JOHANNES KEPLER
UNIVERSITY LINZ
Research and teaching network

Christian Wimmer

Linear Scan Register Allocation for the Java HotSpot™ Client Compiler

A thesis submitted in partial satisfaction of
the requirements for the degree of

Master of Science
(Diplom-Ingenieur)

Supervised by

o.Univ.-Prof. Dipl.-Ing. Dr. Hanspeter Mössenböck

Institute for System Software
Johannes Kepler University Linz

Linz, August 2004

Abstract

Register allocation is the task of assigning local variables and temporary values to physical registers of a processor. It is crucial for the efficiency of compiled code. The most commonly used algorithm treats the task of register allocation as a graph coloring problem. It generates code of good quality, but is too slow for just-in-time compilers because of its quadratic runtime complexity. For such compilers, the linear scan algorithm is an efficient alternative: It generates code of nearly the same quality, but is much faster than the graph coloring algorithm because it needs only a single pass over the lifetime intervals.

The *Java HotSpot Virtual Machine* by Sun Microsystems uses a just-in-time compiler to generate native code for frequently executed methods. To achieve a high compilation speed and a low startup time, the HotSpot client compiler avoids time-consuming optimizations. The current product version assigns registers using a local heuristic. In the context of this master thesis, a research version of the compiler was extended with the linear scan algorithm for register allocation. The implemented variant improves the basic algorithm with more advanced optimizations: It makes use of lifetime holes, splits intervals if necessary and models register constraints of the target architecture with fixed intervals.

Benchmark results prove that the linear scan algorithm is a good tradeoff if both compilation time and runtime of a program matter: The compilation time is only slightly higher in comparison with the old local heuristic for register allocation, but the resulting code executes about 30% faster. The benchmarks also indicate the high impact of the Intel SSE2 extensions on the speed of numeric Java applications.

Kurzfassung

Eine der wichtigsten Compileroptimierungen ist die Registerallokation, die lokale Variablen und temporäre Werte auf die Register des Prozessors abbildet. Das am häufigsten verwendete Verfahren basiert auf Graphfärbung. Es erzeugt hochqualitativen Code, ist aber wegen seiner quadratischen Laufzeitkomplexität zu langsam für Just-in-Time-Compiler. Für solche Anwendungen ist das Linear-Scan-Verfahren eine effiziente Alternative. Es erzeugt zwar nicht ganz so guten Code, ist aber in der Laufzeit im Wesentlichen linear.

Die *Java HotSpot Virtual Machine* von Sun Microsystems verwendet einen Just-in-Time-Compiler, um Maschinencode für häufig ausgeführte Methoden zu erzeugen. Um eine hohe Übersetzungsgeschwindigkeit zu erreichen, führt der HotSpot Client Compiler dabei keine zeitaufwendigen Optimierungen durch. Die aktuelle Produkt-Version verwendet derzeit eine einfache Heuristik für die Registerallokation. Diese Diplomarbeit beschreibt die Registerallokation nach dem Linear-Scan-Verfahren für eine Forschungs-Version des Compilers. Optimierungen wie die Ausnutzung von Löchern in Live-Intervallen, die Möglichkeit zur Teilung von Intervallen und die Verwendung von vorgefärbten Intervallen führen zu einer Verbesserung der Code-Qualität.

Benchmarks zeigen, dass das Linear-Scan-Verfahren einen guten Kompromiss zwischen Übersetzungszeit und Laufzeit eines Programms darstellt: Die Übersetzungszeit steigt im Vergleich mit dem alten, heuristischen Registerallokations-Verfahren nicht wesentlich an, die Geschwindigkeit des erzeugen Codes ist jedoch um etwa 30% höher. Zusätzlich zeigen die Benchmarks den großen Einfluss der Intel SSE2-Erweiterungen auf die Geschwindigkeit von numerischen Java-Anwendungen.

Table of Contents

Chapter 1

Introduction

This chapter introduces the problem of register allocation in the context of a fast just-in-time compiler. Then, the history of the research collaboration between Sun Microsystems and the Institute for System Software at the Johannes Kepler University Linz is presented. The linear scan algorithm for register allocation is implemented in a research version of the Java HotSpot client compiler of Sun Microsystems.

Two opposing goals influence the design decisions for a just-in-time (JIT) compiler: On the one hand, the compilation time should be low because it is part of the total runtime of the application. On the other hand, the generated code should run as fast as possible, which requires extensive and time-consuming optimizations. One of these optimizations is the register allocation.

The *Java HotSpot Virtual Machine* by Sun Microsystems reduces the compilation time by executing all methods in interpreted mode first. When a method was interpreted several times, it is considered "hot" and scheduled for compilation. Therefore only few but important methods are compiled. The virtual machine comes with two different JIT compilers: the fast client compiler providing a low startup time and a low response time, and the server compiler providing the best possible peak performance.

The client compiler serves as the basis for the work of this master thesis. It was designed as a straightforward and fast compiler that omits all time-consuming optimizations. Regarding register allocation, only the innermost loops are optimized by a simple, yet effective heuristic.

However, global register allocation is known as a profitable optimization that should be utilized also by the client compiler. The standard algorithm for register allocation used in most modern compilers is based on graph coloring. It generates good code, but is too slow for JIT compilers because even the best heuristic implementations have a quadratic runtime complexity.

The linear scan algorithm for register allocation was developed for time-critical compilers. It generates code that is nearly as good as the code generated by a graph coloring register allocator, but is much faster because of its linear runtime complexity. The goal of this master thesis is the implementation of the linear scan algorithm for the Java HotSpot client compiler. The implementation is evaluated by comparisons with the old heuristic for register allocation in the Sun JDK 1.4.2.

1.1 Project History

The first version of the Java HotSpot client compiler was developed by Robert Griesemer and Srdjan Mitrovic [Griesemer00]. This compiler is part of the Sun JDK since version 1.3. Originally, it used only a graph-based high-level intermediate representation (HIR) for optimizations. For the release of the JDK 1.4 in 2002, the compiler was extended with a second low-level intermediate representation (LIR).

The research collaboration between Sun Microsystems and the Institute for System Software (named Institute for Practical Computer Science before 2004) at the Johannes Kepler University Linz started in 2000, when Hanspeter Mössenböck spent a sabbatical at Sun Microsystems. He extended the client compiler to generate the intermediate representation in static single assignment (SSA) form and added a graph coloring register allocator [Mössenböck00].

The research was continued from 2001 to 2003 by Michael Pfeiffer at the University of Linz. One major step was the replacement of the graph coloring register allocator by the linear scan algorithm because the graph coloring algorithm was too slow for the overall fast client compiler [Mössenböck02]. The register allocator first operated on the HIR. When the LIR was added to the compiler, the algorithm was adapted to work on the LIR.

The work for this master thesis started in 2003. The first implementation of the linear scan algorithm worked well, but had certain drawbacks: Since intervals were allocated always as a whole, only whole intervals could be spilled. This required a complicated handling of scratch registers when a spilled interval was required in a register by an instruction. The result of this master thesis is a more flexible version of the linear scan algorithm that can split intervals. This increases the complexity of the algorithm, but makes the later handling of scratch registers unnecessary.

The second important optimization is the generation of code for the Intel SSE2 extensions. Originally, all floating point computations were performed in the processor's floating point unit (FPU), but the complicated structure of the FPU prevents an efficient register allocation. The SSE2 extensions of modern processors allow a much faster execution of floating point operations and are regular enough to be handled with the linear scan algorithm.

Another successful output of the cooperation with Sun Microsystems is a port of the client compiler to Java by Thomas Kotzmann. Originally, the whole virtual machine is written in C++. This is a necessity for the low level functionality of the runtime system, but higher level subsystems like the just-in-time compiler could also be written in Java itself. One of the goals of this project was the comparison of the compilation speed between the C++ and the Java version of the compiler [Kotzmann02].

1.2 Structure of this Master Thesis

Chapter 2 presents the basic principles for register allocation. After a short overview of local methods, the standard algorithm for register allocation used in many modern compilers is presented. This algorithm treats register allocation as a graph coloring problem. It produces high quality code, but is rather slow. Then, the linear scan algorithm

is presented as a good tradeoff between compilation speed and quality of the resulting native code.

Chapter 3 first presents the abstract specification of a Java virtual machine. Afterwards, the HotSpot virtual machine of Sun Microsystems is described in detail. The client compiler of the HotSpot VM serves as the basis for the research compiler that was extended with the linear scan algorithm. Chapter 4 presents all compilation steps of this compiler, together with the intermediate representations used.

Chapter 5 explains the implementation of the linear scan algorithm for register allocation in all details. The algorithm is presented in pseudo-code and illustrated with examples. The special handling necessary for the Intel floating point unit is then explained in Chapter 6. The implementation is evaluated in Chapter 7 using two different benchmarks. Both the compile time and the run time of the generated code are considered.

Chapter 8 summarizes the result of this master thesis and gives a short outlook of planned future work. Appendix A completes the thesis with a larger compilation example, where all used intermediate representations and data structures are visualized.

1.3 Acknowledgements

I want to thank all people that supported the development of this thesis. At first, I want to thank my advisor Hanspeter Mössenböck for offering me this project and the employment at his institute and for his continued guidance and support of the whole project.

Then I want to thank the HotSpot compiler team at Sun Microsystems for their persistent support and sponsorship, especially Kenneth Russel, Thomas Rodriguez and David Cox for contributing many ideas and helpful comments about all parts of the HotSpot virtual machine, for tracking down bugs and for the continuous integration of our research work with the latest product version of the compiler.

I want to acknowledge the staff at the institute, especially Thomas Kotzmann for the numerous discussions on the algorithms, for his valuable hints on the architecture of the compiler and for his comments on this thesis. Working with him on this project is always a pleasure. Additionally, I thank Michael Pfeiffer, whose original implementation of the linear scan algorithm provided a solid basis for my work.

Finally, I am most grateful to my parents, my brother and my sister for their encouragement and support during my whole study and in particular for proofreading this thesis.

Chapter 2

Algorithms for Register Allocation

This chapter presents some widely used algorithms for register allocation. First, a short description of local methods, which are fast but do not generate optimal code, is given. Then, today's standard algorithm, based on graph coloring, is explained and visualized with a short example. After reasoning why graph coloring is too slow for just-in-time compilers, the linear scan algorithm is presented as a good tradeoff between compilation speed and runtime of the compiled program. References to other projects implementing this algorithm emphasize this.

Register allocation, the task of assigning variables and temporary values to physical registers of a processor, is commonly known as one of the most important optimizations for compilers. The main goal is the minimization of the traffic between the main memory and the processor. Memory bandwidth is often a bottleneck of today's computer systems because a modern processor is much faster than its attached main memory. Even with a hierarchy of caches providing a faster access to frequently used areas of the main memory, accessing a register is several times faster than loading a value from memory.

In most processor architectures, registers are a limited resource. For example, the Intel IA-32 architecture [Intel1] described in this thesis offers only eight general-purpose registers of which only six can be used in normal computations. Therefore, only the most frequently accessed values can be kept in registers. The proper register usage is crucial for the overall performance of a program. All values that cannot be kept in a register must be stored on the stack before the register is overwritten with another value. This process is called *spilling*. When the spilled value is used later on, it must be reloaded to a register again.

The importance of register allocation is observable by the large number of algorithms that are available today. A coarse classification distinguishes two kinds of algorithms:

- Local methods limit the view of the algorithm to a small part of the currently compiled method. Especially the innermost loops are identified and optimized.
- Global methods try to optimize whole methods or even groups of methods. They can achieve the best possible result, but are considerably slower.

With sufficient knowledge about the target architecture, it would be possible to compute an optimal register allocation, where the execution time is as low as possible. But it is proven that optimal global and even local register allocation is an NP-complete problem that is not feasible for practical usage [Sethi73] [Farach98]. As a result, every algorithm must find a

tradeoff between compile time during allocation and runtime of the resulting code. The more time is invested for allocation, the faster is the generated code. This tradeoff is especially important for just-in-time (JIT) compilers where the time required for compilation is part of the total runtime of the application. Therefore, many JIT compilers use local methods, although global methods are available that generate better code but require too much compilation time.

2.1 Local Methods

The straightforward method for register allocation is to allocate a register when it is needed for a computation, and free all registers after the statement was processed. This implies that registers are used only for short-living temporary values within a single statement and not for local variables. This allocation can be done on the fly while emitting machine code and is very fast, but leads to repeated accesses of the same values in memory, while some registers remain completely unused.

These unused registers can be utilized to cache frequently used local variables during the whole method or an often executed loop. Especially the identification and special treatment of loops is worthwhile: Most of the execution time of a program is spent inside loops. Even conservative static estimations state that a nested loop of depth d is executed 10^d times, so moving a memory access out of a loop is always beneficial. This simple and fast heuristic leads to surprisingly good results and is therefore still used in compilers.

Loads and stores of local variables can be further optimized: When a register is stored to a local variable and reloaded immediately afterwards to another register, then the load from memory can be replaced by a register move, provided that the original register was not overwritten in between. If it is known that the local variable is never used later on, the store operation can also be eliminated.

2.2 Graph Coloring Algorithm

Global methods for register allocation generate better code than local methods. They have a precise overview of all values that could be stored in registers, which allows the selection of the best ones. The most commonly used algorithm treats the task of register allocation as a graph coloring problem. The first implementation was presented by G.J. Chaitin in [Chaitin81] and [Chaitin82]. An improved design was proposed by P. Briggs in [Briggs89]. This chapter briefly describes a general standard algorithm as presented in [Muchnick97].

2.2.1 Building the Interference Graph

The graph coloring algorithm works on an intermediate representation of the code where all values that can be assigned a register get a unique virtual register number. Each virtual register has a live range that starts at its first definition and ends at its last use, i.e. a virtual register is live when it contains a valid value that must be preserved. A live range needs not be continuous, but can have holes resulting from multiple definitions and uses.

The *N* virtual registers must then be mapped to the available *R* physical registers, where *N* is usually much larger than *R*. The physical registers are treated as "colors" that are used to color an undirected graph—called the register interference graph—whose nodes are virtual registers. Two nodes are connected by an edge if and only if the two corresponding virtual registers must not get the same physical register because they are live at the same time.

The simple example code shown in Figure 2.1 contains five virtual registers `v1`, `v2`, `v3`, `v4` and `v5`. Assume that registers should be allocated for a target architecture with only two physical registers `r1` and `r2` available. The instructions are numbered from (1) to (7). The right side of Figure 2.1 shows resulting live ranges for the virtual registers. The live range of `v1` starts at the definition in instruction (1) and ends at the last use in (7). Because of the second definition in (5), there is a hole between (3) and (5). All other live ranges are continuous from their definition to their last use.

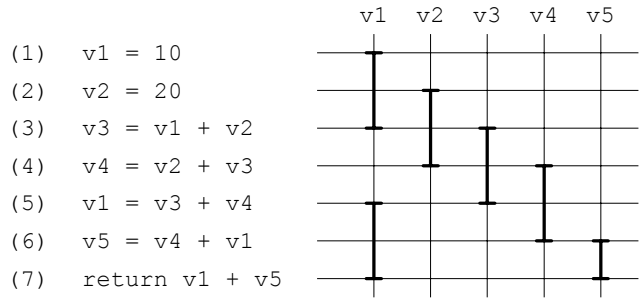


Figure 2.1: Graph coloring example—code with live ranges

It is easy to construct the register interference graph using the live ranges. For example, the nodes `v1` and `v2` are connected by an edge because the virtual register `v2` is defined by instruction (2), between the definition of `v1` in (1) and its use in (3). Also, `v1` interferes with `v4` because of instruction (5), and `v1` interferes with `v5` because of instruction (6). All other interferences can be obtained likewise. The complete interference graph is shown in Figure 2.2.

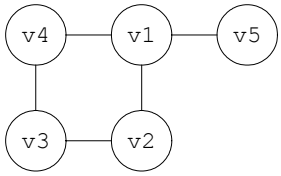


Figure 2.2: Complete register interference graph

2.2.2 Pruning the Graph

The register interference graph must now be colored with *R* colors (called *R*-coloring) so that any two adjacent nodes get different colors, where each color represents a physical register. Finding an *R*-coloring exhaustively is not feasible because the problem is known to be NP-complete. Instead, an iterative approach is used to simplify the graph. If no *R*-coloring can be found, it is not possible to allocate a physical register to each virtual register and some virtual registers must be spilled to memory. A cost function is used to find the values that have the least negative impact on the total performance when they are spilled.

To color the graph, it is iteratively pruned. In each iteration, one node with all its edges is removed from the graph and pushed on a stack. For the selection of this node, two rules described below are applied. When the graph is empty, the removed nodes are popped from the stack and re-added in reverse order. Each added node is assigned a color that is not used by any adjacent node yet. If no color is available, then the node is marked for spilling.

The first rule for removing nodes is called d*egree* < *R* rule: If the graph contains a node with a degree less than *R*, i.e. a node with less than *R* adjacent nodes, then it is *R*-colorable if and only if the graph without this node is *R*-colorable. So it is enough to search for a coloring of the reduced graph. If the reduced graph was successfully colored, there is always a color available for the removed node: Because there are less than *R* adjacent nodes, these nodes cannot occupy all *R* colors and so there must be a free color.

This rule is very effective for reducing a graph, but it is not sufficient. In the previous example, the node `v5` can be removed by the *degree* < *R* rule: It has only one adjacent node, so it is removed from the graph and pushed on the stack. The remaining graph and the current state of the stack are shown in Figure 2.3 a). Now all remaining nodes have a degree of two, so the first rule cannot be applied and the second rule is needed.

The second rule selects the least important node using the cost-function, removes it from the graph and pushes it on the stack. This rule is called *optimistic heuristic*: Even if all nodes have a degree higher than *R*, it might be possible to find an *R*-coloring of the graph. When some adjacent nodes are not connected among themselves, then they can have the same color and an unused color can be found for the current node. But it is also possible that all colors are occupied by adjacent nodes. In this case, the node must be spilled.

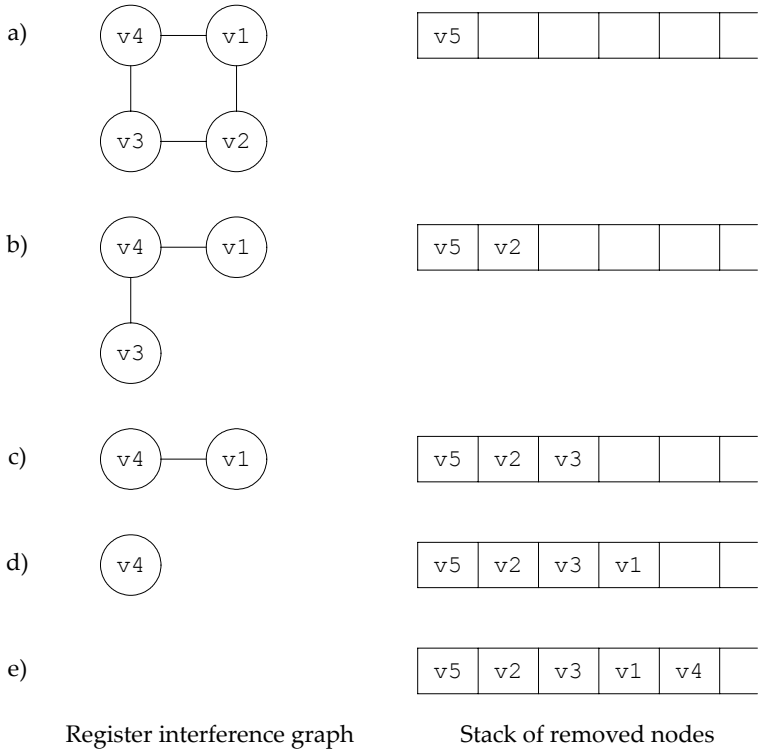


Figure 2.3: Pruning of the register interference graph

In the example, assume that the cost function selected `v2` to be removed from the graph. The resulting graph is shown in Figure 2.3 b). The remaining nodes can be processed by the *degree* < *R* rule because there is always a node with only one edge available now. Figure 2.3 c) - e) shows the graph where the nodes `v3`, `v1` and `v4` are removed. Afterwards, the graph is empty and all nodes are present on the stack.

