Optimized Interval Splitting in a Linear Scan Register Allocator*

Christian Wimmer
Institute for System Software
Johannes Kepler University Linz
Linz, Austria
wimmer@ssw.jku.at

Hanspeter Mössenböck
Institute for System Software
Johannes Kepler University Linz
Linz, Austria
moessenboeck@ssw.uni–linz.ac.at

ABSTRACT

We present an optimized implementation of the linear scan register allocation algorithm for Sun Microsystems' Java HotSpot^{TM} client compiler. Linear scan register allocation is especially suitable for just-in-time compilers because it is faster than the common graph-coloring approach and yields results of nearly the same quality.

Our allocator improves the basic linear scan algorithm by adding more advanced optimizations: It makes use of lifetime holes, splits intervals if the register pressure is too high, and models register constraints of the target architecture with fixed intervals. Three additional optimizations move split positions out of loops, remove register-to-register moves and eliminate unnecessary spill stores. Interval splitting is based on use positions, which also capture the kind of use and whether an operand is needed in a register or not. This avoids the reservation of a scratch register.

Benchmark results prove the efficiency of the linear scan algorithm: While the compilation speed is equal to the old local register allocator that is part of the Sun JDK 5.0, integer benchmarks execute about 15% faster. Floating-point benchmarks show the high impact of the Intel SSE2 extensions on the speed of numeric Java applications: With the new SSE2 support enabled, SPECjvm98 executes 25% faster compared with the current Sun JDK 5.0.

Categories and Subject Descriptors

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

General Terms

Algorithms, Languages, Performance

Keywords

Java, compilers, just-in-time compilation, optimization, register allocation, linear scan, graph-coloring

*This work was supported by Sun Microsystems, Inc.

© ACM, 2005. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in the Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments, pp. 132–141.
VEE'05, June 11–12, 2005, Chicago, Illinois, USA.
http://doi.acm.org/10.1145/1064979.1064998

1. INTRODUCTION

Register allocation is one of the most profitable compiler optimizations. It is the task of assigning physical registers to local variables and temporary values. The most commonly used algorithm treats the task of register allocation as a graph-coloring problem [1][2]. It uses an interference graph whose nodes represent the values that should get a register assigned. Two nodes are connected with an edge if they are live at the same time, i.e. when they must not get the same register assigned. Then the graph is colored such that two adjacent nodes get different colors, whereby each color represents a physical register. The graph-coloring algorithm generates code of good quality, but is slow for just-in-time compilation of interactive programs because even heuristic implementations have a quadratic runtime complexity.

In comparison, the linear scan algorithm is simpler and faster and yields results of nearly the same quality. Lifetime intervals store the range of instructions where a value is live. Two intersecting intervals must not get the same register assigned. The algorithm assigns registers to values in a single linear pass over all intervals. The basic linear scan algorithm [12][13] can be extended to support holes in lifetime intervals and the splitting of intervals [17].

We implemented and extended the linear scan algorithm for the Java HotSpot^{TM} client compiler [4] of Sun Microsystems. The compiler is invoked by the Java HotSpot^{TM} VM [16] for frequently executed methods. The client compiler is designed as a fast compiler that omits time-consuming optimizations, achieving a low startup and response time. For this reason, the current product version of the client compiler uses only a local heuristic for register allocation.

In contrast, the Java HotSpot^{TM} server compiler [10] produces faster executing code, but at the cost of a more than ten times higher compilation time. This is not suitable for client applications because of the higher response time. Linear scan register allocation for the client compiler reduces the gap of peak performance between the two compilers without degrading the response time of the client compiler.

This paper presents the details of a successful implementation of the linear scan algorithm for a production-quality just-in-time compiler. The paper contributes the following:

- We introduce *use positions* to decide which interval is to be spilled. They mark instructions where a lifetime interval *should* or *must* have a register assigned.
- We present three fast optimizations—*optimal split positions*, *register hints* and *spill store elimination*—that improve the code quality without a data flow analysis.

- We compare our research compiler with the client compiler and the server compiler of the Sun JDK 5.0. The measurements show that the linear scan algorithm decreases the run time of compiled code without increasing the compilation time when compared with the product client compiler.

2. DATA STRUCTURES

Our version of the HotSpot™ client compiler uses two intermediate representations: When a method is compiled, the front end transforms bytecodes to the graph-based high-level intermediate representation (HIR) that uses SSA form [3][8]. Several optimizations are applied before the back end converts the HIR to the low-level intermediate representation (LIR). After linear scan register allocation, machine code is created from the LIR. More details of the compiler architecture can be found in [18]. The compiler can generate code for the Intel IA-32 architecture and the SPARC architecture of Sun. The examples and measurements in this paper are given for the IA-32 architecture.

2.1 Intermediate Representation

The linear scan algorithm operates on the LIR that is conceptually similar to machine code. It allows platform-independent algorithms that would be difficult to implement directly on machine code. Each basic block of the control flow graph stores a list of LIR instructions. Before register allocation, all basic blocks are sorted into a linear order: The control flow graph is flattened to a list using the standard reverse postorder algorithm. To improve the locality, all blocks belonging to a loop are emitted consecutively. Rarely executed blocks such as exception handlers are placed at the end of the method.

An operand of a LIR instruction is either a *virtual register*, a *physical register*, a *memory address*, a *stack slot* of the stack frame, or a *constant*. While the number of virtual registers is unlimited, the number of physical registers is fixed by the target architecture. When the LIR is generated, most operands are virtual registers. The register allocator is responsible for replacing all virtual registers by physical registers or stack slots.

