbered consecutively through all basic blocks in a topological order of the control flow graph without backward edges. The live variable information is obtained by data-flow analysis [1]. Fig.1 shows an example of four live intervals computed from a linear sequence of instructions.

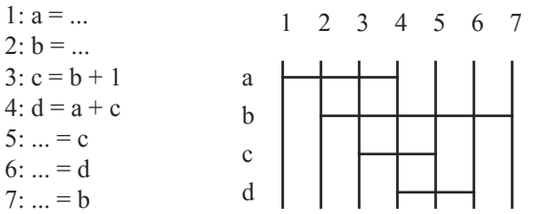

Fig. 1. A simple instruction sequence and its live intervals

The linear scan algorithm traverses all intervals in the order of increasing start points maintaining a list, *active*, which contains those intervals that overlap the start point of the current interval. Initially all registers are free. For every interval *i* the algorithm performs the following steps:

- If there are live intervals *j* in *active* that already expired before *i* begins (i.e., $j.end \leq i.beg$), remove them from active and add *j.reg* to the set of free registers.
- If there are still free registers, assign one of them to *i* and add *i* to *active*. If there are no free registers, spill the interval with the largest end point among *i* and all intervals in *active*. If an interval from *active* was spilled, assign its register to *i*, and add *i* to *active*.

Assuming that we have 2 registers, r1 and r2, the algorithm processes the intervals of Fig. 1 as follows:

interval	free	active	action
a	r1, r2	-	assign r1 to *a*; make *a* active
b	r2	a_{r1}	assign r2 to *b*; make *b* active
c	-	a_{r1}, b_{r2}	spill *b* since it ends after *c*; make r2 free
	r2	a_{r1}	assign r2 to *c*; make *c* active
d	-	a_{r1}, c_{r2}	remove *a* from active (expired); make r1 free
	r1	c_{r2}	assign r1 to *d*; make *d* active

In this example, *a* and *d* end up in r1 and *c* in r2. The value *b* was first assigned to a register, but later it was spilled and thus resides in memory.

2.2 Holes in Live Intervals

Between the first definition and the last use of a value there may be points at which the value is not live. Consider for example the program in Fig. 2 in which we look at the live intervals of the variables *a* and *b*.

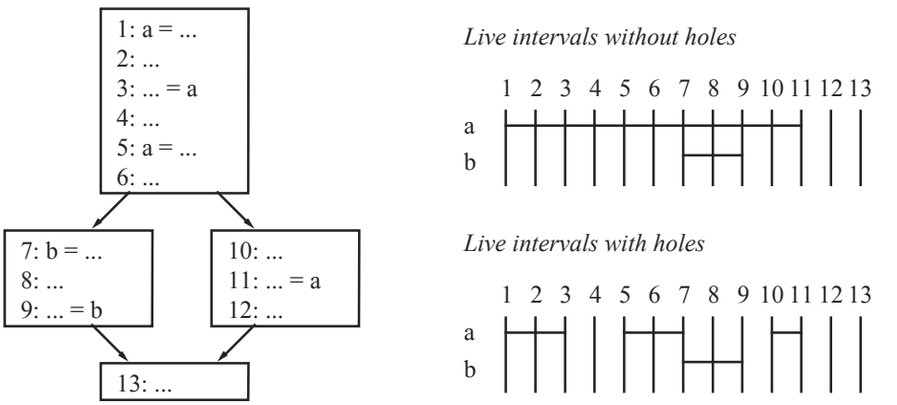

Fig. 2. Holes in live intervals

The live interval of *a* has two holes, the first one between instructions 3 and 5 where *a* is not used any more before it is redefined, and the second one between instructions 7 and 9 resulting from the order in which we numbered the instructions. Since the interval of *b* exactly falls into such a hole it can be assigned the same register as the interval of *a*.

Keeping track of holes in live intervals makes the linear scan algorithm more complicated but it pays off since we get more values into registers. The refinement of linear scan with lifetime holes was described by Traub et al. [13]. The idea is also used in our algorithm, which we will describe in Section 5.

Traub et al. add a second improvement to the linear scan algorithm. If an interval is assigned a register but gets spilled later, a spill instruction is inserted at that point and the interval is split into two halves. In the first half the value resides in a register, in the second half it resides in memory unless it is selected for being reloaded into a register later. They call their algorithm *second-chance binpacking* because a spilled value gets a second chance to reside in a register later. We did not use this idea in our algorithm, because our live intervals tend to be shorter due to SSA form as we will describe in Section 2.3.

In Traub's algorithm the decision which interval is spilled if the allocator runs short of registers is based on weights that are computed from the distance to the next use of a value and the nesting level. We use similar weights based on the number of accesses to the value and the nesting level.