2.2.3 Reconstruction of the Graph

In the next step of the algorithm, the graph is reconstructed by popping nodes from the stack and restoring the edges to the originally adjacent nodes. The re-added node is assigned a color that is not used yet by any adjacent node. If no such color is available, the corresponding virtual register is marked for spilling. When all nodes were processed and no spilling was necessary, then the graph is completely colored and each virtual register can be replaced with its assigned physical register. If spilling was necessary, the appropriate spill code is inserted into the intermediate representation. Because this code also needs registers, the complete algorithm is repeated, i.e. a new graph is constructed and colored. This is repeated until no further spilling is necessary.

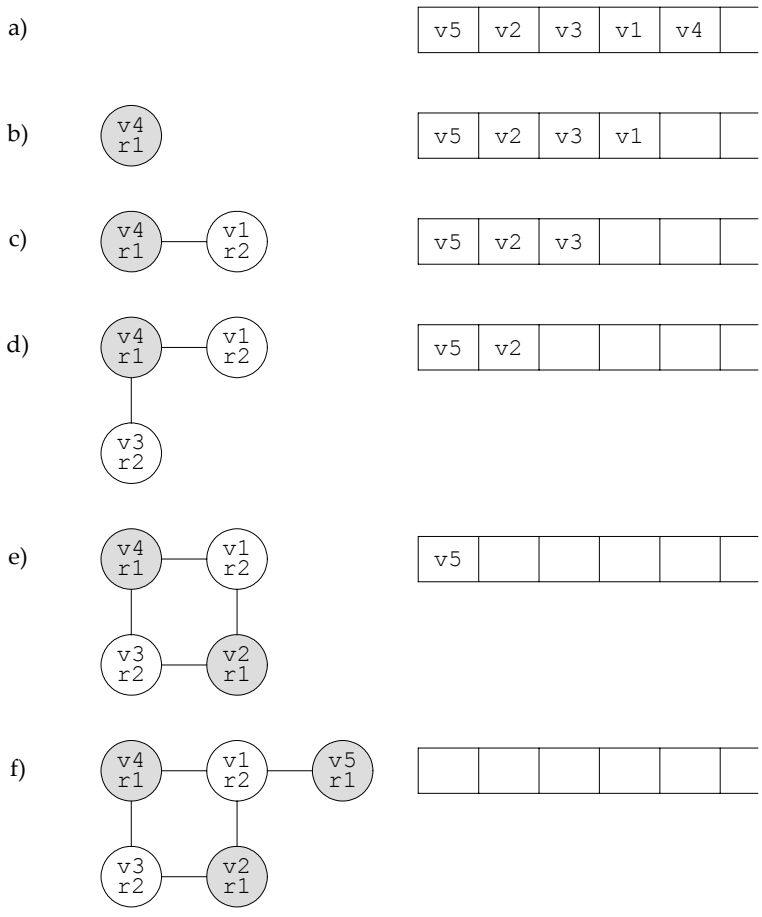


Figure 2.4: Reconstruction of the register interference graph

In the example, the graph is reconstructed in the following order: First `v4` is added and gets the first physical register `r1` assigned (Figure 2.4 b). Then `v1` is added, and because of the edge between `v1` and `v4` the physical register `r2` is assigned (Figure 2.4 c). Similarly `v3` is added and gets the register `r2` (Figure 2.4 d). The node `v2` was removed by the optimistic heuristic rule. Now it is obvious that the optimism was justified because both adjacent nodes `v1` and `v3` have the same physical register `r2` assigned. So `r1` can be assigned to `v2` (Figure 2.4 e). Finally, `v5` is added and gets the register `r1` (Figure 2.4 f).

Now the graph is completely restored, and each node has a physical register assigned. No spilling was necessary, so the algorithm completed successfully. Figure 2.5 shows the resulting code where the virtual registers are replaced by their allocated physical registers.

```
(1)   v1 = 10           r2 = 10
(2)   v2 = 20           r1 = 20
(3)   v3 = v1 + v2      r2 = r2 + r1
(4)   v4 = v2 + v3      r1 = r1 + r2
(5)   v1 = v3 + v4      r2 = r2 + r1
(6)   v5 = v4 + v1      r1 = r1 + r2
(7)   return v1 + v5    return r2 + r1
```

Figure 2.5: Example code before and after register allocation

To reduce the number of nodes that must be colored, most implementations try to coalesce nodes before allocation. When the intermediate representation contains a move from one node to another and the live ranges of the nodes do not overlap, then both nodes can be coalesced to a single node with the union live range. The lower number of nodes increases the compilation speed, but longer live ranges also tend to need more spilling. Exaggerated coalescing can degrade the quality of the resulting code. The decision whether two nodes are coalesced is therefore determined by heuristics.

The graph coloring algorithm is frequently used in state-of-the-art compilers. Several optimizations for compilation speed and code quality were developed, so the time needed for creating and coloring the register interference graph is acceptable for most compilers. But the asymptotic time complexity of the algorithm always remains $O(n^2)$, where n is the number of virtual registers, because each node could be connected with each other. Also, the repetition of the whole algorithm until no more spilling is necessary consumes much time. Therefore, the algorithm is not suitable for compilers where compilation speed is important, such as JIT compilers.

2.3 Linear Scan Algorithm

The linear scan algorithm was described first by M. Poletto et al. in [Poletto97] when they implemented a system for dynamic code generation. The algorithm, described in more detail in [Poletto99], is very fast because the allocation is done in one linear pass over the lifetime intervals. The basic idea of this algorithm is presented in Chapter 2.3.1. An improved version, called *second chance binpacking*, was described by O. Traub et al. in [Traub98]. This algorithm, presented in Chapter 2.3.2, spends more time to get a better

allocation, e.g. it considers holes in lifetime intervals and allows the splitting of lifetime intervals during allocation.

The linear scan algorithm is also implemented in other production quality compilers, especially inside virtual machines for several programming languages: The *Jalapeño Dynamic Optimizing Compiler* is part of the Jalapeño Java virtual machine built at IBM Research. It uses a compile-only approach for executing Java applications, implementing different levels of compilation depending on how frequently methods are executed. The optimizing compiler, presented in [Burke99], uses the linear scan algorithm for register allocation.

Another successful implementation is presented in [Jonasson02]. E. Johansson and K. Sagonas adapted the algorithm for HiPE, their high-performance native code compiler for the concurrent functional programming language Erlang, and compared it with a graph coloring register allocator. They concluded that the linear scan algorithm should be used when compilation time is a concern. Consequently, they use it as the default register allocation algorithm when the compiler is run by the interactive development environment.

2.3.1 Basic Linear Scan Algorithm

The linear scan algorithm first arranges all instructions of a method in a linear order where all control flow structures like conditions and loops are hidden. Then, the lifetime intervals for all virtual registers are computed. Each interval starts at the first definition of the register and ends at its last use. A dataflow analysis is needed to take the effect of loops and conditions into account. The calculation of lifetime intervals is very conservative: Because holes are not allowed, registers are considered continuously live from the first definition to their last use.

Figure 2.6 shows the same example as presented for graph coloring in Chapter 2.2. It should be processed again with two physical registers `r1` and `r2`. Because of the conservative approach, the lifetime interval of register `v1` is continuous from instruction (1) to instruction (7).

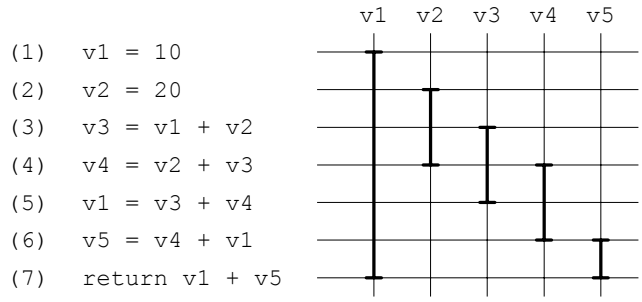


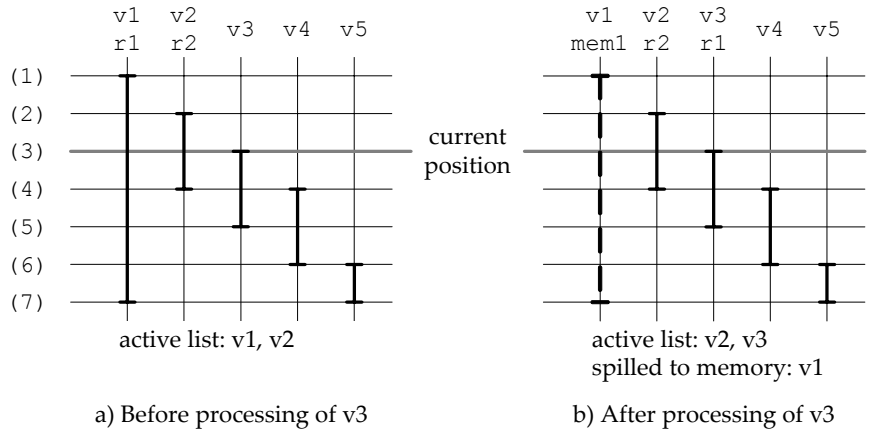
Figure 2.6: Linear Scan example—code with live ranges

The linear scan algorithm operates directly on the list of intervals, sorted by their start positions. The compiler iterates over the list and assigns a physical register to the interval immediately. If no physical register is available for the whole lifetime, then some intervals must be spilled to memory. Two lifetime intervals interfere if their ranges intersect. So two intervals that do not intersect can get the same physical register assigned.

The allocation is done by iterating over the sorted list of intervals. At each step, the algorithm maintains a list, called *active* list, which contains all intervals that overlap with the current position and have registers assigned. Intervals that ended already before the current position are removed from the active list because they are no longer relevant. The interval starting at the current position gets a physical register assigned that is not used by any interval in the active list. If all registers are already in use, one interval must be spilled—either an interval of the active list or the currently processed interval. It has turned out to be a good heuristic to spill the interval with the highest end position.

In the example, the intervals are processed in the order `v1`, `v2`, `v3`, `v4`, `v5`. The algorithm starts with an empty active list. At the first step, the interval `v1` is processed. Since the active list is empty, the first physical register `r1` is assigned to `v1`, and `v1` is added to the active list. When `v2` is processed at the next step, `v1` is still active and so `r2` is assigned to `v2`. Then `v2` is added to the active list.

Next, the interval `v3` is processed. Because the active list already contains `v1` and `v2` with the physical registers `r1` and `r2` assigned, no physical register is available for `v3` and one interval must be spilled. The algorithm selects `v1` for spilling because it has the highest end position and removes it from the active list. The memory location that is assigned to `v1` is called `mem1`. The register `r1` is no longer blocked and can be assigned to `v3`, which is added to the active list. Figure 2.7 shows the state of the intervals and the active list before and after processing of `v3`.



a) Before processing of v3 b) After processing of v3

Figure 2.7: Interval state before and after allocation of v3

When `v4` is processed, the end of `v2` has been reached and so `v2` is removed from the active list. Now `r2` is unused and can be assigned to `v4`. When `v5` is allocated, no other intervals are active, so `r1` can be assigned. Now all intervals have a register or a memory location assigned and the algorithm stops. Figure 2.8 shows the code where the virtual registers are replaced with their allocated physical registers.

```
(1)   v1 = 10            mem1 = 10
(2)   v2 = 20            r2 = 20
(3)   v3 = v1 + v2       r1 = mem1 + r2
(4)   v4 = v2 + v3       r2 = r2 + r1
(5)   v1 = v3 + v4       mem1 = r1 + r2
(6)   v5 = v4 + v1       r1 = r2 + mem1
(7)   return v1 + v5     return mem1 + r1
```

Figure 2.8: Example code before and after register allocation

The resulting allocation is not as good as the result obtained by graph coloring in Chapter 2.2 because one virtual register must be spilled to memory. This is a consequence of the conservative construction of lifetime intervals without holes. This slightly worse allocation is compensated by a much faster allocation. Since only one linear pass over the lifetime intervals is required, the linear scan algorithm has an asymptotic time complexity of O(n), where n is the number of virtual registers.

2.3.2 Second Chance Binpacking

Second chance binpacking is an extension of the basic linear scan algorithm which produces better code, but basically preserves the linear time complexity. One major shortcoming of the basic linear scan algorithm is the fact that it does not allow holes in live ranges. Especially complex control flow graphs tend to produce holes because of conditions and loops. Even in the simple example presented in the last chapter, the interval `v1` has a hole from instruction (3) to (5). Because this hole is neglected by the basic linear scan algorithm, the interval `v1` must be spilled to memory. Second chance binpacking is capable of handling holes in lifetime intervals.

Another extension of second chance binpacking is the possibility for splitting intervals: When an interval starts in an area with low register pressure, but then enters an area with high register pressure where no registers are available, the basic linear scan algorithm spills the entire interval to memory. So the interval is spilled, even if a register is available for a part of the interval. Second chance binpacking solves this problem by splitting intervals: The interval starts in a register, but is then split and spilled if the register is no longer available. It is also possible that a spilled interval gets reloaded into a different register later in its life—it gets a second chance to reside in a register.

Splitting intervals leads to a much better utilization of registers, but also has some drawbacks. Because the linear ordering of blocks does not take the real control flow into account, a second pass called resolution is needed. Move-instructions are inserted at control flow edges when an interval has multiple locations assigned. If, for example, an interval is in a register at the end of a basic block, but spilled to memory at the beginning of a successor block, a move must be inserted to save the register to memory when this control flow edge is processed. Second chance binpacking performs a data flow analysis to minimize the number of inserted moves.

If the example code of the last chapter is processed with second chance binpacking, the result is equal to graph coloring. Because lifetime holes are allowed, the live ranges shown in Figure 2.9 are identical to the ones presented for graph coloring in Figure 2.1 on page 7.

The first two intervals `v1` and `v2` get the physical registers `r1` and `r2` assigned, respectively. When `v3` is reached in the linear pass over all intervals, the interval of `v1` has just reached a lifetime hole. So `v1` is not contained in the *active* list, but in a new list called *inactive* list. This list contains all intervals that start before and end after the current position, but are currently in a lifetime hole. The physical registers of inactive intervals can partly be assigned to other intervals. In the example, the interval `v3` gets the physical register `r1` without any spilling.

Figure 2.9 shows the example code before allocation, the state of the intervals after allocation and the resulting code after assigning the physical registers. The resulting code is nearly the same as in the graph coloring result shown in Figure 2.5 on page 10, only some physical register numbers are swapped.

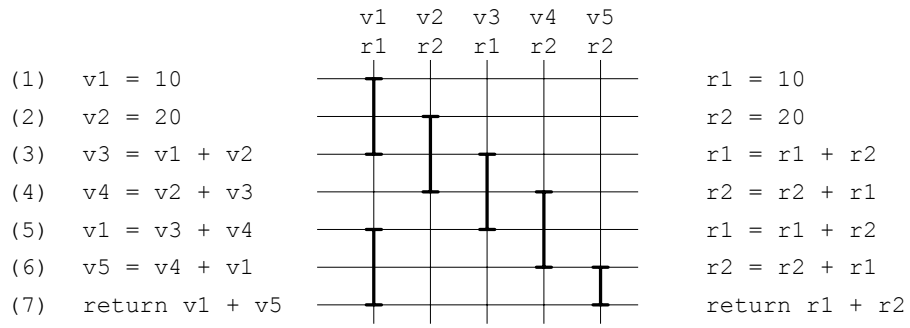


Figure 2.9: Second chance binpacking example

Analyzing the asymptotic time complexity of second chance binpacking is more complicated. While the actual pass over all intervals runs in linear time as in the basic algorithm, other parts like the data flow analysis cannot be performed in linear time. In summary, the overall asymptotic time complexity is higher than O(n). However, measurements show that only some percents of the total allocation time is spent in non-linear parts, so sacrificing linearity does not have a major impact. Second chance binpacking is nearly as fast as the basic linear scan algorithm and produces nearly as good code as graph coloring. It is a good tradeoff if both compilation time and runtime of a program matter.

Chapter 3

The Java Virtual Machine

This chapter starts with a description of the design goals of both the Java programming language and the Java virtual machine (JVM). The Java HotSpot VM is presented as the current JVM of Sun Microsystems. It is available in two variants, the server and the client version. Both share the same code base, but have different just-in-time compilers. The HotSpot Client Virtual Machine serves as the foundation for the register allocator presented in the next chapters.

The Java programming language [Gosling00] was developed by Sun Microsystems as a general-purpose, object-oriented and concurrent language. Although the syntax is similar to C++, the complex and unsafe features of C++ were omitted. Instead, many sophisticated concepts were added to simplify development and increase security. Java was designed as a portable language that runs on multiple host architectures and allows a secure delivery of software components.

The emerging of the World Wide Web contributed much to the success of Java. While the interactivity of plain HTML pages is limited, the integration of small Java programs into web pages enables the designers to use a full-blown programming language and to develop interactive applications that are seamlessly integrated in the web browser. Transferring executable code over an untrusted network like the Internet requires careful checks before execution to guarantee that no virulent code is executed on the client, as enforced by the Java specification.

Today, Java is used on a wide variety of systems: Small embedded systems like mobile phones and PDAs can be programmed easily without having to know much about the target architecture using *Java 2 Platform Micro Edition* (J2ME). Midway in the spectrum, *Java 2 Platform Standard Edition* (J2SE) provides a complete environment for desktop applications, supporting the developer with an extensive library for graphical user interfaces, network programming, XML processing, and multimedia applications. Most integrated development environments for Java are also written in Java itself.

The development of component-based multi-tier enterprise applications is facilitated by *Java 2 Platform Enterprise Edition* (J2EE), providing a large framework that significantly simplifies the development of secure and transaction-oriented server applications. Using one programming language for all types and sizes of systems is an advantage over specialized languages and can reduce the time and costs of software development.

To guarantee the portability and platform independence, Java applications are not distributed in native code for a specific hardware platform. Instead, the concept of a *Java virtual machine* (JVM) is used for abstraction. Java source code is compiled to a compact binary representation called *Java bytecodes* which is interpreted by the JVM. The application is stored in a well defined binary format, the *class file format*, containing the bytecodes together with a symbol table and other ancillary information. The Java virtual machine is defined independently from the Java programming language; only the class file format connects these parts.