2.2 Lifetime Intervals

For each virtual register, a *lifetime interval* is constructed that stores the lifetime of the register as a list of disjoint ranges. In the simplest case, there is only a single live range that starts at the instruction defining the register and ends at the last instruction using the register. More complicated intervals consist of multiple ranges. The space between two ranges is commonly referred to as a *lifetime hole*.

In order to create accurate live ranges, a data flow analysis is performed before the intervals are built. This is necessary to model the data flow in a non-sequential control flow. For example, an operand that is used once in a loop must be alive in the entire loop; otherwise it would not be available in further iterations.

2.3 Use Positions

The *use positions* of an interval refer to those instructions where the virtual register of this interval is read or written. They are required to decide which interval is to be split and spilled when no more physical registers are available. They are also used to determine when a spilled interval must be reloaded into a register.

Each use position has a flag denoting whether it requires the value of the interval to be in a register or not: If the use position *must* have a register, the register allocator guarantees that the interval has a register assigned at this position. If the interval was spilled to memory before this position, it is reloaded into a register. This avoids the reservation of a scratch register for temporary computations. If the use position *should* have a register, then the interval may be spilled. This allows the modeling of machine instructions of the IA-32 architecture that can handle memory operands.

Together with the lifetime intervals, use positions form a clear interface between the register allocator and the rest of the compiler. This is beneficial for the speed of the allocation because accessing the LIR, especially iterating over instructions or operands, was identified as the most time-consuming part. To get the next use of an interval, we must access only the next element in the list without scanning multiple instructions. Changes in the LIR, e.g. the addition of new instructions, do not require any change in the register allocator. Additionally, use positions avoid platform-dependent code in the allocator.

2.4 Fixed Intervals

Some machine instructions require their operands in fixed registers. Such constraints are already considered during the construction of the LIR by emitting physical register operands instead of virtual register operands. Although the register allocator must leave these operands unchanged, they must be considered during register allocation because they limit the number of available registers. Information about physical registers is collected in *fixed intervals*.

For each physical register, one fixed interval stores where the register is not available for allocation. Use positions are not needed for fixed intervals because they are never split and spilled. Register constraints at method calls are also modeled using fixed intervals: Because all registers are destroyed when the call is executed, ranges of length 1 are added to all fixed intervals at the call site. Therefore, the allocation pass cannot assign a register to any interval there, and all intervals are spilled before the call.

2.5 Example

Figure 1 illustrates the lifetime intervals for an instruction sequence. The four virtual registers `v1` to `v4` are represented by the intervals `i1` to `i4`. Use positions are denoted by black bars. The fixed intervals of all registers but `ecx` are collapsed to a single line because they are equal.

The interval `i1` has a lifetime hole from 10 to 24, which allows the allocator to assign the same register to `i1` and `i2`. The shift instruction requires the second operand in the fixed register `ecx`, so a range from 20 to 22 is added to the fixed interval for `ecx`. All registers are destroyed at the call instruction 16, so a short range of length 1 is added to all fixed intervals.

To allow ranges that do not extend to the next instruction, the instruction numbers are always incremented by 2, i.e. only even numbers are used. Therefore, the ranges from 16 to 17 do not interfere with the allocation of `i4` starting at 18. Additionally, the numbering by two allows the allocator to insert spill loads and stores at odd positions between two instructions.

```
10:  v2 = v1
12:  v3 = 10
14:  v3 = v3 + v2
16:  call foo()
18:  v4 = v3
20:  ecx = v2
22:  v4 = v4 << ecx
24:  v1 = v4
```

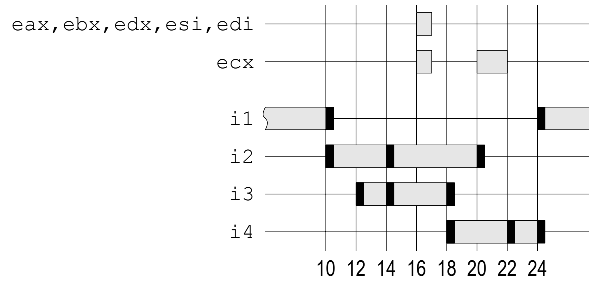


Figure 1: Instruction sequence with lifetime intervals and use positions.

3. LINEAR SCAN ALGORITHM

The main allocation loop processes all lifetime intervals in the order of increasing start positions. The interval currently processed is called the *current* interval. The start position of this interval, called *position*, divides the remaining intervals into the following four sets:

- *Unhandled* intervals start after *position*.
- *Active* intervals cover *position* and have a physical register assigned.
- *Inactive* intervals start before and end after *position*, but do not cover it because of a lifetime hole.
- *Handled* intervals end before *position* or are spilled to memory.

The algorithm LinearScan (see Figure 2) illustrates how the sets are adjusted before a register is selected for *current*: Intervals from *active* that do not cover *position* are either moved to *handled* if they end before *position* or moved to *inactive* otherwise. Similarly, intervals from *inactive* are moved to *handled* or *active*.

The selection of a register for *current* operates in two stages: First, the algorithm TryAllocateFreeReg (see Figure 4) tries to find a free register without spilling an interval. In the best case, a register is free for the entire lifetime, but it is also sufficient if a free register is found only for the beginning of the interval. If no free register is found, AllocateBlockedReg (see Figure 5) tries to make a register free by spilling one or several intervals. The spilling decision is based on the use positions; the interval not used for the longest time is spilled. The next sections present the details of these algorithms.

3.1 Allocation without spilling

The algorithm TryAllocateFreeReg (see Figure 4) is used for allocation without spilling. All *active* intervals and all *inactive* intervals that intersect with *current* possibly affect the allocation decision. They are used to compute the *freeUntilPos* for each physical register. Each register is available for allocation until its *freeUntilPos*.