2.3 Live Intervals and Static Single Assignment Form

Many optimizing compilers keep the intermediate program in Static Single Assignment Form (SSA form) [6, 9] because it simplifies data flow analysis and optimizations. In SSA form, every assignment introduces a new and uniquely named variable so that there is never more than one assignment statement per variable. Thus, given a variable name one immediately knows where this variable received a value. If two variables have the same name they must also have the same value. Fig. 3 shows a statement sequence and its transformation to SSA form.

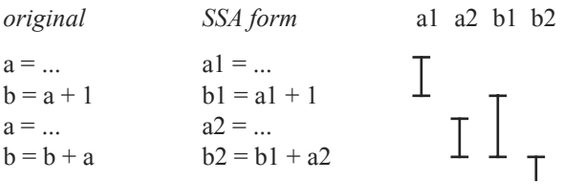

Fig. 3. A statement sequence and its SSA form

If there are multiple assignments to a variable we get several smaller live intervals—one for every copy of this variable—instead of a single large interval for the original variable. Each of these intervals can reside in a different register, decreasing register pressure. Thus, the need for splitting intervals as it is done in second-chance binpacking is not as important as without SSA form.

If two values of a variable flow together at a basic block, SSA form requires the insertion of a so-called ϕ-function (or phi-function), which is a pseudo instruction that creates yet another copy of this variable. This is shown in Fig. 4.

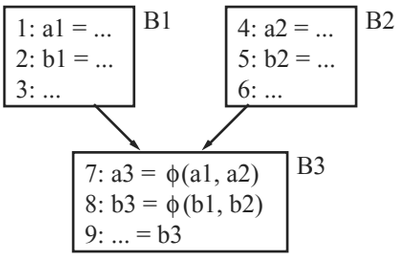

Fig. 4. ϕ-functions in a merge block

The ϕ-function in instruction 8 means that if the control flow comes via the left branch *b3* becomes *b1*, otherwise *b3* becomes *b2*. It creates a single definition point for the value of *b* that flows from here and is used in instruction 9.

Unfortunately, ϕ-functions become a problem in the computation of live intervals. For example, the live interval of *b1* is [2,4[, [7,8[and the live interval of *b2* is [5,7[, [7,8[. This would lead to an overlap of the two intervals in instruction 7 forcing them into different registers. However, this is exactly what we do not want, since *b1* and *b2* are two values of the same variable and should end up in the same register if possible so that the ϕ-function in instruction 8 can be eliminated and the same register can be used for *b1*, *b2* and *b3*.

In fact, *b1* and *b2* are not live at the same time in instruction 7. *b1* is only live if we come via the left branch and *b2* is only live if we come via the right branch. If we could insert move instructions at the end of *B1* and *B2* and eliminate the ϕ-functions in *B3* the overlap would be removed (Fig. 5a). However, this would invalidate SSA form. The solution is to insert move instructions while keeping the ϕ-functions, and to treat ϕ-functions as special cases for liveness analysis (Fig. 5b).

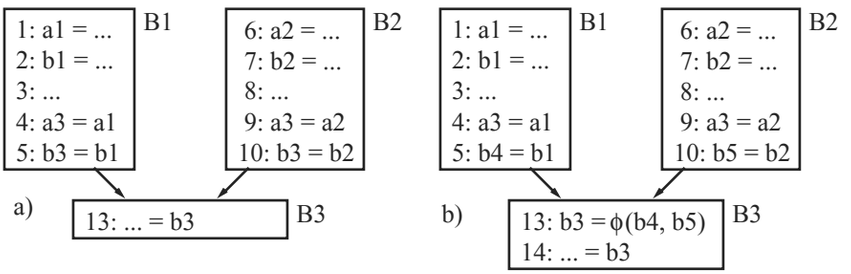

Fig. 5. Move instructions are inserted for the operands of ϕ-functions

In Fig. 5b the live interval of *b1* is [2,5[, the live interval of *b2* is [7,10[, and the live interval of *b3* is [14,14[(ϕ-functions are excluded from live intervals as described in Section 4.3). There is no overlap any more and *b1*, *b2* and *b3* can be put into the same register. By coalescing (Section 4.4) we can possibly also eliminate instructions 5 and 10. If only *b1* and *b3* can be put into the same register but not *b2* (e.g., because this register is used for some other purpose in *B2*) instruction 10 remains a register move.

3 Data Structures

The data structures for basic blocks are as described in Fig. 6. Every block has pointers to its successors and predecessors as well as a pointer to its first and last instruction and to the first ϕ-function (ϕ-functions precede the ordinary instructions).