3.1 Abstract Specification of a JVM

Virtual machines are a widely known concept to obtain platform independence and to conceal limitations of specific hardware architectures. In general, a virtual machine emulates an abstract computing architecture on a physically available hardware. Because virtual machines are just a piece of software, the restrictions of hardware development are not relevant. It is possible to extend the core execution unit with high-level components, e.g. for memory management, thread handling and program verification. The instruction set of a virtual machine can therefore be on a higher level than the instruction set of a physical processor. This leads to a small size of the compiled code, where a single-byte instruction can perform a quite complex action.

3.1.1 Structure of a JVM

The Java virtual machine, as specified in [Lindholm99], is a stack machine that executes bytecodes. It defines various runtime data areas that are used for the execution of a program. While some data areas exist only once per virtual machine, others are created for each executed thread. Figure 3.1 shows the basic structure of a Java virtual machine.

When a Java virtual machine is started, the global data structures are allocated and initialized. The *heap* models the main memory of a JVM. All Java objects are allocated on the heap. While the allocation of an object is invoked by the executed program, the deallocation is never performed explicitly. Instead, objects that are no longer reachable by the program are automatically reclaimed by a garbage collector. As an advantage, a Java program cannot cause memory errors such as memory leaks or accesses to already freed objects.

Before a method of a class can be executed, the class must be loaded into the JVM. The main parts of a class are the bytecodes that are later executed, the constant pool that acts as an extended symbol table, and some other data structures. The bytecodes of the class are loaded to the *method area* that is shared among all threads. The constants are loaded to the *constant pool*.

The starting of a new thread implies the creation of the per-thread data structures. Because threads are part of the Java specification, each JVM must be capable of executing multiple threads simultaneously. Basic means for the synchronization of threads are also part of the specification. Each thread has its own *stack* and a register for the *program counter*.

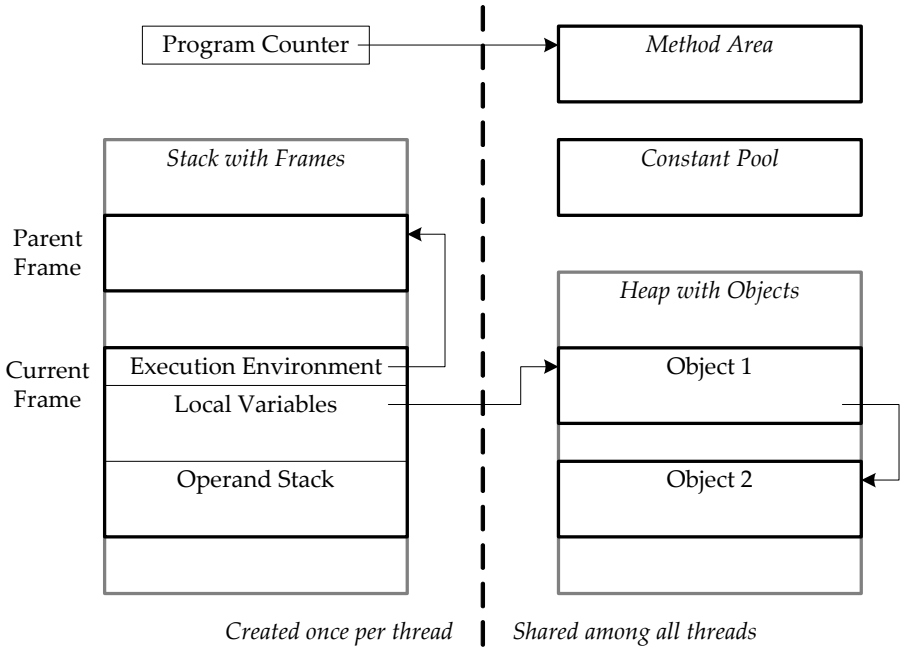


Figure 3.1: Structure of a Java virtual machine

When a method is called, a new frame is allocated on the stack. This frame is then referred to as the *current frame*. The program counter points to the bytecode in the method area that is currently executed. A frame contains the following data structures:

- The *execution environment* is used for bookkeeping of stack frames. It contains at least a dynamic link to the frame of the caller method (the parent frame) where the control flow returns to on exit of the current method.
- The *local variables* section contains all local variables of the current method invocation. Local variables are loaded and stored explicitly with dedicated bytecodes.
- The *operand stack* is used as a temporary workspace for the execution of bytecodes. Most bytecodes take their parameters from the operand stack and put their result back on it. Because bytecodes always operate implicitly on the top of the stack—arguments are popped from, results are pushed onto the stack—it is not necessary to specify the arguments explicitly for most bytecodes.

3.1.2 Implementation

The actual implementation of a Java virtual machine must adhere to the JVM specification to guarantee portability, but implementation details are not part of the specification. This allows the vendor of a JVM to implement sophisticated optimizations. Especially, it is not specified how bytecodes are executed. The JVM can interpret them, compile them to native code before execution or mix both kinds. Similarly, the specification does not mandate any particular internal structure for the representation of objects or a concrete algorithm for garbage collection. So the abstract specification of a JVM is on the one hand detailed enough to guarantee portability and compatibility, and on the other hand abstract enough to cede implementation details to the actual vendor of a JVM.

3.2 The Java HotSpot Virtual Machine

The *Java HotSpot Virtual Machine* is the current JVM by Sun Microsystems that provides the foundation of their Java Development Kit. It covers the whole lifecycle of a Java application—from development and debugging in integrated development environments up to the execution on enterprise servers. This chapter presents a brief overview of the HotSpot virtual machine, described in more detail in [Sun02].

The HotSpot VM is available on a wide variety of platforms and operating systems: Sun supports the Sparc architecture of Sun, the IA-32 and IA-64 architectures of Intel and the 64-bit extensions of AMD, running with different operating systems like Sun Solaris, Microsoft Windows and Linux. Editions for other platforms and operating systems, such as Apple's Mac OS X, are also available through Java technology licensees. This guarantees the platform-independent execution of Java applications on all major architectures available today.

3.2.1 Subystems

The core runtime system of the Java HotSpot VM is responsible for initializing the internal data structures and starting the Java application. This includes all steps that are necessary for loading and verifying class files. Then the execution of an application starts in the interpreter.

Java programming language threads are mapped one-to-one to operating systems threads. Therefore, all thread scheduling strategies of the host operating system are available automatically. The synchronization of threads is implemented very efficiently to support the fine-grained locking of the Java programming language [Agesen99].

Because the Java programming language is highly object-oriented and encourages the creation of objects even for small intermediate data structures, the memory model of the JVM must support fast access to objects. To prevent negative impacts due to subtype checks and calls to virtual methods, they are highly optimized: Subtype checks are implemented with caches covering nearly all checks [Click02], and virtual calls are optimized with polymorphic inline caches [Hölzle91].

The Java HotSpot VM uses a uniform and handleless memory model for all sorts of objects, including arrays and internal data structures. Implementing object references as direct pointers without using handles provides a very fast access to instance variables, but requires additional effort during garbage collection. Each object has a small header of only two machine-words for internal status information and a reference to the class of the object.

The VM uses a fully accurate garbage collector to free memory of objects that are no longer reachable. This means that the garbage collector can decide exactly for each object if it is still reachable from other objects or can be freed. Also, objects can be relocated by moving them to another location and updating all references to them. Because of the handleless memory model, the garbage collector must know all positions where an object is referenced, including references from other objects, the stack and even registers.

To make garbage collection efficient, it is necessary to employ different algorithms for different kinds of systems and applications. For example, on a single processor system it is possible to stop the entire application during garbage collection. This would lead to a significant decrease of performance on multiprocessor systems, where it is desired that garbage collection runs concurrently with the normal application threads. Therefore, the HotSpot VM implements different garbage collection algorithms [Sun03].

Debugging a Java application, e.g. by using the debugger of an integrated development environment, needs support from the underlying virtual machine. Because the internal data structures such as the stack and heap layout are not exposed to the running Java application, the VM must provide a special interface for debuggers to retrieve this information. For these purposes, the Java HotSpot VM implements the *Java Virtual Machine Debugger Interface*.

3.2.2 Just-in-Time Compilation

When the execution of an application starts, all methods are interpreted first. Execution can start immediately after a class is loaded without any further delay. The interpreter is generated once at startup [Griesemer99]. It consists of a dispatch loop that executes a fixed code template for each bytecode. The interpreter is a simple simulation of a processor that executes bytecodes: Each bytecode is loaded, the corresponding code template is searched and then executed.

Interpreting a method is rather slow because the template for each bytecode consists of several machine instructions, so the achievable performance is limited. Therefore, it is necessary to compile the bytecodes of the most frequently executed methods to machine code that can be executed directly without the interpreter. Because the compilation takes place while the program is executed, it is called just-in-time compilation.

The strategy for selecting the methods that are compiled is based on runtime information collected during interpretation. Each method has a method-entry and a backward-branch counter that are incremented at each start of the method and when a backward branch is executed, respectively. If these counters exceed a certain threshold, the method is scheduled for compilation. This strategy is based on the observation that virtually all programs spend most of their time in a small range of code. Therefore, the counters of frequently executed methods, called the “hot spots” of a program, soon reach the threshold and the methods are compiled without wasting much time interpreting them.

Methods that are executed infrequently, e.g. only once at the startup of the application, never reach the threshold and are never compiled. This greatly reduces the number of methods that are compiled, and the compiler can spend more time optimizing the machine code of the remaining methods. Using a mixture of interpreted and compiled code guarantees an optimal overall performance.

Additionally, this approach guarantees that each method is interpreted before it is compiled. So all classes that are used by the method are already loaded and methods that are called are known. Additionally, the interpreter collects runtime information such as the common runtime type of local variables. This information can be used by the compiler for

sophisticated optimizations that would not be possible if the methods were compiled before their first execution.

Some highly effective compiler optimizations are complicated by the semantics of the Java programming language. For example, most methods cannot be inlined because most method invocations are virtual. The actually called target is not known statically because the semantics of a call can change later on when classes are loaded dynamically into the running program.

Nevertheless, the compiler performs inlining of such methods optimistically. Hence, it is possible that a compiled method is later invalidated when a new class is loaded. In such a rare case, the method is compiled again without this optimization. But things are much more complicated if the invalidated method is currently executed and therefore stack frames of this method are active. In such situations, it must be possible to switch back from the compiled code to the interpreter. This transition is called *deoptimization* [Hölzle92]. The compiler must create meta data that allows the reconstruction of the interpreter state at certain points of the compiled code.

Deoptimization allows the compiler to perform aggressive optimizations that speed up the normal execution, but may seldom lead to situations where the optimization was too optimistic and must therefore be undone. There are some additional cases where a compiled method is deoptimized, e.g. when an asynchronous exception is thrown. The compiled code does not need to handle such complicated, uncommon cases.

A method is compiled when the counters of the method exceed a certain threshold. Typically, the decision is made before the execution of the method starts because no special handling is needed in this case to switch from the interpreted to compiled code: Instead of the interpreter, the compiled code is called. But this solution is not always sufficient. When an interpreted method executes a long running loop, i.e. when many backward branches are executed in the interpreter, then it is necessary to switch to compiled code while a method is running. This is called *on stack replacement* (OSR) of a method. In this case, a special version of the method is compiled with an OSR entry point that jumps directly into the loop.

Although the Java programming language is a structured language that does not allow arbitrary goto-operations, the bytecodes are not required to be structured. Therefore, the compiler can encounter situations that occur rarely, but are difficult to handle. Instead of inflating the compiler with code for handling all special cases that are probably never needed, these situations are handled with a compilation *bailout*. The compilation of the method is stopped and the execution is continued in the interpreter. Because compilers for the Java programming language do not create such complicated structures of bytecodes, this is not a real limitation and does not degrade the performance.

Figure 3.2 summarizes the possible transitions between interpreted and compiled methods: Normally methods are compiled on method-entry counter overflows, but methods with long running loops can be OSR-compiled. When the compilation is not possible because the method is too complicated, then the compiler stops with a bailout, otherwise the compiled code is executed until a deoptimization is necessary. It should be noted again that bailouts and deoptimizations occur very rarely.

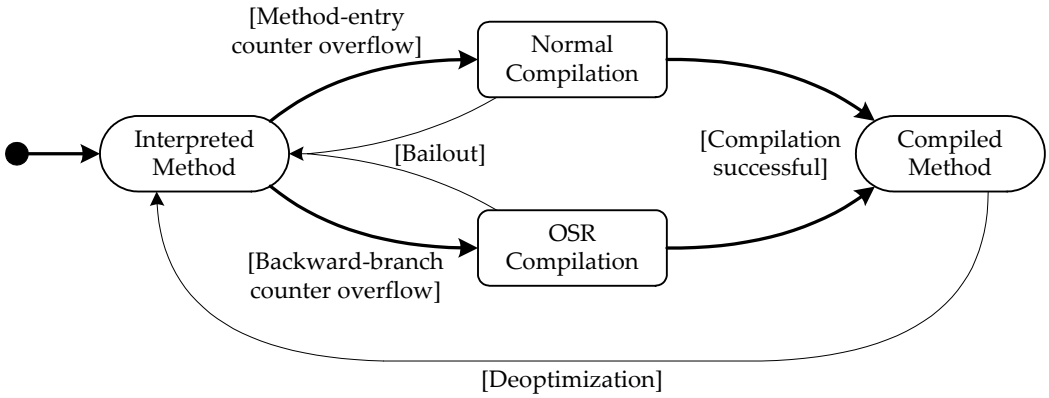


Figure 3.2: Transitions between interpreted and compiled methods

The just-in-time compiler is separated from the runtime of the HotSpot VM by a well-defined compiler interface. It initiates the compilation of a method and provides all necessary data such as access to the bytecodes, to the type information of variables and fields and to methods that are called. The compiler interface supports parallel compilation and execution of code, parallel compilation and garbage collection, and parallel compilation of different methods.

Currently, the HotSpot VM is available in two versions: the client and the server VM. The *Java HotSpot Client VM* is best for running interactive applications and is tuned for fast application start-up and low memory footprint. The *Java HotSpot Server VM* is designed for maximum execution speed of long running server applications. Both share the same runtime, but include different just-in-time compilers (the client compiler and the server compiler). The client compiler is internally named C1, the server compiler C2. The next chapters describe both compilers and present their differences.

3.2.3 Server Compiler

The *Java HotSpot Server Compiler* (described in [Paleczny01]) is a fully optimizing compiler that performs all classic optimizations of traditional compilers, like common subexpression elimination, loop unrolling and graph coloring register allocation. It also features Java specific optimizations, such as inlining of virtual methods, null-check elimination and range-check elimination. These optimizations reduce the overhead necessary for guaranteeing safe execution of Java code to a minimum. The compiler is highly portable and available for many platforms. All machine specific parts are factored out in a machine description file specifying all aspects of the target hardware.

The extensive optimizations lead to a high code quality and therefore to a short execution time of the generated code. But the optimizations are very time-consuming during compilation, so the compilation speed is low compared with other just-in-time compilers. Therefore, the server compiler is the best choice for long running applications where the initial time needed for compilation can be neglected and only the execution time of the generated code is relevant.

The server compiler uses an intermediate representation (IR) based on a static single assignment (SSA) graph [Click95]. Operations are represented by nodes, the input

operands are represented by edges to the nodes that produce the desired input values (data-flow edges). The control flow is also represented by explicit edges that need not necessarily match the data-flow edges. This allows optimizations of the data flow by exchanging the order of nodes without destroying the correct control flow.

The server compiler proceeds through the following steps when it compiles a method: Parsing of the bytecodes, machine-independent optimizations, instruction selection, global code motion and scheduling, register allocation, peephole optimization and at last code generation.

The parser needs two iterations over the bytecodes. The first iteration identifies the boundaries of basic blocks. A basic block is a straight-line sequence of bytecodes without any jumps or jump targets in the middle. The second iteration visits all basic blocks and translates the bytecodes of the block to nodes of the IR. The state of the operand stack and local variables that would be maintained by the interpreter is simulated in the parser by pushing and popping nodes from and to a state array. Because the instruction nodes are also connected by control flow edges, the explicit structure of basic blocks is revealed. This allows a later reordering of instruction nodes.

Optimizations like constant folding and global value numbering for sequential code sequences are performed immediately during parsing. Loops cannot be optimized completely during parsing because the loop end is not yet known when the loop header is parsed. Therefore, the above optimizations, extended with global optimizations like loop unrolling and branch elimination, are re-executed after parsing until a fixed point is reached where no further optimizations are possible. This can require several passes over all blocks and is therefore time-consuming.

The translation of machine-independent instructions to the machine instructions of the target architecture is done by a bottom-up rewrite system (BURS, [Pelegri88]). This system uses the architecture description file that must be written for each platform. When the accurate costs of machine instructions are known, it is possible to select the optimal machine instructions.

Before register allocation takes place, the final order of the instructions must be computed. Instructions linked with control flow edges are grouped to basic blocks again. Each block has an associated execution frequency that is estimated by the loop depth and branch prediction. When the exact basic block of an instruction is not fixed by data and control flow dependencies, then it is placed in the block with the lowest execution frequency. Inside a basic block, the instructions are ordered by a local scheduler.

Global register allocation is performed by a graph coloring register allocator as presented in Chapter 2.2 on page 6. First, the live ranges are gathered and conservatively coalesced, afterwards the nodes are colored. If the coloring fails, spill code is inserted and the algorithm is repeated. After a final peephole optimization, which optimizes processor-specific code sequences, the executable machine code is generated. This step also creates additional meta data necessary for deoptimization, garbage collection and exception handling. Finally, the executable code is installed in the runtime system and is ready for execution.

3.2.4 Client Compiler

The server compiler provides an excellent peak performance for long running server application. However, it is not suitable for interactive client applications because the slow compilation leads to noticeable delays in the program execution. The peak performance is not apparent to the user because client applications spend most of their time waiting for user input.

The client compiler has a directly opposing goal to the server compiler: It achieves a significantly higher compilation speed because it omits time-consuming optimizations. As a positive side effect, the internal structure of the client compiler is much simpler than the server compiler. It is separated into a machine-independent front end and a partly machine-dependent back end.

First, the front end builds a high-level intermediate representation (HIR) by iterating the bytecodes twice (similar to the parsing of the server compiler). Only simple optimizations like constant folding are applied. Then, the innermost loops are detected to facilitate the register allocation of the backend.

The back end converts the HIR to a low-level intermediate representation (LIR) similar to the final machine code. A simple heuristic—similar to the local method described in Chapter 2.1 on page 6—is used for register allocation: At the beginning it assumes that all local variables are located on the stack. Registers are allocated when they are needed for a computation and freed when the value is stored back to a local variable. If a register remains completely unused inside a loop or even in the entire method, then this register is used to cache the most frequently used local variable. This reduces the number of loads and stores to memory especially on architectures with many registers.

To determine the unused registers, the same code generator is run twice: In the first pass, the code emission is disabled and only the allocation of registers is tracked. After any unused registers are assigned to local variables, the code generator is run again with code emission enabled to create the final machine code.