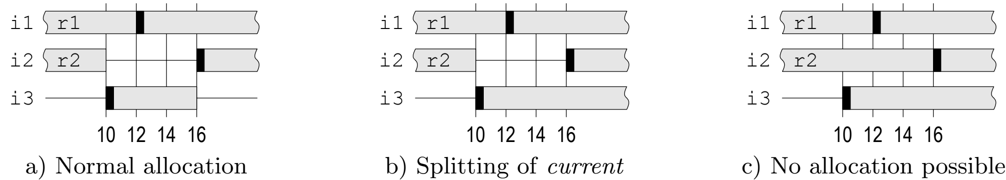
If a register is already assigned to an *active* interval, it must be excluded from the allocation decision. This is represented by a *freeUntilPos* of 0. For *inactive* intervals, the *freeUntilPos* is set to the next intersection with *current* because the register is not available after this position. If the *freeUntilPos* for one register is set multiple times, the minimum of all positions is used. The register with the highest *freeUntilPos* is searched and used for allocation.

```
LinearScan
  unhandled = list of intervals sorted by increasing start positions
  active = { }; inactive = { }; handled = { }

  while unhandled ≠ { } do
    current = pick and remove first interval from unhandled
    position = start position of current

    // check for intervals in active that are handled or inactive
    for each interval it in active do
      if it ends before position then
        move it from active to handled
      else if it does not cover position then
        move it from active to inactive

    // check for intervals in inactive that are handled or active
    for each interval it in inactive do
      if it ends before position then
        move it from inactive to handled
      else if it covers position then
        move it from inactive to active

    // find a register for current
    TryAllocateFreeReg
    if allocation failed then AllocateBlockedReg

    if current has a register assigned then add current to active
```

Figure 2: Basic linear scan algorithm.

Three cases must be distinguished, as illustrated in Figure 3. In all three examples, the intervals `i1` and `i2` already occupy the physical registers `r1` and `r2`. No other registers are available for allocation. Interval `i3` is the *current* interval for allocation.

- In Figure 3 a), the intervals `i2` and `i3` do not intersect, so `r2` is available for the whole lifetime of `i3`. This is represented by a high *freeUntilPos* (the maximum integer number is used). The interval `i3` gets the register `r2` assigned and the allocation completes successfully; no interval must be split.
- In Figure 3 b), the *freeUntilPos* of `r2` is set to 16 because `i2` and `i3` intersect. The register `r2` is available only for a part of `i3`, so `i3` is split at position 16. The split child, a new interval starting at position 16, is added to the unhandled set and will be processed later. The allocation completes successfully.
- In Figure 3 c), both registers are blocked by *active* intervals, so their *freeUntilPos* is 0. It is not possible to allocate a register without spilling. The first stage of allocation fails.



a) Normal allocation b) Splitting of *current* c) No allocation possible

Figure 3: Examples of allocation without spilling.

```
TryAllocateFreeReg
  set freeUntilPos of all physical registers to maxInt

  for each interval it in active do
    freeUntilPos[it.reg] = 0
  for each interval it in inactive intersecting with current do
    freeUntilPos[it.reg] = next intersection of it with current

  reg = register with highest freeUntilPos
  if freeUntilPos[reg] = 0 then
    // no register available without spilling
    allocation failed
  else if current ends before freeUntilPos[reg] then
    // register available for the whole interval
    current.reg = reg
  else
    // register available for the first part of the interval
    current.reg = reg
    split current before freeUntilPos[reg]
```

Figure 4: Allocation without spilling.

Assigning a register only for the first part of an interval is an important optimization, especially for architectures with few registers. Long intervals can switch between different registers, so the probability for spilling is lower. In addition to that, interval splitting is necessary to handle method calls when compiling for the Intel IA32 architecture: Since all registers are considered as blocked at a method call, an interval spanning a call site can never get a register for its entire lifetime. Such intervals are split automatically before the call. They start in a register, then they are split and spilled at the call site and later reloaded to a register if necessary.

If more than one register is available for the whole lifetime of an interval, the selection is based on how these registers are used after the end of the interval: The register that is blocked first by a fixed interval is selected. This optimizes the use of callee-saved registers available on architectures like SPARC: Because caller-saved registers are blocked at each method call, they are preferred when registers of both kinds are available. The callee-saved registers are kept free for intervals spanning method calls.

3.2 Spilling of Intervals

When more intervals are simultaneously live than physical registers are available, spilling is inevitable. Finding the optimal interval for spilling, i.e. the one with the smallest negative impact on the overall performance, would be too time-consuming. Instead, we apply a heuristic that is based on the use positions of the *active* and *inactive* intervals: The interval that is not used for the longest time is spilled. This interval is selected by the algorithm AllocateBlockedReg (see Figure 5).

The *nextUsePos* for each physical register is computed from the use positions of the *active* and *inactive* intervals.

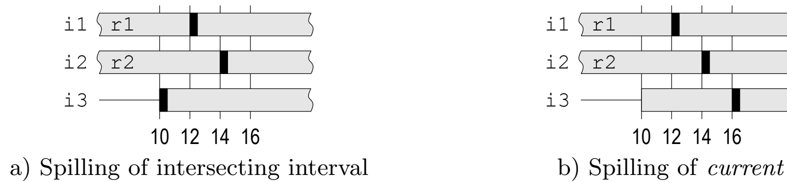
```
AllocateBlockedReg
  set nextUsePos of all physical registers to maxInt

  for each interval it in active do
    nextUsePos[it.reg] = next use of it after start of current
  for each interval it in inactive intersecting with current do
    nextUsePos[it.reg] = next use of it after start of current

  reg = register with highest nextUsePos
  if first usage of current is after nextUsePos[reg] then
    // all other intervals are used before current,
    // so it is best to spill current itself
    assign spill slot to current
    split current before its first use position that requires a register
  else
    // spill intervals that currently block reg
    current.reg = reg
    split active interval for reg at position
    split any inactive interval for reg at the end of its lifetime hole

  // make sure that current does not intersect with
  // the fixed interval for reg
  if current intersects with the fixed interval for reg then
    split current before this intersection
```

Figure 5: Allocation with spilling.