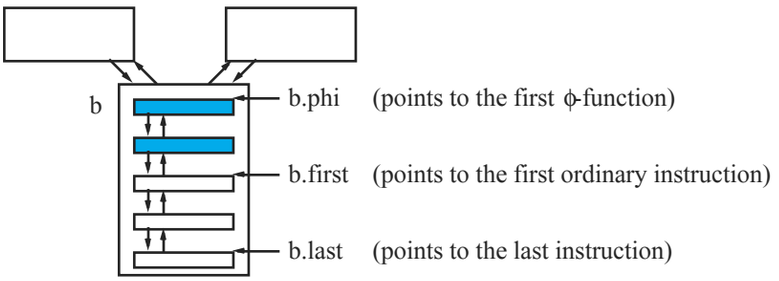

Fig. 6. Data structures for basic blocks

Every instruction *i* has an instruction number *i.n* and a field *i.reg* that holds the register that the allocator assigns to the value created by *i*. The *reg* fields are initialized to -1 (*any*) meaning that no register was assigned so far. If an instruction *i* should produce a value in a specific register *r* (as it is sometimes the case on Intel processors) *i.reg* is initialized to *r* ($r \geq 0$) and the register allocator does not overwrite this value. This technique is sometimes called *precoloring* and is described in more detail for example in [5].

When the bytecodes are transformed to instructions of the intermediate representation (IR) we eliminate stores and loads for local variables (except for loads of parameters). Every instruction produces a value that is stored in a new virtual register,

assuming that we have an unlimited number of virtual registers. Fig. 7 shows an example of a Java function and the IR instructions generated for it.

```
int f(int a) {          1: i1 = load a
  int b = a * a;        2: i2 = i1 * i1
  return b + a;         3: i3 = i2 + i1
}                       4: ret i3
```

Fig. 7. Every instructions produces a value in a virtual register

Instructions 1, 2 and 3 produce a value in a new virtual register (*i1*, *i2*, *i3*), thus the IR is in SSA form. The *reg* fields of these instructions are initialized to *any*; the register allocator will assign physical registers to them later. Instruction 4 does not produce a value. Nevertheless it has a *reg* field, which the register allocator ignores. Stores and loads of the variable *b* have been eliminated.

Live intervals are stored as a sorted sequence of sub-intervals (*ranges*) that are open on the right-hand side. For example, the interval [3,5[, [10,15[, [18,20[consists of three ranges. The first range starts at instruction 3 where the value is live and ends at instruction 5 where the value may be used but is not live any more when a new value is defined there. All live intervals are kept in an array *interval* (see Fig. 8). The live interval of a value defined in instruction *i* can be found in *interval[i.n]*.

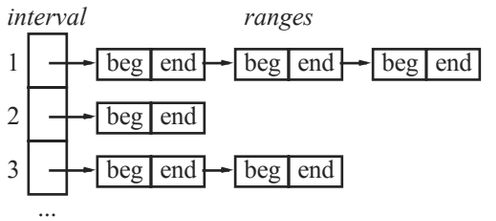

Fig. 8. Live intervals and their ranges

Note that the array *interval* is automatically sorted in the order of increasing start points of the live intervals, since every instruction (except return, goto, etc.) creates a new value and is the start of this value's live interval.

We say that the live interval of a value *v* is *fixed* if $v.reg \geq 0$ prior to register allocation. Fixed intervals with the same register are joined (see Section 4.4) into a single interval. In order to make sure that fixed intervals of the same register do not overlap, we insert moves before or after the instructions that generate or use values in fixed registers. If an instruction

```
x = y op z
```

requires *y* to be in a specific register *r*, we insert a move instruction in front of it

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

and set *u.reg* to *r*. If the instruction leaves its result *x* in a specific register *r*, we insert a move instruction after it

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

and set *v.reg* to *r*. The moves make sure that fixed intervals of the same register do not overlap. Many of these moves can be eliminated by coalescing (see Section 4.4).

Finally we use *live sets* that we obtain by live variable analysis [1] and store them as bit sets. Live variable analysis is considerably simplified by SSA form as described for example in [9]. Every basic block *b* stores in *b.live* the set of values that are live immediately before the instruction *b.first*.

4 Preparing the IR for Linear Scan

4.1 Generating Moves for ϕ-Operands

As explained in Section 2, we have to generate moves for the operands of ϕ-functions. Fig. 9 shows the result of this process and algorithm GENMOVES() explains the details. Since there is no block in the original graph to hold instruction 6 we have to insert one.