The first implementation of the client compiler (described in [Griesemer00]) used the HIR only; the back end generated native code without a prior generation of the LIR. This version was shipped with the Sun JDK 1.3. The LIR was implemented for the client compiler of the JDK 1.4 to enable peephole optimizations after register allocation.

3.2.5 Research Client Compiler

The linear scan register allocator developed for this master thesis is implemented in a research version of the client compiler. It mainly uses the same structure as the product compiler shipped with the current Sun JDK 1.4.2 and the upcoming version 1.5. It uses the same intermediate representations, i.e. HIR and LIR. The front end was modified to generate the HIR in static single assignment (SSA) form that helps to implement other optimizations like common subexpression elimination. The back end now uses the linear scan algorithm for register allocation instead of the old heuristic. The detailed structure of this new compiler is presented in Chapter 4.

Compiler Architecture

This chapter presents all steps necessary to compile the bytecodes of a method to native code that is directly executable by the processor. It describes the structure and instruction set of the bytecodes, the native code and the two intermediate representations that are used by the compiler. When a method is compiled, the bytecodes are first transformed to the graph-based high-level intermediate representation (HIR). Several optimizations are applied before the HIR is converted to the low-level intermediate representation (LIR). After register allocation, the native code is created from the LIR, together with additional meta data that are required by the virtual machine, e.g. for garbage collection.

The linear scan register allocator presented in this thesis is integrated in the research version of the Java HotSpot client virtual machine. Compared with the product version shipped with the current Sun JDK 1.4.2, the just-in-time compiler was extended in a research project to support more general optimizations. The history of this research work was already presented in Chapter 1.1.

This chapter explains the architecture of the research compiler. Whereas the overall structure is equal to the product version, many details are different. Consequently, the following explanations cannot be used as a reference for the product compiler. The term "compiler", when used without prefix, will henceforth refer to the research version of the Java HotSpot client compiler. Because the research compiler is work in progress, concepts and algorithms might be replaced by better ones. This thesis is based on the snapshot from August 2004.

4.1 Overall Structure

The compiler is responsible for translating the bytecodes of a method to native machine code while the VM is already executing the application. Although a direct compilation without an intermediate representation would be possible (and is implemented in other projects, for example in [AdlTabatabai98]), the options for optimizations would be very limited. Intermediate representations simplify the implementation of optimizations because they represent methods in a regular and easy to manipulate form, independent from the target architecture.

Two intermediate representations are used during compilation: The *high-level intermediate representation* (HIR) and the *low-level intermediate representation* (LIR). They separate the compiler into a *front end* that constructs the HIR from the bytecodes, and a *back end* that generates the LIR from the HIR and finally the machine code from the LIR. Figure 4.1 shows the overall structure of the compiler.

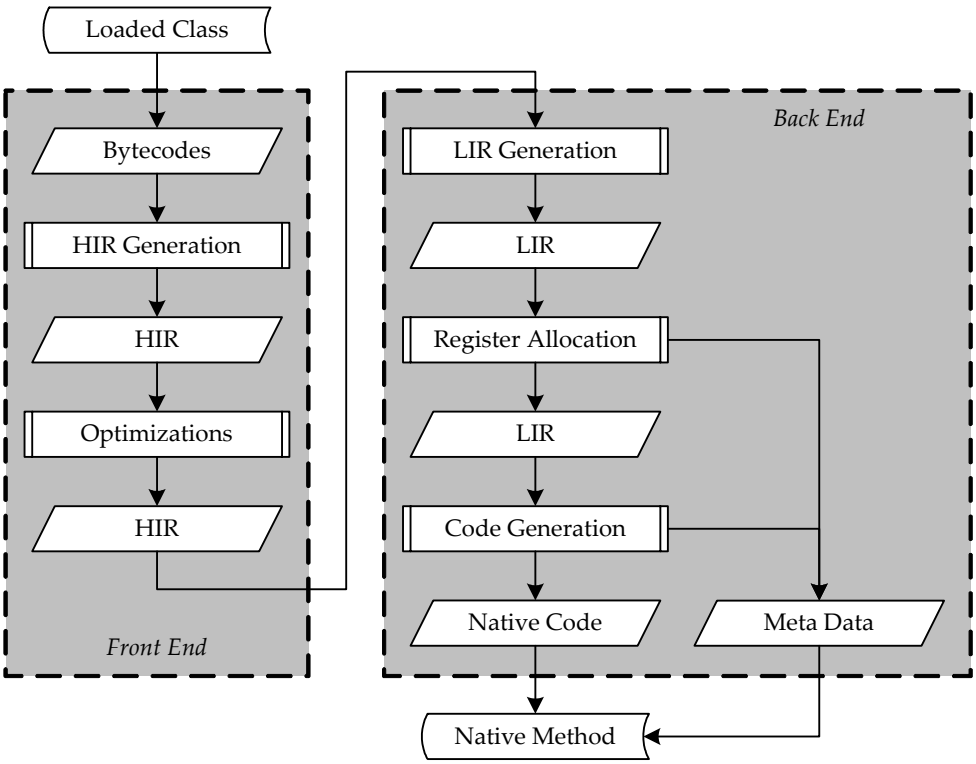


Figure 4.1: Overall compiler architecture

In the following chapters, details and descriptions for each item mentioned in Figure 4.1 are presented. The order does not strictly follow the actual data and control flow; instead a logical order is used.

4.2 Bytecodes

Before a Java class can be executed in the JVM, the Java source code must be compiled to Java bytecodes. This frees the JVM from the time-consuming task of parsing and analyzing plain-text source code. Instead, the bytecodes provide a compact binary representation of the class that can be executed directly by an interpreter. It also simplifies the validity checks of bytecodes because strict rules are defined in the specification [Lindholm99].

4.2.1 Example

The example in Figure 4.2 presents the Java source code of a short method that calculates and returns the factorial of an integer number. This example will be used throughout this chapter to illustrate the different intermediate representations and algorithms.

```
public static int factorial(int n) {
  int p = 1;
  while (n > 0) {
    p = p * n;
    n = n - 1;
  }
  return p;
}
```

Figure 4.2: Compilation example—Java source code

This source code is compiled to the bytecodes in Figure 4.3. Note that the bytecodes are stored in a compact binary representation; the example method needs only 19 bytes. The number to the left of each bytecode refers to its index relative from the beginning of the method. It is commonly called *bytecode index* (bci). The comments on the right side show the corresponding source code instructions.

```
0:  iconst_1
1:  istore_1    // p = 1
2:  iload_0
3:  ifle 17     // while (n > 0)
6:  iload_1
7:  iload_0
8:  imul
9:  istore_1    // p = p * n
10: iload_0
11: iconst_1
12: isub
13: istore_0    // n = n - 1
14: goto 2      // end of while-loop
17: iload_1     // return p
18: ireturn
```

Figure 4.3: Compilation example—Java bytecodes

4.2.2 Instruction Set

The instruction set of the bytecodes consists of over 200 different instruction codes. This chapter gives a coarse classification; a detailed description of each instruction is contained in the specification [Lindholm99]. Each instruction code is stored in a single byte. Some instructions take additional parameters, but many consist of the instruction code only. As described in Chapter 3.1.1, the instructions are executed using an operand stack. The current top of the operand stack is available as an implicit parameter for all instructions. The instruction codes can be grouped into the following categories:

- Local variables are accessed with instructions that push a single local variable on the operand stack or store the stack top back into a local variable. Each local variable has a unique number that is supplied as a parameter to these instructions. There are also instructions that push a constant on the operand stack.
- Access to objects is provided by instructions that load an object field or an array element to the operand stack. Similar instructions are available for storing. These instructions throw a runtime exception if the referenced object is null of if the array index is out of bounds.

- Arithmetic and logical instructions usually pop two parameters from the operand stack, perform the specified operation, and push the result back on the stack. All common instructions like addition, subtraction, multiplication and division are available, together with instructions that perform logical operations or compare two values.
- Instructions for type conversion are used to convert between the different integer and floating point types of the Java programming language. Explicit conversion instructions are necessary because all instructions are strictly typed and operate only on operands of a single type.
- Conditional and unconditional jumps are available for branches and loops inside a single method. The target of the jump is the bytecode index, supplied as an explicit parameter.
- Call instructions are used to call other methods. The receiver of the method and the parameters must be present on the operand stack, whereas the name of the method is supplied as an explicit parameter. A method is ended normally with one of the return instructions.
- Special instructions are available for direct manipulations of the operand stack, e.g. the duplication of the current stack top. These are the only instructions that are not strictly typed.
- Some high-level instructions are available for performing type checks, synchronization of threads, exception handling and allocation of objects and arrays.

In addition to the bytecodes, a class file contains meta information for each method, such as the number of local variables, the maximum size of the operand stack and exception handler tables. These tables are used to find the appropriate exception handler if an exception is thrown at a certain bytecode index.

4.3 Native Code

The main result of the compilation is native code that can be executed directly by the processor. The compiler can be built for two platforms: The Intel IA-32 architecture and the Sparc architecture. This thesis deals only with the IA-32 architecture because all implementation and testing was done on it. The porting to Sparc is periodically done by Sun Microsystems.

Most algorithms are implemented in a platform-independent way. Only special characteristics of a certain platform that are not available on other architectures require platform-dependent code. Focusing on the Intel IA-32 architecture does not restrict the generality because it needs most of the special handling: While Sparc is a regular RISC architecture, IA-32 is a CISC architecture with a highly irregular instruction set.

The floating point unit (FPU) is one of the most irregular parts of the IA-32 architecture. Chapter 6 starting on page 79 describes the structure of the FPU in detail, so it is not discussed further here. The following parts of this section present the basic principles of the IA-32 architecture [Intel1], together with the stack layout used by the compiler.

4.3.1 Intel IA-32 Architecture

The roots of the Intel IA-32 architecture date back to the Intel 8086 processor presented in 1978 and even earlier processors. Although the 8086 processor used only a 16-bit architecture and implemented a small subset of today's processors, binary code for it still executes on the newest Pentium 4 processors. This absolute compatibility is the main reason for the irregular instruction set.

The basic execution environment consists of the main memory accessible through the address space, general-purpose data registers, segment registers, the flags register and the instruction pointer register.

4.3.2 Address Space

Any program running on an IA-32 processor can address its own virtual address space of up to 4 GBytes that is mapped to a physical address space of up to 64 GBytes. Normally the memory is accessed using the memory management facilities of the processor. Two major addressing modes are available:

- When the *flat memory model* is used, the memory appears as a single, continuous address space. All information of the program like code, data and procedure stacks are located in this address space.
- In the *segmented memory model*, the memory is separated into independent address spaces called segments. Six segment registers are available for the fast access to segments.

The HotSpot VM uses the flat memory model, only some special functions use segments. For example, the current thread can be accessed fast because the pointer to the thread object is stored on a fixed address of a special segment.

4.3.3 Register Set

The IA-32 architecture provides eight general-purpose registers, called `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp` and `esp`. They are used for operands of arithmetic and logical instructions, for operands of address calculations and for memory pointers. Although all registers could be used freely, two registers have a special meaning:

- The `esp` register holds the *stack pointer* (the current top of the procedure stack) and should not be used for any other purpose. All instructions supporting the stack management implicitly use this register.
- The `ebp` register is often used as the *base pointer* of the current method on the stack, i.e. the two registers `ebp` and `esp` span the range of the stack frame for the current method.

Although it is possible to generate machine code that does not use `ebp` for stack handling, many systems—including the HotSpot client compiler—always use `ebp` for this purpose because it simplifies the handling of method calls. Therefore, only six general-purpose registers are available for free use. In general, the compiler can decide freely which operands and memory pointers are stored in which registers, because most instructions can

operate on all registers. But there are also instructions that require their operands in fixed registers, e.g. the instructions for divisions and shifts. Fixed registers complicate the work of a compiler, as it is described in Chapter 4.8.3.

The flags register (called `eflags`) is used as a bit map for status flags, control flags and system flags. Some flags can be modified by special instructions, other flags are set implicitly by arithmetic operations. But it is not possible to modify or examine the whole register directly. In normal methods, the flags register is mostly used for conditional branches: arithmetic operations set or clear flags depending on the result value, e.g. flags are set if the result is zero or the operation generated a carry. Succeeding conditional branches use these flags to decide if the branch must be taken.

The instruction pointer register (called `eip`) holds the address of the next instruction to be executed. It cannot be accessed directly, but is modified implicitly by control flow instructions like jumps, calls and returns. In a normal sequential control flow, it is increased automatically by the length of the current instruction.

Floating point instructions operate on eight floating point registers (called the x87 FPU registers) that are organized as a stack. The MMX extensions use eight MMX registers that are mapped to the x87 FPU registers. The SSE and SSE2 extensions operate on their own sets of eight XMM registers.

4.3.4 Operands

IA-32 instructions operate on zero or more operands. Some operands are specified implicitly by the instructions, but most operands are specified explicitly. The IA-32 architecture uses immediate operands, register operands and memory operands.

- *Immediate operands* are encoded in the instruction itself. They are mainly used for constants in arithmetic and logical operations and for targets of jumps. Only integer values are allowed as immediate operands, so they cannot be used in floating point operations.
- *Register operands* can be used as the source and result of all instructions. Depending on the instruction being executed, the general-purpose registers, the x87 FPU registers, the MMX registers or the XMM registers are used.
- *Memory operands* are also allowed as the input and result of many instructions. The address is calculated in its most general form by adding up a base register, an index register multiplied by a scale factor and a constant displacement. The address is calculated as `base` + (`index` * `scale`) + `displacement`. The base register and the index register are general-purpose registers. The scale factors is limited to 1, 2, 4 or 8, whereas the displacement can be an arbitrary integer number.

The general instruction format of the IA-32 architecture allows the specification of two operands only, whereof one operand must be a register operand. This enforces the two-operand form for arithmetic and logical instructions: The result is always stored in the left input operand, e.g. it is not possible to add two registers and store the result in a third register.

4.3.5 Instruction Set

The instructions of the IA-32 instruction set can be classified into the following coarse groups. In general, each new processor generation introduced a new group of instructions to the existing instruction set.

- *General-purpose instructions* are used for basic data movement, for arithmetic and logical computations of integer values and for controlling the program flow. They operate on data and addresses stored in the general-purpose registers. The core instructions were already available in the Intel 8086 processor.
- *Floating point instructions* are executed in the floating point unit (FPU) of the processor. Because of the historic separation of the FPU in a coprocessor for the Intel 386 processor, the floating point operations do not use the general instruction format of all other IA-32 instructions. Since the Intel486 processor, the FPU is integrated in all processors and therefore generally available.
- The *MMX extensions* introduced the single-instruction multiple-data (SIMD) concept to the IA-32 architecture: One 64-bit MMX register contains up to 8 independent integer values, so one MMX operation executes up to 8 calculations at once.
- The *SSE* and *SSE2 extensions* expanded the SIMD concept to floating point values. SSE instructions operate on four single-precision floating point values, SSE2 instructions on two double-precision floating point values. All SSE and SSE2 instructions are also available in a scalar form operating only on one value, therefore the SSE and SSE2 extensions can be used as a complete replacement of the FPU. The compiler uses this approach, as described in Chapter 6.4 on page 87.

4.3.6 Stack Layout

The IA-32 architecture provides basic instructions for manipulating a procedure stack. The stack is a continuous array of memory locations, where items are placed on the stack via push instructions and removed from the stack via pop instructions. The `esp` register always contains the address of the current stack top. The stack always grows downwards, so a push decrements `esp` and a pop increments it.

The stack is used to store local data of a method and to pass parameters between methods. While the basic layout is fixed by the IA-32 architecture, the details of the stack frame for a method can be chosen freely. The stack layout used by the compiler is shown in Figure 4.4 on the next page.

The base pointer (`ebp`) always points to the beginning of the stack frame for the current method. The stack pointer (`esp`) points to the end of the stack frame, which is also the current stack top. Each stack frame contains the following three parts:

- The monitor area is used for synchronization. When an object is locked by the current method, internal parts of the object are moved to the stack, ensuring a fast synchronization of objects. The size of this area depends on the maximum number of objects that are locked simultaneously by the method.

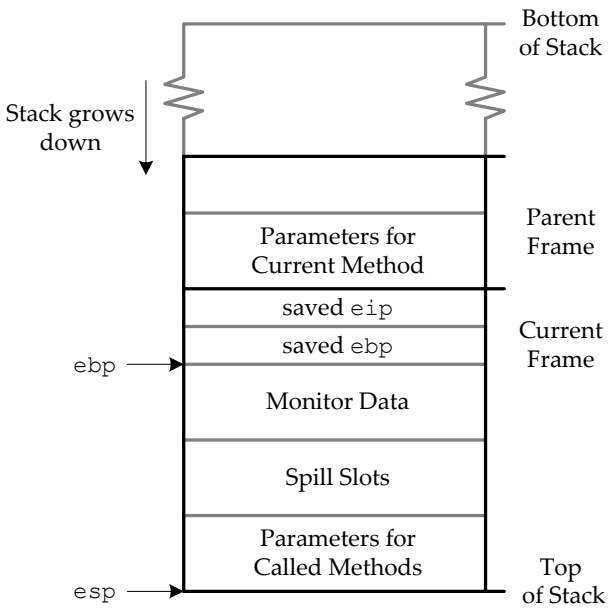


Figure 4.4: Stack layout

- The spill area contains spill slots where values used by the method are stored if no registers are available for them. Conceptually, a spill slot of a compiled method is similar to a local variable used by the Java bytecodes. But there is no direct relationship between certain spill slots and local variables. Spill slots are assigned by the register allocator where local variables are no longer explicitly visible.
- The parameter area is used for passing method parameters when a method is called. The size depends on the maximum number of arguments of all methods that may be called by the current method.

When a method calls another method, the calling method stores the parameters into its own parameter area on the stack. The parameter area belongs to the stack frame of the calling method, although the called method can access it. The instruction pointer (`eip`) and the base pointer (`ebp`) of the calling method are pushed onto the stack to allow a later return to the caller. The stack frame of the called method is completed by setting the new values for `ebp` and `esp`. Stack slots are not initialized because it is guaranteed that each slot is written before it is read.

Most stack slots are addressed via `ebp`: The parameters of the current method are accessed with a positive offset added to `ebp`, the spill slots with a negative offset. Only the parameters of called methods are accessed with a positive offset to `esp`. In contrast to the usual calling convention of most programming languages, the register `esp` is not changed itself when the parameters are stored because the parameter area is initialized with a sufficient size when the stack frame is created. This simplifies the handling of method parameters by the garbage collector.

4.4 High-Level Intermediate Representation

The high-level intermediate representation (HIR) is a graph-based representation of the method using static single assignment (SSA) form [Cytron91]. In the compiler, the HIR is mostly referred to as IR for historic reasons, so the classes that represent the HIR start with the prefix IR. The HIR is completely platform-independent and represents the method at a high level where global optimizations are easy to apply.

4.4.1 Instruction Set

All nodes of the HIR are subclasses of the base class Instruction. The class hierarchy consists of about 50 classes that store additional data about each instruction. Figure 4.5 shows a small subset of all classes together with their most important fields. The class diagram is incomplete and simplified to abstract from implementation details, but shows the coarse structure of the HIR.