If the *nextUsePos* for one register is set multiple times, the minimum of all positions is calculated. This number denotes when the register is used for the next time. The register with the highest *nextUsePos* is selected because this frees a register for the longest possible time. If the first use position of *current* is found after the highest *nextUsePos*, it is better to spill *current* itself. It is split before its first use position because there it must be reloaded. All other intervals are not changed and remain in their registers.

Figure 6 shows two examples with slightly different intervals. The ranges are equal, only the use position of interval `i3` is different. Again, only two physical registers `r1` and `r2` are available for allocation. The intervals `i1` and `i2` have already the registers `r1` and `r2` assigned. Interval `i3` is the *current* interval for allocation. Normally, each interval starts with a use position as in Figure 6 a), but situations like Figure 6 b) can occur for split children.

The calculated *nextUsePos* of `r1` is 12, the *nextUsePos* of `r2` is 14 in both examples. Therefore, `r2` is the best candidate for allocation. In Figure 6 a), the interval `i3` is used at position 10, before the *nextUsePos* of `r2`. So the interval `i2` is split at position 10 to free the register `r2`, which is then assigned to `i3`. In Figure 6 b), the interval `i3` is used at 16. This position is higher than the *nextUsePos* of `r2`, so the *current* interval `i3` is spilled itself. Because the use position at 16 requires a register, `i3` is split before this position. The split child with the use position at 16 will get a register later when it is processed.



a) Spilling of intersecting interval

b) Spilling of *current*

Figure 6: Examples of allocation with spilling.

Even with spilling, it is possible that no register is available for the whole lifetime of *current*: When all registers are blocked by fixed intervals, e.g. because of a method call, *current* must be split before the intersection with the fixed interval. It is assigned a register for the first part of its lifetime as described in the previous section.

If an interval is used in a loop, it should not be spilled because this requires a load and a store in each loop iteration. The spilling heuristic only considers future use positions, but does not look backward to previous uses. To prevent the spilling of intervals that are used in a loop, pseudo use positions are inserted for them at the end of the loop just before the backward branch. So the heuristic prefers to spill intervals that are used after the loop instead of intervals that are used in the loop.

3.3 Resolution

The linear scan approach to register allocation flattens the control flow graph (CFG) to a linear list of blocks before allocation. When an interval is split within a basic block, a move instruction is inserted at the split position. Additionally, the splitting can create conflicts at control flow edges: When an interval is in a register at the end of the predecessor, but is expected on the stack at the start of the successor (or vice versa), a move instruction must be inserted to resolve the conflict. This can happen if an interval flows into a block via two branches and is split in one of them. In this case, the incoming interval resides in different locations and a move instruction must be inserted.

For resolution, all control flow edges are iterated in a separate pass. If the locations of intervals at the end of the predecessor and the start of the successor differ, appropriate move instructions are inserted. Care must be taken to order the moves if the same register is used as input and output of moves. Our resolution algorithm is similar to the one in [17].

3.4 Exception Handling

Many Java bytecodes can throw runtime exceptions, e.g. if null is dereferenced, if an array index is out of bounds or a number is divided by 0. Therefore, exception handling must be designed carefully so that it does not slow down the normal execution where no exceptions are thrown. In particular, splitting a basic block after each potential exception-throwing instruction and drawing normal control flow edges to the exception handler would fragment the non-exceptional control flow and lead to a large number of short basic blocks.

Instead, we maintain a list containing all instructions that may branch to an exception handler. The SSA-based front end creates phi functions at the start of the exception handler if a local variable has different values at different throwing instructions. For a naive resolution of the phi functions, an adapter block that stores the resulting moves would have to be created for each edge from an instruction to an exception handler.

To reduce the number of adapter blocks, we create them lazily after register allocation: In contrast to the non-exceptional control flow, the phi functions of exception handlers are still present in the LIR. During resolution, the phi functions are resolved and adapter blocks are created on demand. This saves about 75% of all adapter blocks that would be required if the phi functions were resolved before register allocation.

We use an extended version of the resolution algorithm to create moves for exception edges: For phi functions, the correct input operand must be searched first. Then a move instruction from the operand's interval to the phi function's interval is created if the locations are different. Only if moves are necessary at an exception edge, a new adapter block is created for them.

4. OPTIMIZATIONS

4.1 Optimal Split Positions

Large and even medium-sized methods contain many intervals that must be split and spilled. The negative impact of spilled intervals can be reduced by searching the optimal positions for splitting, i.e. the split positions that minimize the number of spill loads and stores executed at runtime. In general, the position where an interval is spilled or reloaded can be moved to a lower (i.e. earlier) position, while the upper bound is specified by the split position calculated by the algorithms. The following rules are used to find an optimal split position.

Move split positions out of loops: Loop blocks are executed more often than blocks of a sequential control flow. Accordingly, spilling or reloading inside a loop leads to a higher number of memory accesses than spilling before or after a loop. In the example in Figure 7, block `B2` is contained in a loop, while `B1` is before the loop. Assume that `i1` must be spilled before 22 because `i2` requires its register. When the split position is moved from 22 to 18, `i1` is spilled before the loop, eliminating the spilling and reloading in each loop iteration.