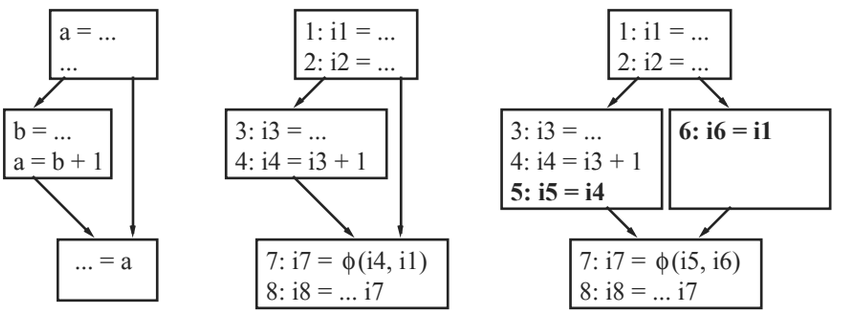

Fig. 9. Move instructions 5 and 6 are generated for the ϕ-function 7

```
GenMoves()
          for all blocks b do
            for all predecessors p of b do
              if b.no_of_predecessors > 1 and
p.no_of_successors > 1 then
                insert a new block n between p and b
              else
                n ← p
              for each ϕ-function phi of b do
                i ← new RegMove(phi.opd(p))  // the ϕ-
operand corresponding to p
                phi.opd(p) ← i
                append i to n
                join i with phi     // see Section 4.4
```

4.2 Numbering the Instructions

After moves have been inserted for ϕ-operands the instructions have to be numbered consecutively. In order to do that we traverse all basic blocks in topological order so that a block *b* is only visited after all its predecessors that have forward branches to *b* have been visited. Fig. 10 shows some valid visit sequences.

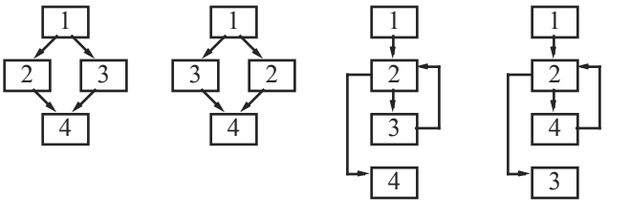

Fig. 10. Valid visit sequences of blocks for instruction numbering

4.3 Computing Live Intervals

In SSA form there is only one assignment to every variable. This assignment marks the beginning of the variable's lifetime. The variable lives in all paths from its definition to its last use. For every block *b* and every variable *v* we compute a range $r_{v,b}$ that denotes the live range of *v* in *b* as shown in Fig. 11.

If *v* is live at the end of *b* it must have been defined either in *b* or in some predecessor block *p*. If *v* was defined in *p* then $r_{v,b}$ begins at *b.first* and ends after *b.last*. If it was defined in *b* then $r_{v,b}$ begins at the instruction *v* and ends after *b.last*.

If *v* is not live at the end of *b* but is used in *b* then $r_{v,b}$ begins as described above and ends at the last use of *v* in *b*. The last use of a variable is detected using the live sets: the instructions of *b* are traversed in reverse order; if a variable *v* is used at instruction *i* but is not in the live set at the end of *i* then *i* is the last use of *v*.

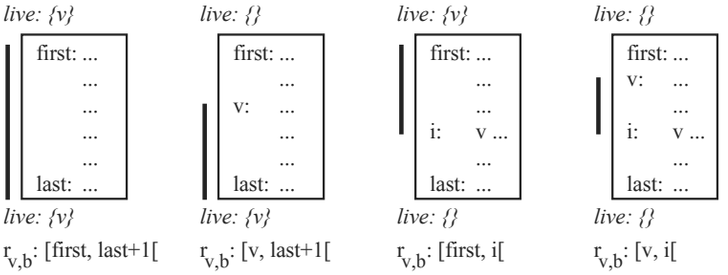

Fig. 11. Computation of the live range $r_{v,b}$ of a variable *v* in block *b*

The live interval of a ϕ-function *i* in block *b* does not start at *i* but at the first ordinary instruction in this block (*b.first*). This avoids undesired conflicts between the ϕ-functions of a block. It is an invariant of our algorithm that the defining instruction of a ϕ-function never appears in a live interval.

The algorithm ADDRANGE(*i*, *b*, *end*) computes the range $r_{i,b}$ of instruction *i* in block *b* (according to Fig. 11) assuming that we already know that *i* ends living at the instruction with the number *end*. It then adds the range to the live interval of *i*.

```
AddRange(i: Instruction; b: Block; end: integer)
          if b.first.n • i.n • b.last.n then range ←
[i.n, end[ else range ← [b.first.n, end[
          add range to interval[i.n]  // merging
adjacent ranges
```

If possible, adjacent ranges of the same live interval are merged. For example, the ranges [1,3[, [3,7[are merged into a single range [1,7[.