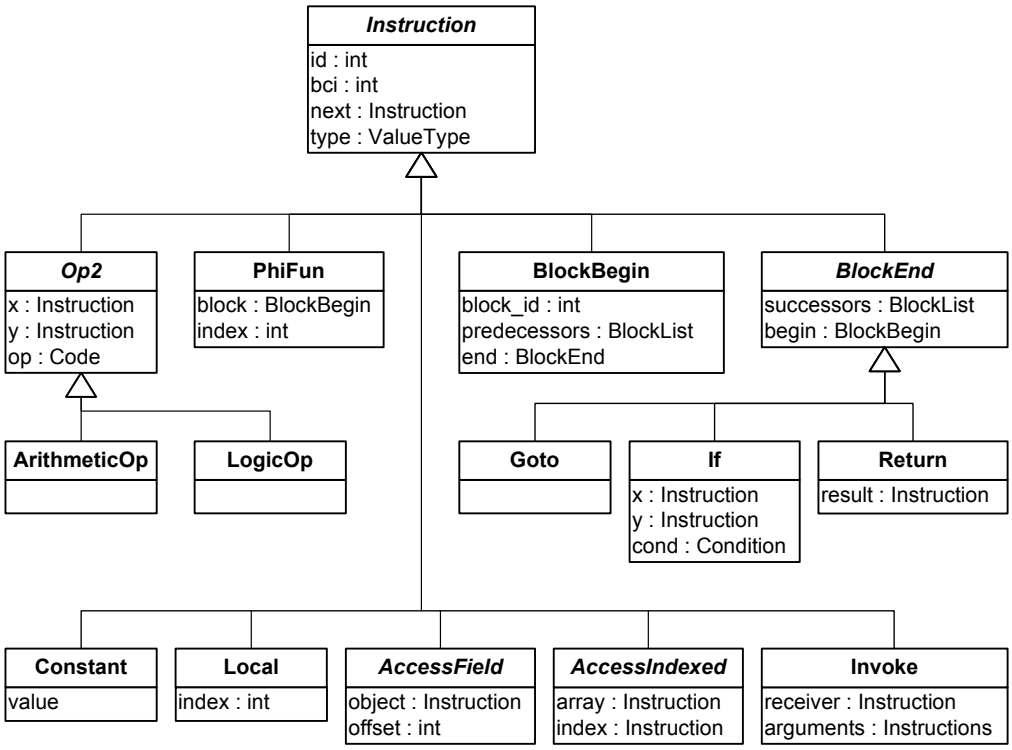


Figure 4.5: Class hierarchy for HIR instructions

4.4.2 Representation of Control Flow

The control flow of basic blocks is represented by BlockBegin and BlockEnd nodes: The first instruction of a basic block is always a BlockBegin, the last instruction always a concrete subclass of BlockEnd. The control flow is represented by the fields predecessors and successors of the two classes, containing a list of BlockBegin nodes of the preceding or succeeding blocks, respectively. The BlockBegin and BlockEnd nodes of a basic block are

linked together by the fields begin and end. This allows the fast traversal of the control flow graph without the need to traverse every single instruction.

While the start of a basic block is represented by the single concrete class BlockBegin, different subclasses of the abstract class BlockEnd are available for the end of a basic block. They differ in the number of successors and the code generated to jump to successors. Goto always has exactly one successor and represents an unconditional jump, If represents a conditional branch to one of its two successors and Return has no successors at all because it ends the current method. Additional classes derived from BlockEnd are available for switch statements with multiple successors.

The body of a basic block is formed by a sequential list of instructions, where the instructions are linked via the field next of the base class Instruction. Using a linked list for the instructions allows fast manipulations of the graph. When the HIR is generated, the instructions are added in the original order of the bytecodes. Instructions can be inserted into and removed from the graph when the HIR is optimized later. The left side of Figure 4.6 shows a simple control flow graph consisting of three basic blocks. The right side shows the details for the middle basic block with two predecessors, two successors and some instructions between the BlockBegin and the BlockEnd.

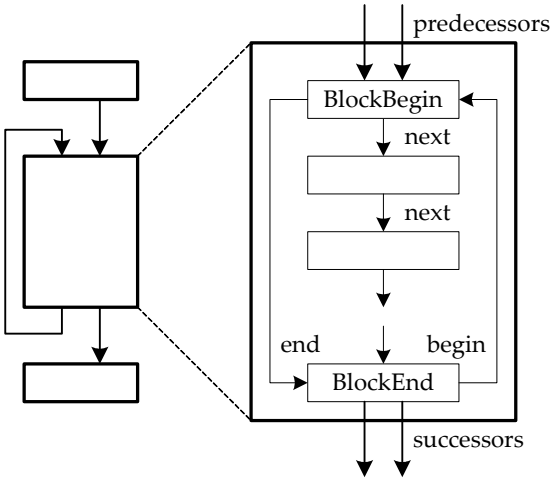


Figure 4.6: Control flow graph with details for one basic block

4.4.3 Representation of Data Flow

The data flow is also embodied in the HIR: Instructions refer to their arguments via pointers, and the arguments are instructions themselves. An instruction represents the computation of a result and the result itself. For example, the class ArithmeticOp represents an arithmetic operation with two input parameters x and y, which are instructions defined before. But it also represents the result of the operation, so it can be used as the input parameter of subsequent instructions. Because of this equivalence, an instruction is often referred to as a *value*.

The class diagram of Figure 4.5 contains only a small subset of all implemented instruction classes. Classes are available for elementary arithmetic and logical operations, loading and storing of fields and arrays, converting between data types and invoking other methods.

Special high-level instructions are used for type checks, synchronization, allocation of new objects and exception handling.

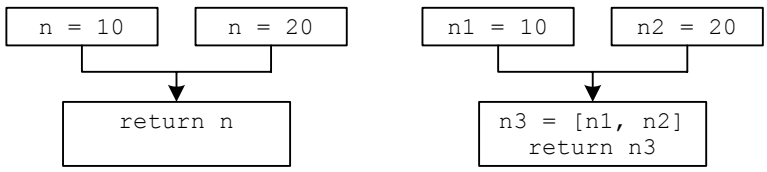
However, no instructions for accessing local variables are necessary. When a local variable is stored, the instruction creating the value is put in a *state array*. For every local variable, the state array stores its current value, i.e. the instruction where this value was computed. The array is indexed by the variable's number. When a local variable is referenced by another instruction, its value is taken directly from the state array, eliminating the need for an explicit load instruction. Only method parameters are explicitly represented by the class Local. An instruction can also reference instructions defined in another basic block.

4.4.4 Static Single Assignment Form

The static single assignment (SSA) form [Cytron91] is a special form of intermediate representations used by many compilers. The basic idea is that for every variable there is only a single location where it is assigned. If there are multiple assignments to the same variable, the program is transformed such that each assignment uses a new variable. This guarantees that two references with the same name always represent the same value.

As described in the previous section, a local variable is represented in the HIR by the instruction that calculated the value, and the instruction is registered in the state array. The state array at the start of a basic block is initialized with the state at the end of its predecessor. When a block has only one predecessor, the state of this block can be copied. When a block has more than one predecessor, the states of the predecessors must be merged. For this purpose, so-called phi functions are used. A phi function belongs to a block and has as many operands as the block has predecessors.

Figure 4.7 shows a simple example of a phi function: The local variable `n` is assigned twice, so the transformation to static single assignment form renames the variable to `n1` and `n2`. At the beginning of the succeeding block, the states of the predecessors must be merged and a phi function, called `n3`, is created. The syntax `n3 = [n1, n2]` means that the current value of `n3` is `n1` if the first (left) predecessor was executed, and `n2` if the second (right) predecessor was executed. As a result, the variable `n3` has a single point of definition, although it gets different values depending on the control flow.



a) without SSA form b) with SSA form

Figure 4.7: Example of SSA form and phi functions

The compiler creates phi functions conservatively for all local variables if a block has more than one predecessor. This can lead to phi functions where all operands are equal. Such phi functions can be simplified after the HIR is constructed. More details about the concept of phi functions used by the compiler are presented in [Mössenböck00].

4.4.5 Example

This chapter shows the HIR for the example presented in Chapter 4.2.1 that calculates the factorial of a number. Figure 4.8 shows the detailed HIR instructions of all basic blocks. The first line of each block represents the BlockBegin instruction. Each block has a unique number for identification, which is followed by the range of bytecode indices that are covered by this block. Finally the predecessors (`pred`) and successors (`sux`) of the block are printed.

Each following line prints one HIR instruction of the block. An instruction has a bytecode index (column `bci`), a use count (`use`), a type and a unique id (`tid`). The use count specifies how often the instruction is referenced by other instructions. Instructions that must be executed in the original order of the bytecodes are marked as pinned, printed out as "`.`" at the beginning of the instruction line. Examples for pinned instructions are loads and stores of fields, because they might have data dependencies. Additionally, instructions that do not compute a result or that are used across block boundaries are pinned for technical reasons.

Because the HIR is typed, each instruction that computes a result also stores the type of the result. The example contains only integer operations, represented by the flag "`i`". Instructions that compute no result, e.g. jumps, have no assigned type. The type is followed by the unique id of each instruction. The first block `B4` is automatically introduced for the entry of the method and has no equivalent in the bytecodes. `B0` is the first block of the method and initializes the loop variable denoted as `i5` here.

```
B4 [0, 0] sux: B0
__bci__use__tid____instr_____________________________________
. 0    0     19    std entry B0

B0 [0, 1] pred: B4 sux: B3
__bci__use__tid____instr_____________________________________
. 0    1    i5     1
. 1    0     6     goto B3

B3 [2, 3] pred: B0 B2 sux: B1 B2
Locals:
   0: i7 [ i0 i13 ]
   1: i8 [ i5 i11 ]
__bci__use__tid____instr_____________________________________
  3    1    i9     0
. 3    0     10    if i7 <= i9 then B1 else B2

B2 [6, 14] pred: B3 sux: B3
__bci__use__tid____instr_____________________________________
. 8    1    i11    i8 * i7
  11   1    i12    1
. 12   1    i13    i7 - i12
. 14   0     14    goto B3 (safepoint)

B1 [17, 18] pred: B3
__bci__use__tid____instr_____________________________________
. 18   0    i15    ireturn i8
```

Figure 4.8: Compilation example—high-level intermediate representation (HIR)

The block B3 has two predecessors, so phi functions are created for all local variables. They are printed in the section Locals before other instructions of the block. The phi function for the local variable 0 (named "n" in the Java source code) has the id i7 and the two operands i0 and i13. The instruction i0 represents the first parameter of the method (not printed explicitly), and the instruction i13 is computed in block B2, which is a predecessor of B3. The instruction i7 gets the value of i0 when B0 was executed before (i.e. for the first iteration of the loop) and the value i13 when B2 was executed before (i.e. for all other iterations of the loop). The phi function for the local variable 1 (named "p" in the Java source code) has the id i8 and gets the value of i5 or i11, depending on the predecessor.

4.5 HIR Generation

To build the HIR, the bytecodes of the method are processed twice. The first, very fast pass identifies the boundaries of basic blocks and constructs the control flow graph only. The second pass then fills the blocks with the HIR instructions.

4.5.1 Identifying Basic Blocks

The first pass, implemented in the class BlockListBuilder, iterates over the bytecodes from the beginning to the end to find the boundaries of all basic blocks. All blocks (represented by BlockBegin instructions) are created and collected in a list, but they are left empty. Additionally, predecessor information for each block is stored.

When all blocks are identified, the class CFGMaker constructs the basic control flow of the method. Blocks are linked together with successor and predecessor edges. This control flow is used to identify and mark loop headers, i.e. blocks that are reachable by a backward branch. When the complete HIR is constructed later, these blocks need a special treatment. Then a visiting order for all blocks is calculated by assigning a number to each block. It is called depth_first_number, although it is not strictly a depth-first numbering of all blocks. A block is not appended to the visiting order until all predecessors are appended. Backward branches are ignored because otherwise loops could not be processed.

Only the loop header information and the visiting order are saved, the control flow is discarded before the blocks are filled with instructions. This allows optimizations of the control flow, because edges between blocks or even blocks themselves may be unnecessary due to never-taken branches. So the control flow of the final HIR can be slightly different as the control flow computed in this step.

4.5.2 Filling Blocks with Instructions

The class GraphMaker fills the basic blocks with HIR instructions by performing an abstract interpretation of the bytecodes. Basically, for each bytecode an HIR instruction is created and appended to the list. The stack-based bytecodes are transformed to the register-based HIR using the state array: The effect of bytecodes on the operand stack and the local variables is simulated.

The following example in Figure 4.9 shows the generation of the HIR for the computations in block B2 of the factorial example. The left side shows the bytecodes as presented in Figure 4.3 on page 27. The right side shows the HIR instructions like in Figure 4.8. In between, the simulated state of the local variables and the operand stack is illustrated. Elements that are changed by the current bytecode are marked as bold.

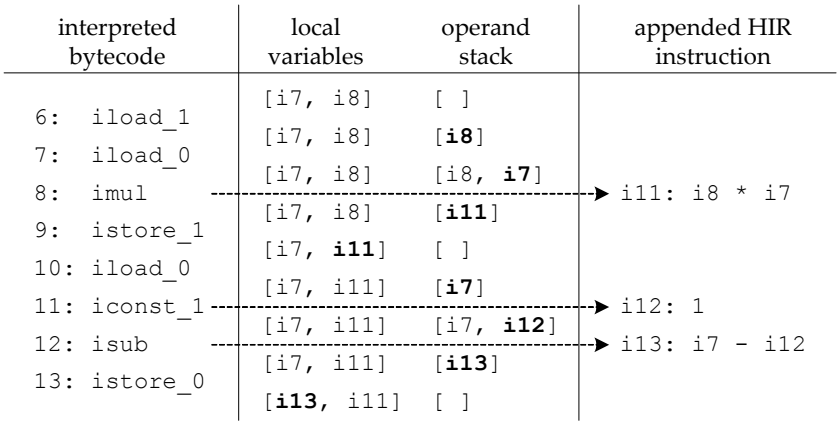


Figure 4.9: Construction of the HIR

At the beginning of the block, the two local variables contain the instructions i7 and i8; the operand stack is initially empty. The iload bytecode loads a local variable to the operand stack. No HIR instruction is necessary, only the state of the operand stack is modified. The iconst bytecode is represented by an HIR instruction of the class Constant that is appended to the HIR and pushed onto the operand stack. For the arithmetic operations, two parameters are popped from the operand stack, and a new HIR instruction of class ArithmeticOp is appended and pushed onto the stack. The istore bytecode pops the result from the operand stack and changes the state of the local variables.

4.6 Optimizations

The simple structure of the HIR allows the easy implementation of global optimizations, which are applied both during and after the construction of the HIR. Theoretically, all optimizations developed for traditional compilers could be applied, but most of them require the analysis of the data flow and are too time-consuming for a just-in-time compiler—even if they are considerably simplified by the SSA form of the intermediate representation. So the compiler implements only simple and fast, but nevertheless effective optimizations.

4.6.1 Canonical Instructions

Before an HIR instruction is appended to the instruction list, the class Canonicalizer tries to simplify the instruction. Especially instructions that involve constants are processed. If both operands of an arithmetic or logical instruction are constants, then constant folding can be applied. The whole calculation is then replaced by a new constant. Also, instructions with one argument being the constant 0 or 1 can often be optimized.

If the condition of a branch is proven to be always true or false, the branch can be replaced by an unconditional jump to the respective block. This can lead to blocks that are never reached and therefore discarded.

4.6.2 Inlining

Calling a method is an expensive operation because parameters must be passed on the stack and a stack frame must be maintained for each method. For short methods, e.g. accessors that just return the value of a field, calling a method can consume more time than the actual execution of the method. Therefore, short methods are inlined into their callers: The call to the method is replaced by a copy of its instructions. Inlining has a high impact on the execution time, but can be applied only if the called method is unambiguously known.

This limitation complicates inlining for virtual methods: The actually called target depends on the dynamic type of the object and is therefore not known during compilation. If a call dispatches to different targets at runtime, it is called polymorphic; if the target is always the same, it is called monomorphic. Because of the semantics of the Java programming language, most method calls are virtual. However, measurements show that most calls are nevertheless monomorphic. Such monomorphic targets can be identified by analyzing the hierarchy of all loaded classes. It is then possible to inline methods even for virtual calls.

4.6.3 Common Subexpression Elimination

Common subexpression elimination (CSE) removes redundant computations of equal subexpressions. If the same instruction with the same operands is contained twice in the instruction stream, all references to the second instruction can be replaced by references to the first one. CSE is implemented via a hash table containing all computations processed previously. Before an instruction is appended, the compiler checks if an equal instruction is already present in the hash table. If an old instruction is found, it is used instead of the new instruction.

Currently, only a local CSE is performed that finds subexpressions only inside a single basic block. A global CSE optimizing the whole method is simplified by the static single assignment form because it is guaranteed that each value is defined only once. However, it is not implemented yet.

4.6.4 Null Check Elimination

Because Java is a safe programming language, null pointer exceptions must be thrown when `null` objects are accessed. The virtual machine must check each access of an object. Two different kinds of null checks are used:

- Most checks are implicitly performed by the processor: When a memory location near to 0 is accessed, the processor raises an internal exception which is processed by the virtual machine. No additional machine instructions are necessary for such checks.

- Some null checks must be performed explicitly because of the Java semantics. For example, a Java null pointer exception must be thrown if a method of a null object is called, even if the method executes no code. So an explicit null check is necessary whenever a method is inlined.

The null check elimination tries to eliminate explicit null checks or replace them with implicit checks. If the input argument of a null check can be proven to be not null, i.e. when it is guaranteed that a null check on the same object has been executed before, then the null check can be eliminated. This optimization succeeds in eliminating most explicit null checks.

4.6.5 Control Flow Optimizations

The *Conditional Expression Elimination* searches the control flow graph for conditional expressions. These are conditional branches that load one of two values depending on a condition and then continue with the same block. The `If` instruction of the branch is replaced by a special `IfOp` instruction that has both values as input parameters. The back end can generate more efficient code for conditional move instructions where no branches are necessary.

All optimizations mentioned in this chapter can lead to blocks that are only connected by a single edge. If a block ends with an unconditional jump to a successor, and this successor has the block as its only predecessor, the two blocks are merged to one larger block. This optimization reduces the number of blocks that must be processed by the back end later on.

4.7 Low-Level Intermediate Representation

The low-level intermediate representation (LIR) is conceptually very similar to native code, but allows platform-independent algorithms that would be difficult to implement directly on native code. The instructions and operands are shared between all platforms, only the generation and some small other parts are platform-dependent. The main optimization that is applied on the LIR is the register allocation. The LIR is more suitable for register allocation than the HIR because all operands requiring a register in machine code are explicitly visible in the LIR.

In the compiler, all classes related to the LIR start also with the prefix `LIR`. In contrast to the HIR, the operations of the LIR use explicit operands, so the operations are not linked together directly. If an operation uses the result of another operation as an input value, then the result operand and the input operand refer to the same register or memory address.