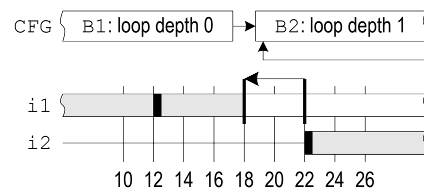


Figure 7: Move split positions out of loops

Move split positions to block boundaries: When an interval is split, a move instruction from the old to the new location is inserted. If the split position is moved to a block boundary, the move might be unnecessary because of the control flow. In the example in Figure 8, `i1` must be spilled at 16 because its register is required by `i2`, but reloaded at 24. When the split position for reloading is moved from 24 to 20, then the spilled part affects only the block `B2`. No resolving move and no spilling is needed at the edge from `B1` to `B3`.

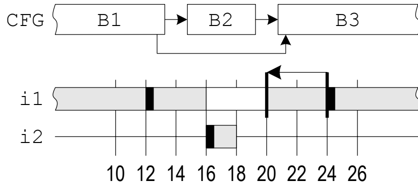


Figure 8: Move split positions to block boundaries

4.2 Register Hints

The most frequently occurring instructions are moves from one virtual register to another. If the intervals for these registers do not overlap, it would be possible to coalesce them to one larger interval. However, coalescing can also decrease code quality because longer intervals tend to require more spilling; many heuristics were proposed for graph-coloring register allocators [11]. Coalescing is an expensive operation because it modifies the intermediate representation after internal data structures of the register allocator have been built, thus leading to an iterative recreation of the data structures until no more moves are coalesced.

Instead of coalescing intervals, we use a dynamic solution called *register hints* that does not require additional compilation time: When two intervals are connected only by a move instruction, the interval for the move target stores the source of the move as its register hint. If possible, the target then gets the same register assigned as the source, even if this register is not optimal according to the criteria described above. In this case, the move instruction is unnecessary and therefore deleted later on. A comparable approach for graph coloring algorithms is called *biased coloring* [1].

4.3 Spill Store Elimination

Moves inserted for spilling can be either stores from a register to the stack or loads from the stack to a register. Loads are always necessary, but stores can be eliminated in certain cases: When the stack slot is known to be up-to-date, i.e. when it can be proven statically that the stack slot already contains the same value as the register, the store can be deleted. Normally, this is difficult to prove and requires a data flow analysis, as implemented e.g. by O. Traub et al. [17]. We identified two common cases that can be optimized without a data flow analysis, but nevertheless cover 95% of all intervals:

- Method parameters are passed on the stack and loaded before the first use. When such an interval is spilled later, no store is necessary as long as the parameter did not change. Especially spill stores for the frequently used this-pointer of a method—passed as the first parameter—are removed.
- Most intervals have only one instruction that defines the value, but are used multiple times later on. If such an interval is spilled and reloaded several times, we insert a spill move directly after the definition. Therefore, the stack slot is up-to-date in all possible code paths, and all further stores to this stack slot can be eliminated.

4.4 Evaluation

To measure the impact of the three optimizations, we executed the SPECjvm98 benchmark suite [15] without optimizations, with only one optimization activated and with all three optimizations activated together. Table 1 shows the number of register-to-register moves, spill loads and spill stores executed. Interval splitting and lifetime holes are enabled in all configurations because they are required for a correct register allocation, e.g. for the handling of method calls.

The results show that each optimization is effective for a particular kind of moves: Splitting at the optimal position reduces the number of spill loads and stores. This optimization is especially effective for methods with small nested loops. For the benchmark _222_mpegaudio, the number of loads and stores is reduced by 25%. In average, the number is reduced by 15%.

Register hints reduce the number of register-to-register moves by 25%, with a maximum of 50% for the benchmark _227_mtrt. The measurements prove that selecting non-optimal registers does not have negative effects on later spilling decisions because the number of spill loads and stores does not increase.

Spill store elimination removes 55% of all spill stores, with a maximum of 70% for _227_mtrt. This optimization overlaps with the optimal split positions; it is more effective for spill stores, but does not optimize spill loads.

The last column of Table 1 shows the speedup when compared to the run without optimizations activated. An overall speedup of 4% is achieved. This speedup comes at no cost of compilation time; there is no measurable change when the optimizations are disabled or enabled.

	million moves executed			**speedup**
	reg-to-reg	spill loads	spill stores	
no optimization	2,553	2,444	1,882	
optimal split pos.	2,672	**2,114**	**1,585**	1.01
register hints	**1,865**	2,439	1,875	1.01
spill store elimin.	2,554	2,444	**834**	1.03
all optimizations	**1,870**	**2,088**	**820**	1.04

Table 1: Move optimizations for SPECjvm98

	million moves executed			**speedup**
	reg-to-reg	spill loads	spill stores	
no optimization	6,493	6,423	6,390	
optimal split pos.	6,557	**1,955**	**1,922**	1.08
register hints	**4,960**	6,390	6,356	1.01
spill store elimin.	6,493	6,423	**1,780**	1.04
all optimizations	**4,960**	**1,955**	**706**	1.10

Table 2: Move optimizations for SciMark 2.0

The optimizations are even more effective for methods with mathematical computations because they usually contain small nested loops. Table 2 shows the results for SciMark 2.0 [14], a benchmark suite for scientific and numerical computing. Moving the split positions out of loops eliminates 70% of all spill loads and stores and leads to a speedup of 8%. With all three optimizations enabled, 90% of all spill stores, 70% of all spill loads and 25% of register-to-register moves are eliminated, leading to a speedup of 10%.