The algorithm BuildIntervals() traverses the control flow graph in an arbitrary order, finds out which values are live at the end of every block, and computes the ranges for these values as described above.

```
BuildIntervals()
          for each block b do
            live ← {}
            for each successor s of b do
              live ← live ∪ s.live
              for each ϕ-function phi in s do
                live ← live - {phi} ∪ {phi.opd(b)}
            for each instruction i in live do AddRange(i,
b, b.last.n+1)
            for all instructions i in b in reverse order
do
              live ← live - {i}
              for each operand opd of i do
                if opd ∉ live then
                  live ← live ∪ {opd}
                  AddRange(opd, b, i.n)
```

Fig. 12 shows a sample program in source code and in intermediate representation with a ϕ-function for the value *d* and corresponding move instructions in the predecessor blocks. Fig. 13 shows the live intervals that are computed for this program by BuildIntervals(). Note that the live intervals of *i2* and *i11* exclude instruction 11 since ϕ-functions never appear in live intervals.

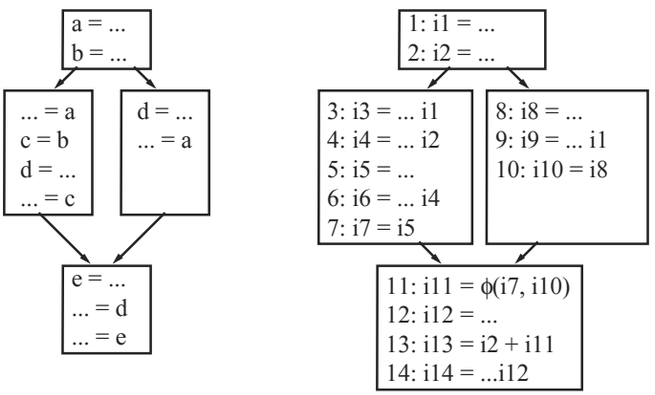

Fig. 12. Sample program in source code and in intermediate representation

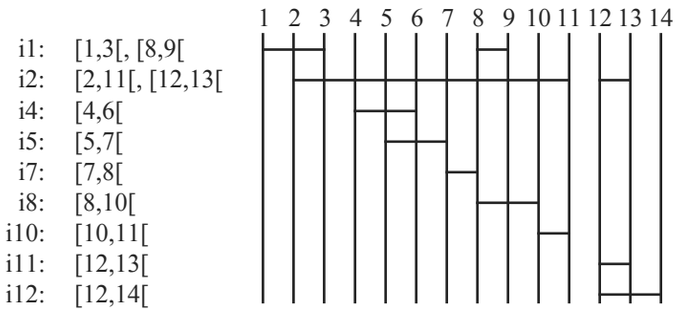

Fig. 13. Live Intervals computed from the program in Fig. 12

4.4 Joining Values

Sometimes we want that two values go into the same register, for example:

- a ϕ-function and its operands (so that the ϕ-function can be eliminated);
- the left-hand and right-hand sides of register moves (so that the move can be eliminated);
- the first operand *y* and the result *x* of a two-address instruction $x = y$ op z as it is required by the Intel x86 architecture.

If the live intervals of the two values do not overlap we can join them, i.e. we merge their intervals so that the register allocator assigns the same register to them. This is also called *coalescing* ([2]). Note that coalescing leads to longer intervals possibly introducing additional conflicts that force more values into memory. Currently we do not try to minimize such conflicts although it could be done as described for example in [2].

A group of joined values is represented by only one of those values, its *representative*, using a *union-find* algorithm ([12]). Every instruction *i* has a field *i.join*, which points to its representative. Initially, *i.join* = *i* for all instructions *i*. If we have three values, *a*, *b*, and *c*, and if we join *b* with *c*, and then *a* with *b* we get a group with *c* as its representative as shown in Fig. 14.

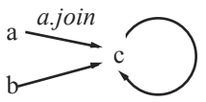

Fig. 14. A group of four joined values with *c* as its representative

Taking into account that certain values have to be in specific registers we can join two values *x* and *y* only if they are *compatible*, i.e. if

- both do not have to be in specific registers, or
- both have to be in the *same* specific register, or
- *x* must be in a specific register and the interval of *y* does not overlap any other interval to which *x.reg* has been assigned (or vice versa). More formally:
- $x.reg \geq 0 \wedge \neg$ (∃ interval *iv*: $iv.reg = x.reg$ & *interval*[*y.n*] overlaps *iv*) ∨
- $y.reg \geq 0 \wedge \neg$ (∃ interval *iv*: $iv.reg = y.reg$ & *interval*[*x.n*] overlaps *iv*)

The algorithm JOIN(*x*, *y*) joins the two values *x* and *y* if they are compatible:

```
JOIN(x, y: Instruction)
          i ← interval[REP(x).n]
          j ← interval[REP(y).n]
          if i ∩ j = {} and x and y are compatible then
            interval[REP(y).n] ← i ∪ j
            drop interval[REP(x).n]
            x.join ← REP(y)
REP(x: Instruction): Instruction
          if x.join = x then return x else return
REP(x.join)
```

If we look at the program in Fig. 12 we can join the values 11, 7 and 10 (the ϕ-function and its operands) as well as 5 with 7 and 8 with 10 (the left- and right-hand sides of the register moves). The resulting intervals are shown in Fig. 15.