All LIR operations of a basic block are stored in an array list. This allows fast iterations over all operations. Only the control flow is shared with the HIR: The list of LIR operations for a block is stored in a field in the `BlockBegin` instruction of the HIR, avoiding the duplication of the control flow nodes that would be necessary otherwise.

4.7.1 Operands

Operands of the LIR must be capable of modeling all operands and addressing modes available in the target architecture. Therefore, the following kinds of operands can be distinguished:

- *Virtual registers* are placeholders for operands whose final location is not known yet. 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. Each virtual register has a unique index. The total number of virtual registers is unlimited.
- *Physical registers* are a direct representation of the target architecture's general-purpose and floating point registers. Their number and data type is fixed.
- *Addresses* are used to reference arbitrary memory locations, e.g. fields of objects and arrays. On Intel processors, an address consists of a base register, an index register, a scale factor and a displacement. The base and index registers can be either virtual or fixed registers.
- *Stack slots* are a special form of addresses that refer to the stack frame of the current method. Although it would be possible to replace stack slots with addresses, stack slots are more convenient for accessing the stack frame because a single index is used instead of an address with a base register and a displacement. The actual location of the stack slot is determined during code generation. The first slots are mapped on the parameter area of the calling method and therefore represent the incoming method parameters, the remaining slots are mapped on the spill area of the current method.
- *Constants* of any type are allowed in the LIR, even if they are not directly supported as immediate values in the target architecture. On Intel, integer constants can be used as immediate operands with nearly all instructions, while floating point constants are stored in a reserved area and referred to by their address later on.

Operands of the LIR are typed to distinguish between integer, object and floating point operands and to determine the size of the operands in memory. The type of an operation is implicitly fixed by the type of its operands.

Because of the high number of LIR operands, the efficient handling is crucial for the performance of the compiler. Consequently, the compiler uses a mixture of objects and direct representation for encoding: Physical registers, virtual registers and stack slots are encoded as a bit field. Addresses and constants are represented as objects because they are too large to be encoded in a single integer value. To allow a consistent use of both kinds of operands, the bit field is directly encoded as a mock pointer. The least significant bit of a mock pointer is always set to 1 to distinguish mock pointers from real pointers, while this bit is always cleared in a real pointer because of the 4-byte alignment of objects. This trick allows the uniform representation of LIR operands as pointers without the need to allocate space for frequently used register and stack operands.

4.7.2 Instruction Set

LIR operations are represented by a class hierarchy with the base class LIR_Op. The result operand is located in the base class because most operations return a value. The class hierarchy is used to group operations with an equal number of operands. Most operations use the classes LIR_Op0, LIR_Op1 and LIR_Op2 that specify a generic operation with zero, one or two input operands, respectively. Figure 4.10 shows the class hierarchy where some classes and fields are omitted for a better readability.

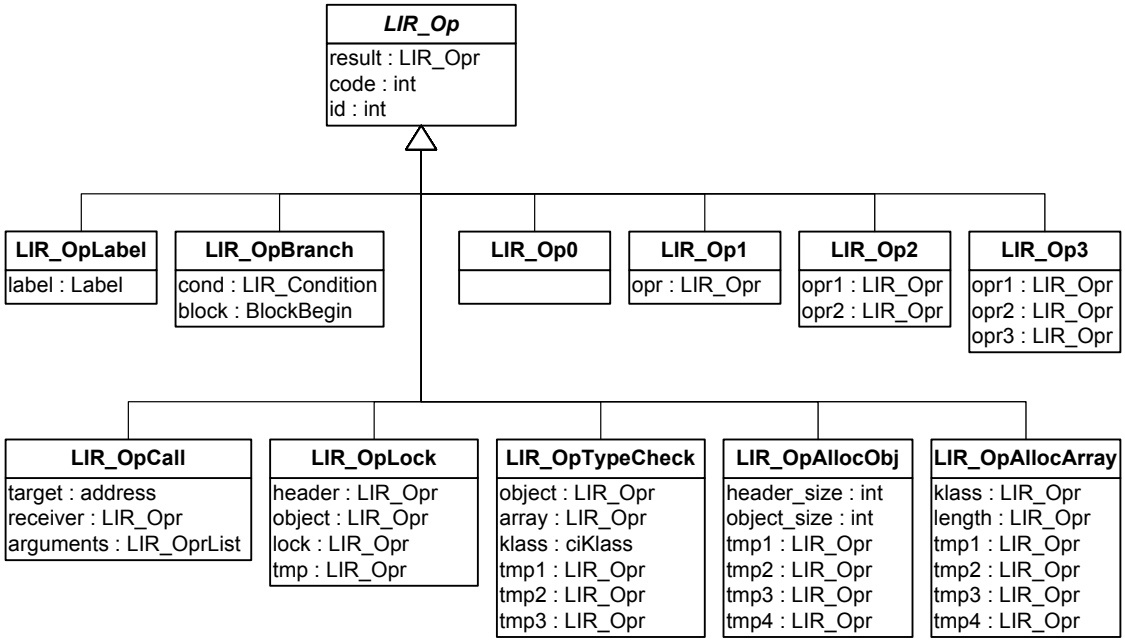


Figure 4.10: Class hierarchy for LIR operations

The classes of the bottom row in Figure 4.10 are used for special higher-level operations. They are later expanded to multiple native instructions. The temporary operands specify registers that are only used for computations inside the operation, but are not valid before or after the operation. The register allocator guarantees that it is save for the operation to overwrite these registers.

Each LIR operation is identified by a unique code, stored in the field code, which is independent from the class hierarchy because it is available in the base class. The following enumeration lists some important codes:

- lir_label is the first operation of each basic block. It specifies a label that is used as the target for jumps to this block.
- lir_std_entry is the very first operation of a method. It is responsible for creating the method's stack frame.
- lir_return returns the control flow to the calling method. The return value is the operand of lir_return.
- lir_move is the most often used operation. It performs a general move between two registers, between the memory and a register or between a constant and a register.

- lir_cmp and lir_branch are used together for conditional branches. Unconditional jumps are represented by a single lir_branch.
- The arithmetic and logical instructions like lir_add and lir_sub use the three-operand form with two input operands and the result operand.
- Various call operations are available for static and virtual calls. The parameters of the called method must be already on the stack, so preceding moves are necessary to store them in the appropriate stack slot.
- Operations for type checks, synchronization and allocation of objects and arrays are modeled as a single LIR operation working on a higher level. They represent the corresponding HIR instructions.

4.7.3 Example

This chapter shows the LIR for the example that calculates the factorial of a number presented in Chapter 4.2.1. Figure 4.11 on the next page shows the detailed LIR operations of all blocks. The first line of each block represents the BlockBegin instruction that is equal to the HIR in Figure 4.8 on page 36. The following lines represent the LIR operations of the block. Each operation has a unique number (column `id`) that is computed during register allocation (see Chapter 5.4 on page 57 for more details).

The list of operands must be read from left to right, so the result operand is always the rightmost operand. The different kinds of operands are printed in the following syntax:

- `[R40|I]` refers to the virtual register with the index 40 (numbers below 40 are reserved). It stores an operand of type integer (represented by "`I`").
- `[ecx|L]` refers to the physical register `ecx` that contains an operand of type object (represented by "`L`").
- `[stack:0|I]` refers to the stack slot with the index 0.
- `[int:1|I]` represents the integer constant 1.
- For an address, the base register, index register, scale factor and displacement are printed.

The first operand of a branch operation specifies the branch condition. The condition `[LE]` ("less or equal") specifies that the branch is taken only if the left operand was less than or equal to the right operand in the preceding compare operation. The condition `[AL]` ("always") specifies an unconditional jump. The second operand of a branch is the target block, identified by its block id.

Some LIR operations must store the state of the local variables and the operand stack as seen by the Java bytecodes. This is necessary to allow the generation of debug information for deoptimization (see Chapter 3.2.2 on page 19). Therefore, the state array of the HIR is propagated to the LIR and later also to the machine code. Examples for such operations are safepoints—explicit positions where the garbage collector is allowed to run—and method calls. In the example, the operation with the id 38 is a safepoint because the succeeding branch is the backward branch of a loop. The tag `[bci:14]` specifies that this operation stores the state array, allowing the reconstruction of the local variables at the bytecode index 14.

```
B4 [0, 0] sux: B0
__id__Operation_____________________________________________
   0  label [label:0x31da99c]
   2  std_entry [ecx|L]
   4  move [stack:0|I] [R40|I]
   6  branch [AL] [B0]

B0 [0, 1] pred: B4 sux: B3
__id__Operation_____________________________________________
   8  label [label:0x31bb9d4]
  10  move [R40|I] [R42|I]
  12  move [int:1|I] [R43|I]
  14  branch [AL] [B3]

B3 [2, 3] pred: B0 B2 sux: B1 B2
__id__Operation_____________________________________________
  16  label [label:0x31da264]
  18  cmp [R42|I] [int:0|I]
  20  branch [LE] [B1]
  22  branch [AL] [B2]

B2 [6, 14] pred: B3 sux: B3
__id__Operation_____________________________________________
  24  label [label:0x31da17c]
  26  move [R43|I] [R44|I]
  28  mul  [R44|I] [R42|I] [R44|I]
  30  move [R42|I] [R45|I]
  32  sub  [R45|I] [int:1|I] [R45|I]
  34  move [R44|I] [R43|I]
  36  move [R45|I] [R42|I]
  38  safepoint [bci:14]
  40  branch [AL] [B3]

B1 [17, 18] pred: B3
__id__Operation_____________________________________________
  42  label [label:0x31da094]
  44  move [R43|I] [eax|I]
  46  return [eax|I]
```

Figure 4.11: Compilation example—low-level intermediate representation (LIR)

4.8 LIR Generation

The LIR is generated by visiting all instructions of the HIR. Each basic block is processed independently. Inside a basic block, all pinned instructions are handled in their original order. Instructions that are not pinned are processed recursively: If the currently processed instruction uses another instruction as an input operand that has not been handled yet, this instruction is processed before. For each HIR instruction, an arbitrary number of LIR operations can be created. The LIR uses an unlimited number of virtual registers for the operands.

4.8.1 Phi Functions

Phi functions necessary for the SSA form do not have a direct representation in any target architecture, so they must be resolved by move operations. This conversion is done during the LIR generation. The LIR does not contain phi functions and is therefore not in SSA form. Phi functions are replaced by moves in the predecessor blocks: Each phi function has

a unique virtual register assigned that is used as the target of move-operations in the predecessors.

Figure 4.12 shows a simple example of a phi function and the appropriate moves necessary for resolution. The virtual register `[R1]` representing the instruction `n3` is assigned in both predecessors: In the first predecessor, the first operand `n1` of the phi function (the constant 10) is used. In the second predecessor, the second operand `n2` (the constant 20) is used.

a) unresolved phi function b) resolved phi function

Figure 4.12: Resolving phi functions with moves

4.8.2 Two-Operand Form

The LIR uses the three-operand form for arithmetic and logical operations: An operation contains the left and right input operand and the result operand. However, the IA-32 architecture uses the two-operand form for all instructions, where the left input operand is equal to the result operand. It is not possible to generate machine code from a LIR operation with three different operands. Therefore, the LIR is constrained when generating code for the IA-32 architecture: The left input operand is always equal to the result operand. Before each operation, a move from the left operand to the result operand is inserted. For example, the operation

```
add  [R1] [R2] [R3]
```

that would add `[R1]` and `[R2]` and store the result in `[R3]` is replaced by

```
move [R1] [R3]
add  [R3] [R2] [R3]
```

This sequence first copies the left input operand into the result operand and adds the right operand to the result. Both operations can be converted directly to machine instructions of the IA-32 architecture. The insertion of moves is not necessary when generating code for the Sparc architecture because the three-operand form is also used by its native code.

4.8.3 Fixed Registers

Most IA-32 instructions can operate on all general-purpose registers. But some instructions are constrained to fixed registers, like instructions for divisions and shifts. To support such instructions, operands referring to physical registers are uses instead of virtual registers in the LIR. Because fixed registers restrict the register allocator, their live ranges are made as short as possible: move operations from and to fixed registers are inserted immediately before and after operations requiring them.

For example, the shift count for shift operations must be always in the fixed register [ecx]. Therefore, the operation

```
shl [R1] [R2] [R3]
```

that would shift [R1] to the left by [R2] bits and store the result in [R3], is replaced by

```
move [R1] [R3]
move [R2] [ecx]
shl  [R3] [ecx] [R3]
```

The first move is necessary because of the two-operand form. The second move copies the virtual register [R2] to the fixed register [ecx]. The register allocator can assign an arbitrary register to [R2] without considering the constraints of the IA-32 architecture. Another common occurrence of a fixed register is the return operation of a method: When a method returns an integer value, the calling convention requires the result in the register [eax]. When the virtual register [R1] should be returned for example, a move to the fixed register [eax] is necessary before the return operation:

```
move   [R1] [eax]
return [eax]
```

The LIR operations for returning a floating point value are similar, except that the first floating point register is used instead of [eax].

4.9 Register Allocation

The register allocator is responsible for replacing all virtual registers with physical registers. After register allocation, the LIR contains only operations with operands that can be mapped directly to machine instructions. The register allocator using the linear scan algorithm is the main result of this master thesis, so the algorithm is described in all details in Chapter 5. In principle, the following steps are performed:

- First, the basic blocks are sorted into a linear order, i.e. the control flow graph is flattened to a list. All LIR operations are numbered increasingly using this block order.
- For each virtual register, the lifetime interval is calculated. A virtual register is live between the operation that defines the value and the operations that use it. The lifetime interval can contain holes where the virtual register does not contain a useful value.
- For the actual register allocation, the list of all intervals—sorted by increasing start position—is traversed. Each interval gets a physical register assigned that is not used by a simultaneously live interval. If more intervals are live than registers are available, then intervals are split and spilled to the stack.
- After register allocation, all operands referring to virtual registers are replaced with the physical registers or stack slots that were assigned to the according intervals.

The register allocator must be capable of handling the fixed registers emitted during the generation of the LIR. Additionally, some LIR operations such as method calls destroy all registers. The register allocator must guarantee that no register is in use at these positions.

Because all constraints of the target architecture are already reflected in the LIR, the register allocator needs not reserve registers as *scratch registers*. Many other compilers exclude one or even more registers from the normal register allocation and use these registers later during code generation. For example, spilled values that are required in a register are loaded to a scratch register before the value is needed. Because the register demands are properly modeled by this version of the linear scan algorithm, no scratch register is needed and all registers are available for allocation. This is especially valuable for the IA-32 architecture since only six general-purpose registers are available. Reserving one of these registers as a scratch register would lead to a significantly higher register pressure.

4.10 Code Generation

Generating machine code from the LIR is straightforward: Because all platform-dependent issues are already represented in the LIR and the register allocator guarantees correct operands, most LIR operations result in one or two native instructions. All arithmetic and logical operations, moves and branches can be converted without using further algorithms.

Only the higher level LIR operations for type checks, synchronization and object allocation are replaced by longer patterns of machine instructions. These operations use the *Focus on the Common Case* principle: The implementation is split into a common and an uncommon case. The code for the common case, which is executed frequently, is inlined directly into the normal code. The code for the uncommon case, which is used if the common case fails, is located outside of the method's regular code and usually calls a function of the runtime environment.

The common case can be executed very fast because no runtime calls are necessary. All machine instructions and therefore all used registers are known, so other values can be kept in registers that are not affected by the common case. For the uncommon case, all registers must be saved because they are destroyed by the runtime call. This extra cost for saving registers in the uncommon case is justified by a faster execution of the common case.

For example, objects can be allocated very fast in nearly all cases. All new objects are allocated from a reserved memory area that is managed by the garbage collector, so the allocation of a new object requires only the following instructions in the common case:

```
obj = top
top = top + size
if (top > limit) then goto slowcase
```

For the allocation, chunks of the reserved memory are returned until the limit of the reserved memory is reached. If this happens, then the uncommon case is called that invokes the garbage collector. Because the garbage collector needs much more time compared with the three machine instructions, the additional overhead for saving and restoring all registers before and after the call is not a significant delay anyway.

4.11 Meta Data

The main output of the compiler is the native code that can be executed directly by the processor. Because the native code runs in the managed environment of the virtual machine, the virtual machine needs some meta data for its work:

- *Debug information* contains a mapping from compiled code back to the state of the interpreter. It is used for deoptimization, when the execution of a compiled method is transferred back to the interpreter. Debug information is emitted for all machine instructions where deoptimization might be possible, e.g. for all method calls and all instructions that are allowed to throw an exception. For each program counter of such machine instructions, the actual location, i.e. register or spill slot, of all local variables and operand stack items is stored.
- *Oop maps* specify the exact location of all oops (ordinary object pointers; pointers to objects that are managed by the garbage collector) for all program counters where garbage collection can happen. During garbage collection, all these locations are treated as root pointers into the heap. If the garbage collector moves an object, then these locations must be updated as well.
- Exception handling is implemented with tables specifying all possible exception handler entry points for a given program counter range. When an exception is thrown at runtime, the correct exception handler is searched using the dynamic type of the exception.

Debug information and oop maps are created during register allocation since the necessary information is contained in the lifetime intervals. Before register allocation, the exact locations are not yet known, and after register allocation the information is no longer available.

Chapter 5

Linear Scan Register Allocation

This chapter presents the linear scan algorithm used for register allocation in detail. The algorithm is presented in pseudo-code and illustrated with examples. First, the basic blocks are ordered in an optimal linear order, using the loop depth of the blocks. Then the lifetime intervals, consisting of multiple ranges and use positions, are constructed. For the actual register allocation, the intervals are walked and each interval gets a register assigned. If no more free registers are available, intervals are split and spilled. In the last step, the allocated registers are written back to the LIR.

The linear scan algorithm implemented for this master thesis in principle follows the algorithm presented by O. Traub et al. in [Traub98], although many details are implemented differently. It adheres to the following basic principles:

- The basic blocks are sorted into a linear order for allocation. The control flow graph is hidden during the allocation. This allows a linear algorithm to work on a non linear control flow graph.
- No scratch register is reserved by the allocator, so all registers are available for allocation. A scratch register is not needed for the code generation since the allocator guarantees that a register is available for all operations that cannot operate on memory operands.
- The lifetime of virtual registers is represented by intervals with multiple ranges. Intervals can have holes between ranges, called lifetime holes, where a virtual register does not contain a useful value.
- The SSA form of the HIR leads to many short intervals, where each interval is assigned only once. Only intervals for the resolving moves of phi functions have multiple definitions. These intervals also have large holes.
- The register allocator assigns a register to each interval in a linear pass over all intervals such that no intersecting intervals have the same register assigned. Intervals that do not intersect can get the same register because they are independent from each other.
- If no register is available for the entire lifetime of an interval, the register available for the longest time is selected. The interval is split at the position where the

register is no longer available, and the decision what to do with the split part is postponed.