5. FLOATING POINT VALUES

The Intel IA-32 architecture has a floating point unit (FPU, [6]) for floating point computations. The historic design of the FPU complicates the compiler's work. In particular, the following two issues must be considered:

- The internal data format of the FPU registers is not compliant with the IEEE 754 standard for floating point arithmetic [5] required by the Java specification. The FPU has a higher precision than specified, so explicit rounding is necessary. Unfortunately, the only way to round values is to store them to memory and then reload them into a register. This undermines the primary goal of register allocation and prohibits the generation of effective floating-point code.
- The FPU register set is organized as a stack. It is not possible to address registers by their number, but only by their offset from the current stack top. This requires an additional phase in the register allocator that converts register numbers to stack indices using a FPU stack simulation.

The SSE2 extensions introduced with the modern Pentium 4 processors offer not only single-instruction multiple-data (SIMD) instructions for floating point values, but also scalar instructions that can completely replace the old FPU. When the compiler detects that the SSE2 extensions are present, the generated code does not use the FPU anymore. The SSE2 instructions adhere to the IEEE standard and allow a direct addressing of registers, so they are easier to handle in the compiler and lead to code that executes faster, as the benchmarks in the next section show.

6. EVALUATION

The original design goal of the Java HotSpot™ client compiler was to provide a high compilation speed at the possible cost of peak performance [4]. This was obtained by omitting time-consuming optimizations. Our main goal was the implementation of a global register allocation algorithm that leads to faster executing code without a significant compile time increase. The measurements of this section prove that this goal was achieved. Additionally, the measurements show the large difference between code using the FPU and the SSE2 extensions for floating point computations. Whereas numeric applications show a speedup below average when the FPU is used, the speedup is above average when the SSE2 extensions are enabled.

6.1 Compared Configurations

We compared our research version of the Java HotSpot™ client compiler with the product version of the Sun JDK 5.0, using the following four configurations:

- *JDK 5.0 Client:* The Java HotSpot™ client compiler of the Sun JDK 5.0. It does not use the SSE2 extensions for floating point operations.
- *Linear Scan FPU:* Our research version of the client compiler with linear scan register allocation. The FPU is used for floating point operations.
- *Linear Scan SSE2:* Our research version of the client compiler, using the SSE2 extensions for floating point operations. The compilation time is equal for both variants of linear scan, so only one number is given.
- *JDK 5.0 Server:* The Java HotSpot™ server compiler [10] of the Sun JDK 5.0, using the SSE2 extensions.

The client compiler of the JDK 5.0 uses a simple heuristic for register allocation: In the first pass, only temporary values get a register assigned. If a register remains completely unused in the whole method or in a loop, this register is assigned to the most frequently accessed local variable. Load and store elimination are used to optimize multiple accesses to the same local variable in one basic block.

The server compiler produces faster executing code than the client compiler, but at the cost of a higher compilation time. It uses a graph-coloring register allocator for global register allocation. First, the live ranges are gathered and conservatively coalesced; then the nodes are colored. If the coloring fails, spill code is inserted and the algorithm is repeated. About half of the compilation time is spent in the register allocator. The server compiler requires much more time for the register allocation of a method than the client compiler for the whole compilation.

For all measurements, the runtime library of the JDK 5.0 was used. All benchmarks were performed on an Intel Pentium 4 processor 540 with 3.2 GHz, 1 MByte L2-Cache and 1 GByte of main memory, running Microsoft Windows XP Professional. The Pentium 4 processor implements the SSE2 extensions, so direct comparisons between FPU and SSE2 code are possible. The JVM was started with the default options; no command line flags were used.

6.2 Compile Time

We used the SPECjvm98 benchmark suite [15] to get a reasonable set of methods compiled. Table 3 summarizes some statistical data about the compilation. The numbers are accumulated over all compiled methods. All of these numbers use the physical size of the methods in bytes, not the number of bytecodes (typical bytecodes have a size of one to three bytes).

The row *Compiled methods* shows the number of times the compiler is invoked. *Compiled bytes* accumulates the size of all methods compiled, i.e. the number of bytes parsed by the compiler. Because of different inlining strategies, the results for the client compiler and linear scan vary slightly. The server compiler uses a much more aggressive inlining strategy, so the number of compiled bytes is higher. Because the compiler is never invoked on small methods that can be inlined, the number of compiled methods is lower.

Code size is the size of the machine code generated by the compiler. *Total size* is the memory size allocated to store the compiled code together with metadata required by the virtual machine. The slightly higher memory footprint of linear scan does not have a negative impact on the performance of the JVM.

	JDK 5.0 Client	Linear Scan	JDK 5.0 Server
Compiled methods	1,250 methods	1,229 methods	741 methods
Compiled bytes	236,016 bytes	253,618 bytes	367,234 bytes
Code size	919,374 bytes	1,159,979 bytes	1,630,713 bytes
Total size	2,367,575 bytes	2,934,195 bytes	3,881,072 bytes
Compilation time	1,056 msec.	1,070 msec.	27,946 msec.
Compilation speed	**223,512 bytes/sec.**	**237,074 bytes/sec.**	**13,140 bytes/sec.**

Table 3: Comparison of compile time

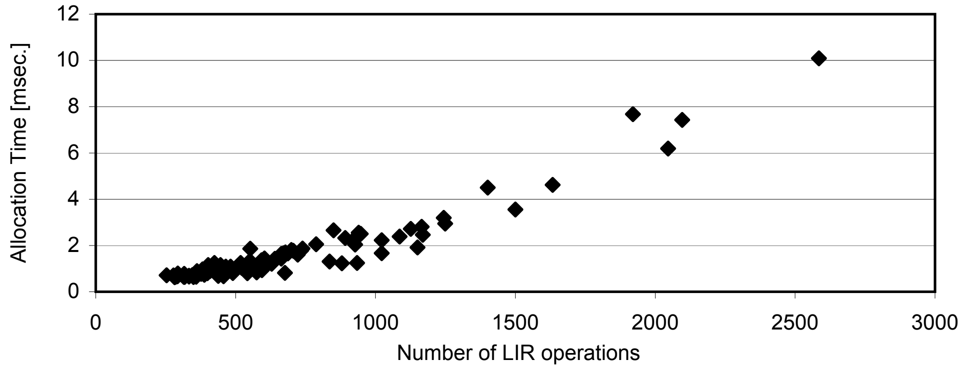


Figure 9: Time for register allocation—100 slowest methods out of 1229.