The live intervals are now in a form that can be used for linear scan register allocation. This will be described in the next section.

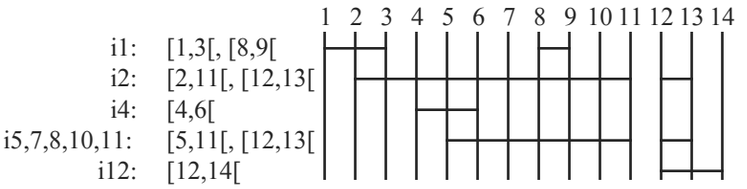

Fig. 15. Live intervals of Fig. 13 after join operations

5 The Linear Scan Algorithm

The register allocator has to map an unbounded number of virtual registers to a small set of physical registers. If a value cannot be mapped to a register it is assigned to a memory location. Many instructions of the Intel x86 allow memory operands so there is a good chance that this value never has to be loaded into a register. If it has to be in a register, however, we load it into a scratch register (one scratch register is excluded from register allocation). If an instruction needs more than one scratch register the code generator spills one of the registers and uses it as a temporary scratch register. When the spilled value is needed again the code generator reloads it into the same register as before. Note that spilling instructions are emitted by the code generator and not by the register allocator, which only decides if a value should reside in a register or in memory.

The register allocator assumes that all live intervals of a method are sorted in the order of increasing start points. It makes the first interval the *current interval* (*cur*) and divides the remaining intervals into the following four sets:

- *unhandled* set: all intervals that start after *cur.beg;*
- *handled* set: all intervals that ended before *cur.beg* or were spilled (see below);
- *active* set: all intervals where one of their ranges overlaps *cur.beg*;
- *inactive* set: all intervals where *cur.beg* falls into one of their holes.

Throughout register allocation the following invariants hold: Registers assigned to intervals in the *handled* set are free; registers assigned to intervals in the *active* set are not free; a register assigned to an interval *i* in the *inactive* set is either free or occupied by a currently active interval *j* that does not overlap *i* (i.e. fully lies in a hole of *i*). When *i* becomes active again, *j* already ended so that *i* can reclaim its register.

The algorithm LINEARSCAN() repeatedly picks the first interval *cur* from *unhandled* updating the sets *active*, *inactive* and *handled* appropriately.

```
LinearScan()
          unhandled ← all intervals in increasing order
of their start points
          active ← {}; inactive ← {}; handled ← {}
          free ← set of available registers
          while unhandled • {} do
            cur ← pick and remove the first interval
from unhandled
            //----- check for active intervals that
expired
            for each interval i in active do
              if i ends before cur.beg then
                move i to handled and add i.reg to free
              else if i does not overlap cur.beg then
                move i to inactive and add i.reg to free
            //----- check for inactive intervals that
expired or become reactivated
            for each interval i in inactive do
              if i ends before cur.beg then
                move i to handled
              else if i overlaps cur.beg then
                move i to active and remove i.reg from
free
            //----- collect available registers in f
            f ← free
            for each interval i in inactive that
overlaps cur do f ← f - {i.reg}
            for each fixed interval i in unhandled that
overlaps cur do f ← f - {i.reg}
            //----- select a register from f
            if f = {} then
              ASSIGNMEMLOC(cur)  // see below
            else
              if cur.reg < 0 then cur.reg ← any register
in f
              free ← free - {cur.reg}
              move cur to active
```

If we cannot find a free register for *cur* we assign a memory location to either *cur* or to any of the other currently active or inactive intervals, whichever has a lower weight. The weights are computed from the accesses to the intervals weighted by the nesting level in which the accesses occur. Here is the algorithm:

```
AssignMemLoc(cur: Interval)
          for all registers r do w[r] ← 0  // clear
register weights
          for all intervals i in active, inactive and
(fixed) unhandled do
            if i overlaps cur then w[i.reg] ← w[i.reg] +
i.weight   // if fixed i.weight = •
          find r such that w[r] is a minimum
          if cur.weight < w[r] then
            assign a memory location to cur and move cur
to handled
          else   // assign memory locations to the
intervals occupied by r
            move all active or inactive intervals to
which r was assigned to handled
            assign memory locations to them
            cur.reg ← r
            move cur to active
```

Table 1 shows how LinearScan() works through the intervals of Fig. 15 assuming that we have 2 registers available. The weights of the intervals can be computed from the accesses to values (see Fig. 12) and are as follows: i1:3, i2:3, i4:2, i5:7, i12:2 (accesses in a ϕ-function are neglected).