- If no register is available because all registers are already blocked by other intervals, then one or more intervals must be spilled to the stack. In this case, the interval is split at this position and a move from the register to the stack is inserted into the LIR.
- The splitting algorithm is very flexible. An interval can be split to change its location everywhere. Because the linear block order cannot model the control flow graph, a resolution pass is necessary that inserts moves at control flow edges.
- If an operation requires an operand to be in a register, a use position is registered. If an interval is split and spilled, it is reloaded to a register at least before the next use position that must be in a register.
- The selection strategy for spilling is not based on the absolute weight of an interval, but on the relative distance to the next use position: In general, the interval with the next use position furthest away is spilled.
- Fixed intervals model operations that require operands in fixed registers. One fixed interval per physical register models the ranges where the register is not available for normal allocation.
- Fixed intervals are also used to block all register at call operations. Because a call to another method destroys all registers, a short range is added to all fixed intervals at the position of the call. This forces a spilling of all non-fixed intervals that are live at a call without further special handling of calls during allocation.
- The rewriting of the LIR where all virtual registers are replaced with physical registers and stack slots is done in a separate pass after all intervals were processed.

5.1 Class Overview

The class diagram in Figure 5.1 shows the structure and dependencies of the classes used during register allocation. It contains the classes together with their most important fields and some methods that are used later on in the algorithms. The actual implementation in the compiler is sometimes slightly different due to optimizations for a higher compile speed. For example, some lists are implemented as linked lists to allow fast insertion and removal of elements. The necessary next-pointers are omitted from the class diagram. Dashed lines in the class diagram represent dependencies between classes where one class is used locally in some methods of the other.

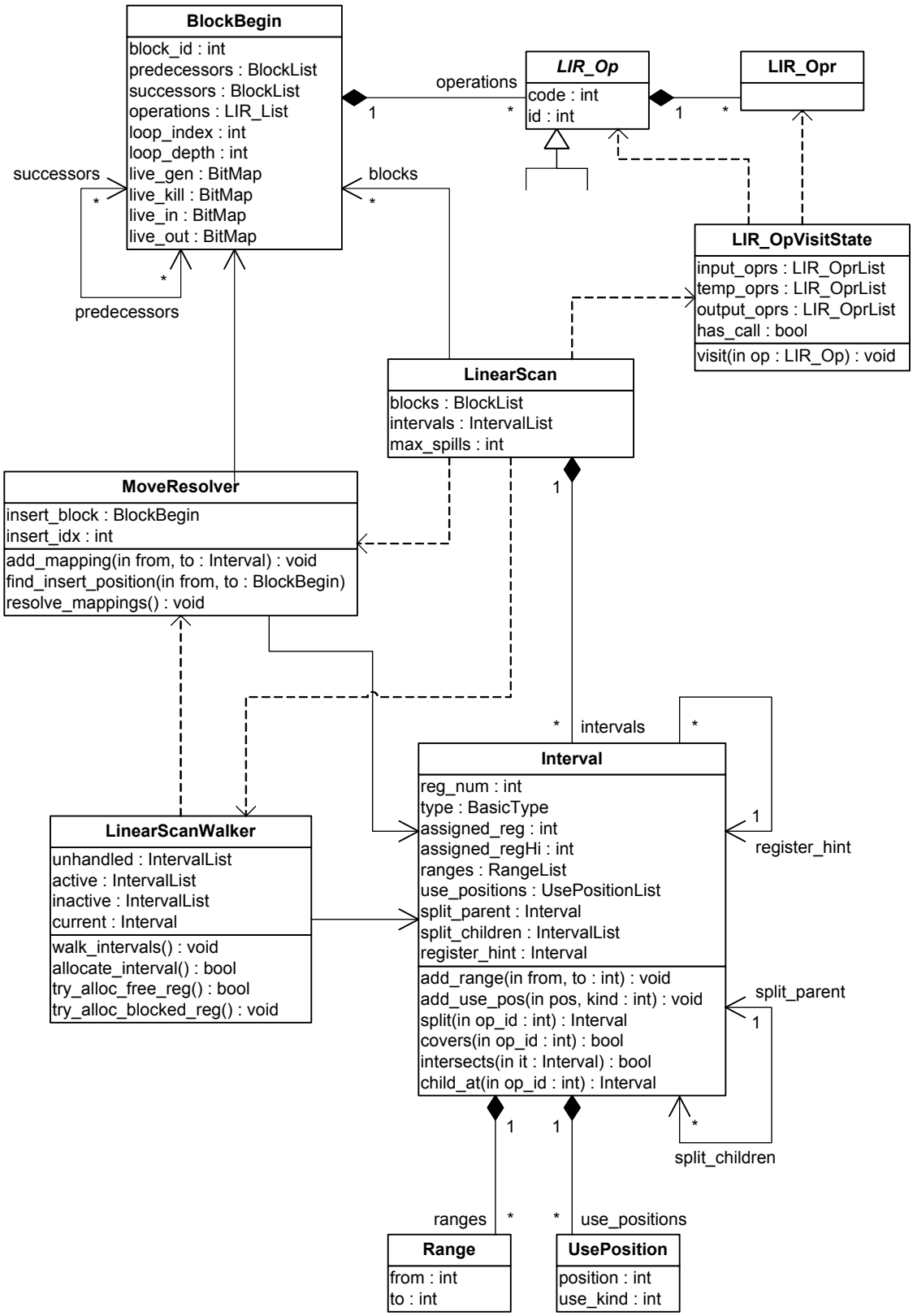


Figure 5.1: Classes uses during linear scan register allocation

5.2 Basic Algorithm

The important parts of the algorithms are presented in pseudo-code where formal instructions and informal text is mixed. Variables are always written in italic, keywords for if- and loop-statements are written in bold. The algorithms are presented on a high level with unimportant details omitted. Algorithm 5.1 shows the basic steps of the linear scan algorithm. Each method that is called herein represents a phase of the algorithm and is described later in its own chapter.

```
LINEAR_SCAN
    // order blocks and operations (including loop detection)
    COMPUTE_BLOCK_ORDER
    NUMBER_OPERATIONS

    // create intervals with live ranges
    COMPUTE_LOCAL_LIVE_SETS
    COMPUTE_GLOBAL_LIVE_SETS
    BUILD_INTERVALS

    // allocate registers
    WALK_INTERVALS
    RESOLVE_DATA_FLOW

    // replace virtual registers with physical registers
    ASSIGN_REG_NUM

    // special handling for the Intel FPU stack
    ALLOCATE_FPU_STACK
```

Algorithm 5.1: Steps of linear scan

5.3 Block Order

The linear scan algorithm does not operate on a structured control flow graph, but on a linear list of blocks. Most parts are not sensitive to the actual order of the blocks, so any linear order could be used theoretically. But the order has a high impact on the quality and speed of the allocation: A good block order leads to short intervals with few holes and reduces the number of intervals that must be split. Additionally, the same block order is used for the generation of native code, so a good block order reduces the number of unconditional jumps necessary in native code. The algorithm for ordering blocks presented in this chapter has the following characteristics:

- Two blocks linked by a jump are emitted consecutively if possible. This reduces the number of unconditional jumps because they are not necessary between consecutive blocks.
- Blocks located close to each other, such as the if- and the else-branch of an if-statement, are also arranged nearby in the block order. This is ensured by emitting a block not before all predecessors of this block except backward branches are emitted.
- Blocks that are part of a loop are executed far more often than blocks of a sequential control flow, so their order is important. The algorithm guarantees that all blocks of

a loop are emitted consecutively, without blocks in between that do not belong to the loop. This ensures a good locality of the frequently executed loop blocks and helps the allocator to assign registers to all intervals used in the loop.

- Blocks that are known to be executed rarely, such as blocks for exception handling, are emitted as late as possible and placed at the end of the method. This increases the locality of frequently executed blocks.

5.3.1 Loop Detection

The loop detection algorithm is integrated in the block ordering because the loop depth is used for ordering blocks. A loop is identified by its loop header block. This block is always the first block of the loop and the target of all backward branches. Because the bytecodes also allow unstructured programming with arbitrary jumps, it is possible to find loops with multiple header blocks. Such loops are ignored because they are very rare. It is guaranteed that the remaining loops have a unique header block. In contrast, many loops have multiple loop end blocks, which is handled correctly by the algorithm.

The following two numbers are computed for each block. They are stored as fields of the BlockBegin instruction because they are also used later during register allocation:

- loop_index stores the unique number of the innermost loop in which this block is contained.
- loop_depth stores the loop nesting level of this block. The higher this number, the more important is this block.

The algorithm needs several iterations over the control flow graph. At first, the graph is iterated forward starting with the first block of the method and using the successors of a block. When a block is reached for the first time, it is marked as *visited*. As long as successors of a block are processed, the block is additionally marked as *active*. The visited-flag is not cleared during the iteration, whereas the active-flag is cleared after all successors were processed.

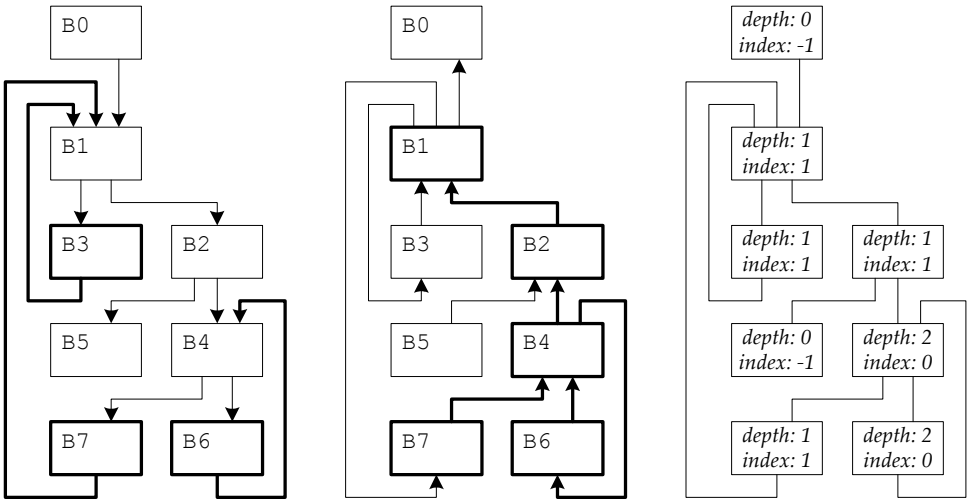
When the iteration reaches a block where the active-flag is already set, a loop is detected: The block with the active-flag set is the loop header, the previously processed block is the loop end. The edge between these two blocks is marked as a backward branch, and the loop end block is added to a list that collects all loop end blocks. Each loop header is assigned a unique loop index. Additionally, this iteration computes the number of incoming forward and backward branches for all blocks. They are needed later for computing the block order. The iteration stops when all blocks are marked as visited.

In the next step, the list of loop end blocks is processed. Each loop end block is the starting point for a backward iteration of the control flow graph using the predecessors of a block. The iteration stops when the loop header block belonging to the loop end is reached. All blocks that are reached during the iteration are contained in the loop. A block is contained in multiple nested loops if it is reachable from multiple loop end blocks with different loop indices. The output of this step is stored in a two-dimensional bit set where the first dimension is the block id and the second dimension is the loop index. A block is contained in a loop if the bit for this block id and loop index is set.

This bit set is used in the third step to calculate the loop depth and the index of the innermost loop: The loop depth of a block is the number of bits that are set for the according block id. The final loop index is the index of the lowest bit that is set, because a nested loop has a lower loop index than its surrounding loops (this is guaranteed by the first step that assigns the loop indices).

5.3.2 Example

The following example illustrates the loop detection algorithm. Figure 5.2 shows a complicated control flow graph with eight basic blocks and two nested loops. The outer loop has two loop end blocks `B3` and `B7` because both blocks have a backward branch to the same loop header `B1`. Block `B0` is the start block; `B5` ends with a return statement and has no successors. The numbering of the blocks is arbitrary.



a) CFG with successors: Loop end blocks and backward branches are marked

b) CFG with predecessors: Blocks reachable from `B7` with predecessors are marked

c) CFG with final loop depth and loop index for each block

Figure 5.2: Example of loop detection

The first step of the algorithm collects the loop end blocks. Three loop end blocks are found: `B3` and `B7` are end blocks of the loop with the index 1, starting at `B1`. `B6` is the only end of the loop with the index 0, starting at `B4`. This loop has a lower index than the first loop because it is nested in the first loop. The loop end blocks with their according backward branches are marked in Figure 5.2 a). Each loop end block has a single successor edge to its loop header. Table 5.1 summarizes the loop end table collected in the first step of the algorithm.

Loop End	*Loop Header*	*Loop Index*
`B3`	`B1`	1
`B6`	`B4`	0
`B7`	`B1`	1

Table 5.1: Loop end blocks of example

The next step marks all blocks of the loops. Figure 5.2 b) shows all blocks that are reachable from the loop end block B7 when the control flow graph is iterated backward using the predecessors: B4 is a direct predecessor of B7, and B1, B2 and B6 are indirect predecessors. The predecessors of B1 are not processed because B1 is the loop header block of B7, so B0 and B3 are not marked as loop blocks. B0 actually is no loop block and remains unmarked, whereas B3 is marked later because it is also present in the list of loop end blocks. B5 is not reachable from any loop end block using the predecessors, so it is not contained in any loop. Table 5.2 shows the two-dimensional bit set of loop blocks after all loop end blocks were processed.

		Block Id							
		B0	B1	B2	B3	B4	B5	B6	B7
Loop Index	0					x		x	
	1		x	x	x	x		x	x

Table 5.2: Two-dimensional bit set of blocks belonging to loops

The loop depth of a block is the number of bits that are set in its column of the table. The loop index of a block is the first row where a bit of the block is set. If no bit is set in a column, then the block is not contained in a loop, represented by a loop depth of 0 and a loop index of -1. Figure 5.2 c) shows the final loop depths and loop indices.

5.3.3 Compute Block Order

The algorithm for computing the block order uses a sorted work list of blocks to process the control flow graph. It is ordered by increasing weight of a block and managed stack-based, so the block with the highest weight is processed first. The most significant part of the weight is the loop depth of a block. If two blocks have the same loop depth, some other criteria are used, e.g. blocks of exception handlers or blocks that throw an exception are sorted down the list. Nevertheless, most blocks of a sequential control flow have the same weight. In this case, the work list behaves like a stack, so the block that was added last is removed first.

Algorithm 5.2 on the next page is used for computing the final block order. The list of blocks is stored in the field blocks of the class LinearScan. Whenever blocks are iterated later on during register allocation, this block order is used.

At the beginning, the first block of the method is added to the work list. Then, the work list is processed until it is empty. The first block with the highest weight is removed from the work list and appended to the final block order. All successors ready for processing are sorted into the work list. A block is ready for processing if all predecessors except backward branches are already present in the final block order. This is decided using the number of incoming forward branches initialized during the loop detection: whenever a successor is processed, its number of incoming forward branches is decremented. When the number reaches 0, all forward branches are processed.

```
COMPUTE_BLOCK_ORDER
    append first block of method to work_list

    while work_list is not empty do
        BlockBegin b = pick and remove first block from work_list
        append b to blocks

        for each successor sux of b do
            decrement sux.incoming_forward_branches
            if sux.incoming_forward_branches = 0 then
                sort sux into work_list
            end if
        end for
    end while
```

Algorithm 5.2: Compute block order

5.3.4 Example

In the example started in Chapter 5.3.2, all blocks have only one incoming forward branch and so they are ready for processing when they appear in a successor list the first time. In this example, the loop depth is used as the only component of the weight for simplicity. Figure 5.3 shows the first iterations of the loop in Algorithm 5.2. The left side represents the work list, the right side the final block order. The number printed beneath each block of the work list is the weight of the block.

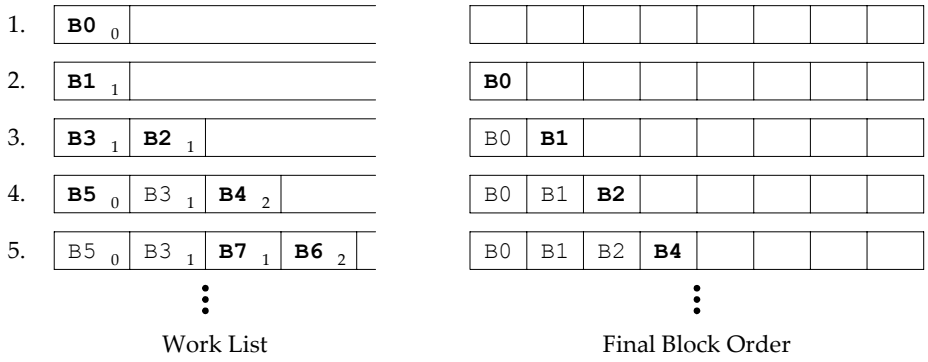


Figure 5.3: Example of computing block order

The following actions are executed:

1. The work list is initialized with `B0`
2. `B0` is processed and appended to the block order. The only successor `B1` is sorted into the work list.
3. `B1` is processed. The two successors `B2` and `B3` have the same weight, so their order is not important. Assume that `B2` becomes the top of the work list.
4. `B2` is processed. The first successor `B4` has the highest weight and is sorted to the top of the work list. The second successor `B5` has a lower weight than `B3` which is already in the work list, so `B5` is sorted in before `B3`.
5. `B4` is processed. `B6` has the highest weight and is sorted to the top. `B7` is sorted in after `B6`, but before `B3` because of the stack-based ordering of blocks with the same weight.

In the remaining steps, no new blocks are sorted into the work list because all successors are backward branches that are ignored. The top of the work list is just appended to the block order without sorting new blocks into the work list. Figure 5.4 shows the resulting final block order. The quality criteria mentioned at the beginning of the chapter are fulfilled, both loops are continuous. Although the return block `B5` was ready for processing during the loop, it is the last block in the linear order.

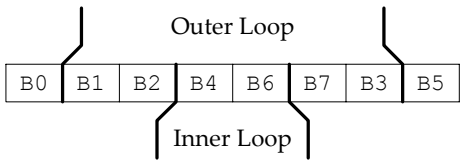


Figure 5.4: Final block order of example

5.4 Numbering of LIR Operations

The block ordering leads to a linear list of all blocks. This order is used to compute a linear order of all LIR operations. The field id of the class LIR_Op is used to store the number of each operation. The numbering, as shown in Algorithm 5.3, is a straightforward iteration of blocks and operations.

```
NUMBER_OPERATIONS
    int next_id = 0

    for each block b in blocks do
        for each operation op in b.operations do
            op.id = next_id
            next_id = next_id + 2
        end for
    end for
```

Algorithm 5.3: Numbering of LIR operations

The number for the next operation is always incremented by 2, so only even numbers are used. This simplifies many algorithms because there is always a free position between two operations where a new operation, e.g. a spill move, can be inserted. Only when more than one operation is inserted at a single position, the inserted operations must be ordered such that no register is overwritten.

5.5 Lifetime Intervals

Lifetime intervals are the main data structure used during register allocation. For each virtual register that is used in the LIR, one interval is constructed that represents the lifetime of the register. There is a strong conjunction between a virtual register and its lifetime interval. Because the same numbering schema is used, the virtual register with the number n corresponds to the interval with the number n when the intervals are created. When intervals are split later during register allocation, one virtual register can also correspond to multiple intervals.