Compilation time is the total time spent in the compiler. The lower the compilation time is, the less time is spent in the compiler and the shorter are the pauses when a method is compiled. The most important number is the *compilation speed*, calculated as the quotient of *compiled bytes* and *compilation time*.

The compilation time of linear scan and the client compiler are nearly equal; the compilation speed of linear scan is even higher. This states that the linear scan algorithm does not have a negative impact on the compilation speed, as opposed to typical implementations of the graph-coloring algorithm. The compilation speed of the server compiler cannot compete with the client compiler because many time-consuming optimizations are performed.

The time needed for linear scan register allocation mainly depends on the size of the methods. Most methods are very small and result in less than 200 LIR instructions. Only about 8% of all methods compiled during SPECjvm98 are larger. Figure 9 shows the 100 methods with the highest allocation time of all 1,229 methods compiled. The measured time includes the data flow analysis before building the lifetime intervals, the building of the intervals, the linear pass over the intervals for allocation, the resolving of the data flow after the allocation and the rewriting of the LIR with the physical registers. Although the time includes parts that are known to be non-linear, such as the data flow analysis, Figure 9 indicates that the overall algorithm nearly has a linear time behavior.

6.3 Run Time

The SPECjvm98 benchmark suite [15] is commonly used to assess the performance of Java runtime environments. It consists of seven benchmarks derived from real-world applications that cover a broad range of scenarios where Java applications are deployed.

It measures the overall performance of a JVM including class loading, garbage collection and loading input data from files. The programs are executed several times until no significant change in the execution time occurs. The slowest and the fastest runs are reported.

The slowest runs indicate the startup speed of the JVM including the time needed for compilation; the fastest runs measure the peak performance and therefore the quality of the generated code. Table 4 shows the detailed results of SPECjvm98[1] for our four configurations. Table 5 illustrates the speedup when the client compiler is compared with the other three configurations.

Linear scan outperforms the client compiler of the JDK 5.0 by 25%, i.e. the peak performance, measured as the mean of the fastest run, is 25% higher. The gap to the server compiler is reduced significantly; the server compiler is only 6% faster than linear scan, while it is 32% faster than the client compiler.

The increase of peak performance comes at no cost of startup time. Even the slowest run of linear scan, which includes all compilations, is 22% faster than the slowest run of the client compiler. The slowest run of the server compiler is not competitive: Because of the low compilation speed, the server compiler is 15% slower than the client compiler and even 40% slower than linear scan.

Because of the complicated structure of the Intel FPU, the two floating point benchmarks _227_mtrt and _222_mpegaudio do not benefit much from linear scan register allocation as long as the FPU is used. The explicit rounding that requires a memory access after most computations does not allow an effective register allocation. However, when the SSE2 extensions are enabled, the speedup is above average.

[1]The results are not approved SPECjvm98 metrics, but adhere to the run rules for research use. The input size 100 was used for all measurements.

	JDK 5.0 Client		Linear Scan FPU		Linear Scan SSE2		JDK 5.0 Server	
	slowest	fastest	slowest	fastest	slowest	fastest	slowest	fastest
_227_mtrt	1.80	1.53	1.67	1.41	1.34	1.09	1.92	1.06
_202_jess	2.13	1.89	1.88	1.61	1.86	1.61	2.50	1.56
_201_compress	7.16	7.11	6.02	5.95	6.03	6.00	5.69	5.55
_209_db	11.89	11.84	11.84	11.69	11.78	11.63	11.64	11.17
_222_mpegaudio	5.50	5.23	5.23	4.97	3.02	2.78	3.45	2.30
_228_jack	3.42	3.09	3.23	2.91	3.20	2.88	8.52	2.83
_213_javac	5.89	4.73	5.34	4.23	5.25	4.09	10.59	4.06

Table 4: Benchmark results for SPECjvm98 (runtime in sec.)

	Linear Scan FPU		Linear Scan SSE2		JDK 5.0 Server	
	slowest	fastest	slowest	fastest	slowest	fastest
_227_mtrt	1.07	1.09	1.34	1.40	0.93	1.44
_202_jess	1.13	1.18	1.14	1.17	0.85	1.21
_201_compress	1.19	1.19	1.19	1.18	1.26	1.28
_209_db	1.00	1.01	1.01	1.02	1.02	1.06
_222_mpegaudio	1.05	1.05	1.82	1.88	1.59	2.28
_228_jack	1.06	1.06	1.07	1.08	0.40	1.09
_213_javac	1.10	1.12	1.12	1.16	0.56	1.17
Geometric Mean	1.09	**1.10**	1.22	**1.25**	0.87	**1.32**

Table 5: Speedup of SPECjvm98 relative to JDK 5.0 Client

The use of the SSE2 extensions contributes to the average speedup of 25%. When only the integer benchmarks _202_jess, _201_compress, _228_jack and _213_javac are considered, the speedup is 15% both in FPU and in SSE2 mode. The server compiler is only 3% faster than linear scan for these benchmarks.

_209_db shows the lowest speedup. It performs database functions on a memory-resident address database and spends most time in the sorting algorithm that offers no possibility for optimizations by register allocation. All values are loaded from arrays inside the innermost loop. Even the more elaborate optimizations of the server compiler cannot reduce the execution time significantly.