Table 1. Simulation of LinearScan() for the intervals of Fig. 15

cur	*action*	*free*	*unhandled*	*active*	*inactive*	*handled*
	initialize	r1, r2	1, 2, 4, 5, 12	-	-	-
1	assign r1 to interval 1	r2	2, 4, 5, 12	1_{r1}	-	-
2	assign r2 to interval 2	-	4, 5, 12	1_{r1}, 2_{r2}	-	-
4	move interval 1 to inactive	r1	5, 12	2_{r2}	1_{r1}	-
	assign r1 to interval 4	-	5, 12	2_{r2}, 4_{r1}	1_{r1}	-
5	put interval 2 into memoy	r2	12	4_{r1}	1_{r1}	2_m
	assign r2 to interval 5	-	12	4_{r1}, 5_{r2}	1_{r1}	2_m
12	move int. 1 and 4 to handled	r1	-	5_{r2}	-	1_{r1}, 2_m, 4_{r1}
	assign r1 to interval 12	-	-	5_{r2}, 12_{r1}	-	1_{r1}, 2_m, 4_{r1}

Interval 2 was put into memory because its weight (3) is less than the cumulated weights of intervals 1 and 4 that occupy the same register at that time (weight = 5) and of the current interval 5 (weight = 7). Fig. 16 shows the result of the register allocation for Fig. 15.

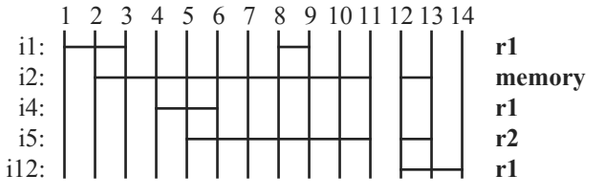

Fig. 16. Result of the register allocation with 2 available registers

6 Evaluation

6.1 Complexity

LINEARSCAN takes linear time to scan the intervals. For every interval it has to inspect the active, inactive and unhandled fixed sets in order to find overlaps. Since there cannot be more active intervals than registers, the length of the active set is bounded by the number of registers, which is a small constant. The length of the inactive set can come close to the total number of intervals, which would lead to a quadratic time complexity in the worst case. In practice, however, there are only very few inactive intervals (typically less than 2) at any point in time so the behavior is still linear. Finally, the number of unhandled fixed intervals is bounded by the number of available registers, because fixed intervals with the same register are joined into a single interval. Therefore, if n is the number of live intervals, the overall complexity of our algorithm is $O(n^2)$ in the worst case but linear in practice.

During preprocessing we have to generate moves for ϕ-functions. This takes time proportional to the number of ϕ-functions, which is smaller than n. Live intervals are generated in sorted order so we do not need a separate pass to sort them.

6.2 Comparison with Related Work

The novelty of our approach lies in the fact that it is applicable to programs in SSA form and that it can deal with values that have to reside in specific registers. The adaptations for SSA form are done in a preprocessing step in which moves are inserted into the instruction stream in order to neutralize the ϕ-functions. After this step, SSA form does not affect the linear scan register allocation since ϕ-functions do not show up in the live intervals any more.

In contrast to Poletto and Sarkar [11] our linear scan algorithm can deal with lifetime holes and fixed intervals, which makes it more complicated: In addition to the three sets *unhandled*, *handled* and *active* we need a fourth set, *inactive*, to hold intervals with a hole into which the start of the current interval falls. We also have to exclude registers that are occupied by overlapping fixed intervals from the register selection. Otherwise our algorithm is very close to the one described in [11].

Traub et al. [13] emit spill and reload instructions during register allocation eliminating a separate pass in which the instruction stream is rewritten. A spilled value can be reloaded into any free register later so that a value can reside in different registers during its life. While the ability to split long intervals is definitely an advantage, SSA form tends to produce shorter intervals from the beginning. For example, the live interval of the value v in Fig. 17a is [1,9[. In SSA form (Fig. 17b) the interval is split into 4 intervals ([1,2[, [4,7[, [9,10[, [12,12[), each of which can reside in a different register. Therefore the need for interval splitting seems not to be as urgent as without SSA form.

Traub's algorithm has to insert register moves at certain block boundaries because values can be in different locations at the beginning and the end of a control flow edge. In a similar way, we insert moves for the operands of ϕ-functions (instructions 7 and 10 in Fig. 17b) and eliminate unnecessary moves by coalescing values later.