The most frequently executed loops of _201_compress and _222_mpegaudio access data stored in arrays. Linear scan succeeds to assign registers to the local variables used in the loops; no unnecessary spill moves are inserted. Only the array loads are still present in the loops, so the optimizations of the server compiler like range check elimination are responsible for the better performance of the code generated by the server compiler.

7. RELATED WORK

The linear scan algorithm was described first by M. Poletto et al. [12][13]. Their version is very fast because the complete allocation is done in one linear pass over the lifetime intervals. However, it cannot deal with lifetime holes and does not split intervals, so an interval has either a register assigned for the whole lifetime, or it is spilled completely. This has a negative effect on architectures with few registers such as the Intel IA-32 architecture because it increases the register pressure. In particular, it is not possible to implement the algorithm without reserving a scratch register: When a spilled interval is used by an instruction requiring the operand in a register, the interval must be temporarily reloaded to the scratch register. Additionally, register constraints for method calls and instructions requiring fixed registers must be handled separately. Our algorithm does not require a scratch register because all constraints of the target architecture, including the calling conventions for method calls, are contained in fixed intervals and use positions.

When an interval must be spilled, their algorithm spills the interval that ends last. This heuristic is good when each interval consists of exactly one definition and one use, but it penalizes long intervals that are frequently used, such as the this-pointer of a method. We estimate the priority of an interval with its distance to the next use position and spill the interval that is not used for the longest time.

The *second chance binpacking* algorithm by O. Traub et al. [17] introduced lifetime holes and interval splitting to the linear scan algorithm. Our algorithm is influenced by Traub's ideas, but has considerable differences:

- While Traub rewrites the code during allocation, we separate the allocation from the rewriting of the LIR. This allows a more flexible splitting of intervals because we can move the split positions of intervals to an earlier position in the code, e.g. out of a loop.
- Our main allocation loop is independent from the intermediate representation. All information required during allocation is contained in the lifetime intervals and use positions.
- The use positions distinguish between uses that *must* have a register and use positions that *should* have a register assigned. So it is possible to use the addressing modes of the IA-32 architecture that allow one memory operand for arithmetic instructions.

- Traub uses an additional data flow analysis when stores are inserted during resolution to guarantee the consistency of registers and their spill slots. Our *spill store elimination* avoids the expensive data flow analysis, but nevertheless covers 95% of all intervals.

An early version of our linear scan register allocator was described in [9]. Compared with this old version, substantial parts have changed:

- The old version operated on the high-level intermediate representation (HIR) using SSA form [3]. In the new implementation, the HIR is converted to the LIR before register allocation.
- Because the HIR did not expose all registers necessary for an instruction, one register was reserved as a scratch register.
- The expensive coalescing of intervals was replaced by the lightweight *register hints* to save compilation time.
- The old version did not split intervals. Splitting of intervals was introduced because it reduces spilling and simplifies the handling of register constraints.

8. CONCLUSIONS

We presented the details of a register allocator using the linear scan algorithm for the Java HotSpot™ client compiler of Sun Microsystems. The algorithm for selecting which register is assigned to an interval is based on elaborate heuristics: With the help of use positions, it is possible to spill as few intervals as possible. When spilling is inevitable, we select the interval that is not used for the longest time because a premature reloading could require another interval to be spilled.

Global register allocation reduces the constraints for other global optimizations in the compiler. While the old register allocator restricted the lifetime of temporaries to a single basic block, they are now handled in the same way as local variables. More sophisticated optimizations such as escape analysis [7] can be implemented now. The linear scan algorithm has meanwhile reached a very stable state: Our compiler passes all internal test cases of Sun Microsystems. There are plans to integrate it into a future version of the JDK.

9. ACKNOWLEDGMENTS

We would like to thank the Java HotSpot™ compiler team at Sun Microsystems, especially Thomas Rodriguez, Kenneth Russell and David Cox, for their persistent support, for contributing many ideas and for helpful comments on all parts of the Java HotSpot™ Virtual Machine.

10. REFERENCES

[1] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, 1994.

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

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[4] R. Griesemer and S. Mitrovic. A compiler for the Java HotSpot™ virtual machine. In L. Böszörményi, J. Gutknecht, and G. Pomberger, editors, *The School of Niklaus Wirth: The Art of Simplicity*, pages 133–152. dpunkt.verlag, 2000.

[5] Institute of Electrical and Electronics Engineers. *IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Standard 754-1985.*

[6] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, 2004. Order Number 253665.

[7] T. Kotzmann and H. Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the First Conference on Virtual Execution Environments*, 2005.

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

[9] H. Mössenböck and M. Pfeiffer. Linear scan register allocation in the context of SSA form and register constraints. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 229–246, 2002.

[10] M. Paleczny, C. Vick, and C. Click. The Java HotSpot™ server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, 2001.

[11] J. Park and S.-M. Moon. Optimistic register coalescing. *ACM Transactions on Programming Languages and Systems*, 26(4):735–765, 2004.

[12] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 109–121. ACM Press, 1997.

[13] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[14] R. Pozo and B. Miller. *SciMark 2.0.* http://math.nist.gov/scimark2/.

[15] Standard Performance Evaluation Corporation. *SPECjvm98.* http://www.spec.org/jvm98/.

[16] Sun Microsystems, Inc. *The Java HotSpot™ Virtual Machine, v1.4.1*, 2002. http://java.sun.com/products/hotspot.

[17] O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 142–151. ACM Press, 1998.

[18] C. Wimmer. Linear scan register allocation for the Java HotSpot™ client compiler. Master's thesis, Johannes Kepler University Linz, 2004.