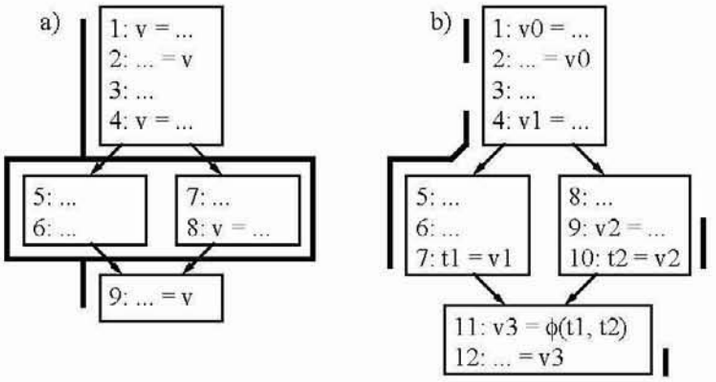

Fig. 17. Length of live intervals a) without and b) with SSA form

6.3 Measurements

The first version of our compiler used a graph coloring register allocator, which we later replaced by a linear scan allocator. In order to compare their speed we compiled the first 1000 classes of the Java class library.

Fig. 18 shows the time used for register allocation (in milliseconds) depending on the size of the compiled methods (in bytecodes). We can see that linear scan has a nearly linear time behavior and remains efficient even for larger methods, whereas the time for graph coloring tends to increase disproportionally. For large programs linear scan is several times faster than graph coloring.

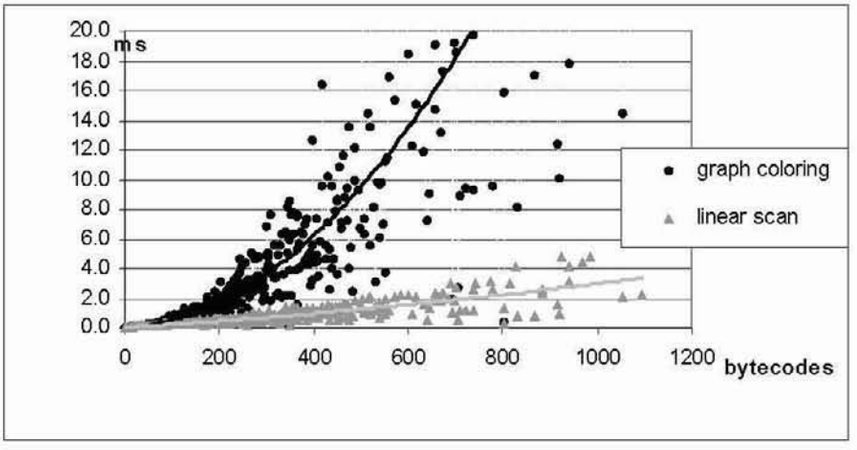

Fig. 18. Run time of graph coloring vs. linear scan

7 Summary

We described how to adapt the linear scan register allocation technique for programs in SSA form. Due to SSA form the live intervals of most values become short and allow us to keep the same variable in different registers during its lifetime without splitting live intervals. We also showed how to deal with values that have to reside in specific registers as it is common in many CISC architectures.

Acknowledgements. We would like to thank Robert Griesemer, Srdjan Mitrovic and Kenneth Russell from Sun Microsystems for supporting our project as well as the anonymous referees for providing us with valuable comments on an early draft of this paper.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
2. Appel, A.W.: Modern Compiler Implementation in Java. Cambridge University Press (1998)
3. Briggs, P., Cooper, K., Torczon, L: Improvements to Graph Coloring Register Allocation. ACM Transactions on Programming Languages and Systems 16, 3 (1994) 428-455
4. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register Allocation via Coloring. Computer Languages 6 (1981) 47-57
5. Chow F. C., Hennessy J. L.: The Priority-Based Coloring Approach to Register Allocation. ACM Transactions on Programming Languages and Systems 12, 4 (1990) 501-536
6. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Transactions on Programming Languages and Systems 13, 4 (1991) 451 - 490
7. Griesemer, R., Mitrovic, S.: A Compiler for the Java HotSpot™ Virtual Machine.. In Böszörmenyi et al. (ed.): The School of Niklaus Wirth. dpunkt.verlag (2000)
8. Johansson, E., Sagonas, K.: Linear Scan Register Allocation in the HiPE Compiler. International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001), Kiel, Germany, September 13-15, 2001
9. Mössenböck, H.: Adding Static Single Assignment Form and a Graph Coloring Register Allocator to the Java HotSpot! Client Compiler. TR-15-2000, University of Linz, Institute of Practical Computer Science, 2000
10. Poletto, M., Engler, D.R., Kaashoek, M.F.: A System for Fast, Flexible, and High-Level Dynamic Code Generation. Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation, Las Vegas (1997) 109-121

11. Poletto, M., Sarkar, V.: Linear Register Allocation. ACM Transactions on Programming Languages and Systems 21, 6 (1999) 895-913
12. Sedgewick, R.: Algorithms, 2nd edition. Addison Wesley (1988)
13. Traub, O., Holloway, G., Smith, M.D.: Quality and Speed in Linear-Scan Register Allocation. Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (1998) 142-151