A lifetime interval is represented by the class Interval. The class diagram in Figure 5.1 on page 51 contains the important fields and methods of this class:

- The field reg_num stores the number of the virtual register that this interval corresponds to.
- The type of an interval is needed to select the appropriate physical register set for the interval: Intervals of the types float and double require a floating point register to be assigned, whereas int and object need a general-purpose register. The type long needs two general-purpose register because the IA-32 architecture provides only 32-bit integer registers.
- The register assigned to the interval during register allocation is stored in the field assigned_reg. The second register for intervals of the type long is stored in assigned_regHi.

5.5.1 Ranges

One lifetime interval may consist of multiple *live ranges*. They model the parts of the method where a virtual register contains a meaningful value that is used later on. In the simplest case there is only a single live range that starts at the operation that defines the register and ends at the last operation that uses the register. More complicated intervals consist of multiple ranges, so each interval stores a list of ranges.

Every live range has two fields from and to, denoting the id numbers of the first and last LIR operation covered by the range. The field from is meant to be inclusive, while to is meant to be exclusive. For example, the range [4, 8[starts at operation 4 and ends at operation 8 where it is not live any more. The ranges [4, 8[and [8, 12[do not intersect because the id 8 is not covered by the first range, whereas [4, 8[and [7, 12[intersect because they have the id 7 in common.

The register allocator later assigns registers to intervals so that intervals with intersecting lifetimes do not get the same register assigned. To reduce the number of intersecting intervals, the lifetime should be as small as possible, i.e. there should be no subrange in the lifetime where the virtual register does not contain a useful value. Therefore, the compiler uses a representation of lifetime intervals with holes, which allows an exact modeling of live ranges. The example in Chapter 5.5.5 illustrates lifetime intervals with holes.

5.5.2 Use Positions

The use positions of an interval store the id numbers of operations where the according virtual register is used. This information is required later on to decide which interval is split and spilled when no more registers are available, and when a spilled interval must be reloaded into a register.

Each use position has a flag use_kind denoting whether a register is required at this position or not: If the use position *must* have a register, the register allocator must guarantee that the interval has a register assigned at this position. If the interval was spilled to memory before this position, it is reloaded to a register. 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. For example, the second input operand of all arithmetic and logical instructions can be a memory operand. If the interval is spilled at such a position, the register allocator needs not reload the interval to a register. This reduces the number of spill moves significantly.

5.5.3 Fixed Intervals

Some LIR operations use operands referring to physical registers (see Chapter 4.8.3 on page 45 for details). Although the register allocator must leave these operands unchanged, they must be considered during allocation because they limit the number of available registers at certain positions. Information about physical registers is collected in *fixed intervals*: These intervals use the same data structure as normal non-fixed intervals, but are handled in a special way during register allocation.

To distinguish between fixed and non-fixed intervals, fixed intervals use a reserved range of the interval numbers. On Intel, the following numbers are used.

- The eight general-purpose registers use the interval numbers 0 to 7
- The eight registers of the FPU stack use the interval numbers 8 to 15
- The eight XMM registers of the SSE2 extensions use the interval numbers 16 to 23

The first non-fixed interval must have a number higher than all fixed intervals. Currently, the numbering starts with 40 (there is no real reason why the numbers 24 to 39 are not used). This implies that there are also no virtual registers with a register number smaller than 40.

The list of ranges is maintained in the same way both for fixed and non-fixed intervals: The lifetime of a fixed interval consists of short ranges that model the parts of the method where the according register is not available for other non-fixed intervals. For each physical register, one fixed interval cumulates all ranges where the register is blocked. Use positions are not needed for fixed intervals because they are never split and spilled.

5.5.4 Splitting of Intervals

The value of an interval need not be stored at the same location during its whole lifetime. It can reside in a register for some time and then change to memory, or vice versa. In order to accomplish this change, the interval has to be split into two intervals, one residing in a register and the other in memory. This avoids storing multiple locations for a single interval. When an interval is split, a new interval is created and appended to the list of intervals. The new interval is called a *split child* of the original interval, which is called *split parent*.

Ranges and use positions after the split positions are moved to the split child. The split parent ends at the split position, the split child starts there. The split child gets its own interval number assigned, although it belongs to the same virtual register as the split parent. Both the split parent and the split child can be split again later on. All these intervals share the same split parent. The split parent maintains a list of all split children. The function `child_at(id)` returns the split child that covers a given operation id.

The example in Figure 5.5 shows the interval with the number 40, modeling the lifetime of the virtual register `[R40]`. It has three ranges and three use positions. The interval was not split yet, so split_parent is empty and there is no list of split_children.

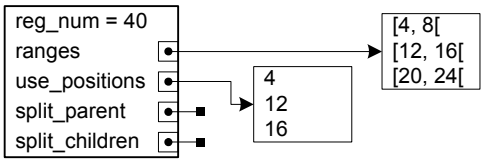


Figure 5.5: Interval with ranges and use positions

Now the interval is split twice:

- At first, the interval 40 is split at the position 14. This creates the new interval 50 with all ranges and use positions above 14. The original range [12, 16[is split into the range [12, 14[for interval 40 and [14, 16[for interval 50.
- Afterwards, the interval 50 is split at the position 20, creating the new interval 51. Although the split occurred at position 20, interval 50 now ends at position 16 because the register is not live between 16 and 20.

The newly created intervals 50 and 51 are both split children of the original interval 40. Figure 5.6 shows the resulting graph of intervals with their according ranges and use positions.

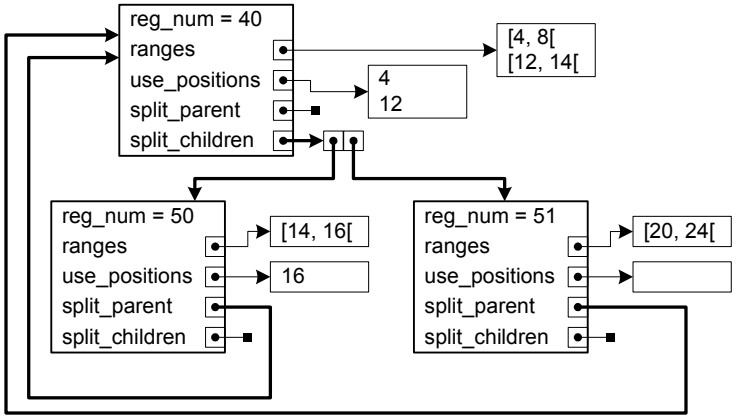


Figure 5.6: Intervals after splitting

5.5.5 Example

Ranges and use positions of lifetime intervals can be understood most easily when shown in a graphical representation. Figure 5.7 shows the intervals for the example that calculates the factorial of a number. The LIR for this example was presented in Figure 4.11 on page 44. Each interval is represented by a line. The grey rectangles show the live ranges, the small black blocks the use positions. For example, the operation with id 10—the move operation between `[R40]` and `[R42]`—is registered as a use position for the intervals 40 and 42.

Interval 43 is one example for an interval with a lifetime hole: Two move operations (with the id 12 and 34) write to the operand `[R43]`, and two move operations (with the id 26 and

44) read it. Between the operations 26 and 34, the operand contains no useful value because it is not read any more before it is overwritten. The lifetime hole avoids wasting a register between 26 and 34. Written in textual form, the interval 43 consists of the two ranges [12, 26[and [34, 44[.

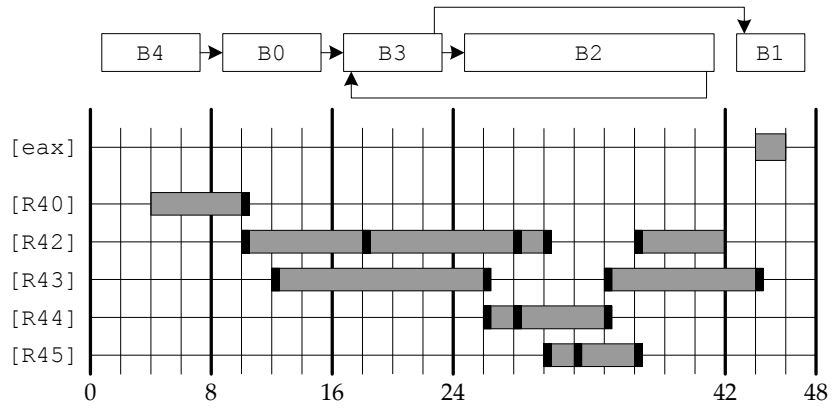


Figure 5.7: Compilation example—lifetime intervals

The first line of the example represents the fixed interval for the register `[eax]`. The operation at the id 44 is the move from `[R43]` to `[eax]`, followed by the return expecting the result in the fixed register `[eax]`. Use positions are not needed for fixed intervals, so they are not collected by the algorithm.

5.6 Building Intervals

In order to create accurate live ranges for each virtual register, a data flow analysis is performed before the intervals are built. This is necessary to model the data flow in a not sequential control flow, e.g. an operand that is defined before a loop and used in the loop is live for the entire loop because the value must be preserved for further loop iterations. Therefore, three steps are necessary to create lifetime intervals:

- At first, all operands read or written in a block are collected in the local live sets.
- Then, a standard backward data flow analysis [Aho86] computes the set of all operands that are live at the beginning and end of a block, called the global live sets.
- Using this information, accurate live ranges can be constructed.

The following chapters present the detailed algorithms for each of the three steps.

5.6.1 Compute Local Live Sets

To generate the local live sets, each block is processed and the fields live_gen and live_kill of the class BlockBegin are filled: live_gen contains all operands that are used in this block before they are defined, so they must be defined somewhere in a predecessor. The set live_kill contains all operands that are defined in the block, so a possible value of a predecessor is overwritten. This complicated handling is necessary because the LIR is not in SSA form: The phi functions of the HIR are already resolved by moves in the LIR. Algorithm 5.4 is used to compute the local live sets.

```
COMPUTE_LOCAL_LIVE_SETS
  LIR_OpVisitState visitor          // used for collecting all operands of an operation

  for each block b in blocks do
    b.live_gen = { }
    b.live_kill = { }

    for each operation op in b.operations do
      visitor.visit(op)

      for each virtual register opr in visitor.input_oprs do
        if opr ∉ block.live_kill then b.live_gen = b.live_gen ∪ { opr }
      end for

      for each virtual register opr in visitor.temp_oprs do
        b.live_kill = b.live_kill ∪ { opr }
      end for

      for each virtual register opr in visitor.output_oprs do
        b.live_kill = b.live_kill ∪ { opr }
      end for
    end for
  end for
```

Algorithm 5.4: Compute local live sets

To abstract the register allocator from the class hierarchy of LIR operations, a visitor is used to collect all operands of an operation. The visitor, implemented in the class LIR_OpVisitState, provides a comfortable access to all input, temporary and output operands of an operation and is used several times in the register allocator. In the algorithm, the operands are accessed via the sets input_oprs, temp_oprs and output_oprs of the visitor.

Temporary and output operands are treated equally here: Both overwrite a possible prior value of the operand and are therefore added to live_kill. Input operands must be defined somewhere before. When no prior definition was found in the same block, i.e. when the operand is not present in live_kill, then it is added to live_gen because it must be defined in a predecessor.

5.6.2 Compute Global Live Sets

To compute the set of all operands that are live at the beginning and end of a block, the backward dataflow analysis shown in Algorithm 5.5 is used.

```
COMPUTE_GLOBAL_LIVE_SETS
  do
    for each block b in blocks in reverse order do
      b.live_out = { }
      for each successor sux of b do
        b.live_out = b.live_out ∪ sux.live_in
      end for

      b.live_in = (b.live_out – b.live_kill) ∪ b.live_gen
    end for
  while change occurred in any live set
```

Algorithm 5.5: Compute global live sets

The live_out set of a block is the union of the live_in sets of all successors. Because no value can be generated at a control flow edge, all operands that are live at the beginning of a successors must also be live at the end of the current block. The live_in set is then calculated from the live_out set using live_kill and live_gen.

Unfortunately, the live sets in a loop cannot be computed in a single pass: When a loop end block is processed the first time, the according loop header was not processed yet because the loop header is always located before the loop end, so the live_in set of the loop header is initially empty. After processing the loop, the live_in set of the loop header is filled correctly, and the second pass over all blocks computes the correct live sets. The number of necessary iterations depends on the maximum nesting level of loops in the method. To simplify the computation, the iteration stops when the live sets do not change any more.

All live sets are internally stored as bit maps, indexed by the register number of the operands. This allows the fast implementation of operations like union and difference using logical operations. Also no iteration of the operations is necessary. Therefore, this step is very fast, even if some iterations are required until a fixpoint is reached.

5.6.3 Build Intervals

After the data flow analysis, all information necessary to construct accurate live ranges and use positions are available. Again, all operations of all blocks are iterated, but this time in reverse order. Algorithm 5.6 on the next page is used to build the intervals.

Before the operations are processed, the live_out set of the block is used to generate the ranges that must last until the last operation of the block. At first, the entire range of the block is added—this is necessary if the operand does not occur in any operation of the block. If the operand is defined in the block, then the range is shortened to the definition position later.

Then, all operations of the block are traversed in reverse order, and the visitor is used to collect the operands. In contrast to the computation of the local live sets, not only the virtual registers are processed here, but also the physical registers. Processing the physical registers creates the fixed intervals without further costs.

If an operation is a call to another method, then all registers are destroyed when the operation is executed. Short ranges of length 1 are added to all fixed intervals, so the later allocation pass cannot assign a register to any non-fixed interval at this position—otherwise two intersecting intervals would have the same register assigned. This guarantees that all intervals live at a call site are spilled to memory before the call. An example with a method call can be found in Appendix A.

Output operands of the operation shorten the first range of the interval: The definition overwrites any previous value of the operand, so the operand cannot be live immediately before this operation. The range that was defined until the start of the block is resized to the accurate length.

```
BUILD_INTERVALS
    LIR_OpVisitState visitor;          // visitor used for collecting all operands of an operation

    for each block b in blocks in reverse order do
        int block_from = b.first_op.id
        int block_to = b.last_op.id + 2

        for each operand opr in b.live_out do
            intervals[opr].add_range(block_from, block_to)
        end for

        for each operation op in b.operations in reverse order do
            visitor.visit(op)

            if visitor.has_call then
                for each physical register reg do
                    intervals[reg].add_range(op.id, op.id + 1)
                end for
            end if

            for each virtual or physical register opr in visitor.output_oprs do
                intervals[opr].first_range.from = op.id
                intervals[opr].add_use_pos(op.id, use_kind_for(op, opr))
            end for

            for each virtual or physical register opr in visitor.temp_oprs do
                intervals[opr].add_range(op.id, op.id + 1)
                intervals[opr].add_use_pos(op.id, use_kind_for(op, opr))
            end for

            for each virtual or physical register opr in visitor.input_oprs do
                intervals[opr].add_range(block_from, op.id)
                intervals[opr].add_use_pos(op.id, use_kind_for(op, opr))
            end for
        end for
    end for
```

Algorithm 5.6: Build intervals

Temporary operands add short ranges of length 1, similar to the processing of calls. A temporary operand has no defined value before and after the operation, so it is not live before and after it either. These operands are another reason for numbering the LIR operations using the distance two: A short range of length 1 is never adjacent to a range starting at the succeeding operation.

Input operands use values that are calculated somewhere before the current operation, but the actual position is not known yet. So a range from the start of the current block to the operation is added. It may be shortened later when output operands are processed, as described above. If the range is already present because the same input operand occurred in another operation, then no new range is necessary; the existing range covers all necessary operations. It is not allowed that multiple ranges of the same interval overlap.

Whenever a range is added to an interval, adjacent ranges are merged to reduce the total number of ranges. For example, when the range [4, 8[is added to the range [8, 12[, both ranges are merged to the single range [4, 12[. A use position is added for each input, temporary and output operand. The new position is simply added to the list of all use positions. The function use_kind_for checks whether this operand requires a register or if the operation can also work directly with a spilled operand, as described in Chapter 5.5.2.

5.6.4 Example

This example explains how ranges are added and then truncated to the correct length. The processing of block `B2` of the factorial example is presented step by step. Figure 5.8 shows the LIR for this block. This is a part of the entire LIR presented in Figure 4.11 on page 44.

```
B2 [6, 14] pred: B3 sux: B3
__id__Operation_____________________________________________
  24  label [label:0x31da17c]
  26  move [R43|I] [R44|I]
  28  mul  [R44|I] [R42|I] [R44|I]
  30  move [R42|I] [R45|I]
  32  sub  [R45|I] [int:1|I] [R45|I]
  34  move [R44|I] [R43|I]
  36  move [R45|I] [R42|I]
  38  safepoint [bci:14]
  40  branch [AL] [B3]
```

Figure 5.8: Compilation example—LIR of block `B2`

Figure 5.9 illustrates the building of intervals using a snapshot before, amid and after processing the operations of the block. It contains the relevant section of the complete intervals shown in Figure 5.7 on page 61.

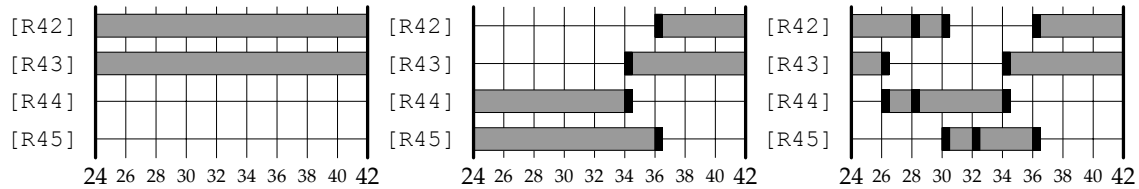


Figure 5.9: Compilation example—building intervals of Block `B2`

The live_out set of Block `B2` contains two operands `[R42]` and `[R43]`. Therefore, whole block ranges are added for both operands, as shown in Figure 5.9 a). Then, the operations are iterated in reverse order, starting with the last operation:

- The operations 40 and 38 have no operands, so they do not change the ranges.
- For operation 36, the output operand `[R42]` is processed first and the range of interval 42 is shortened: The start position is moved from 24 to 36. For the input operand `[R45]`, a range from the start of the block at 24 to 36 is added. Use positions are added to the intervals 42 and 45.
- Operation 34 is processed similarly: The output operand `[R43]` shortens the range of interval 43; the input operand `[R44]` adds a range from 24 to 34 to interval 44. Figure 5.9 b) shows the ranges after processing operation 34.
- Operation 32 has `[R45]` as input and output operand. So the range of interval 45 is first shortened to 32 by the input operand and immediately afterwards prolonged to 24 by the output operand. The range is effectively unchanged, only a new use position is registered at position 32.

- Operation 30 truncates the interval 45 to the position 30 and adds a new range from 24 to 30 to the interval 42. The interval now has a lifetime hole from 30 to 36.
- Operation 28 has `[R42]` and `[R44]` as input operands and `[R44]` as the output operand. Similarly to operation 32, the interval 44 is shortened and prolonged, so the range remains the same. The input operand `[R42]` also does not change interval 42 because the range from 24 to 28 is already present. Only the according use positions are added to the intervals 42 and 44.
- Operation 26 truncates the interval 44 to position 26 and adds a new range from 24 to 26 to interval 43.

The final ranges after the processing of all operations are shown in Figure 5.9 c).

5.7 Allocation

During the allocation phase, the unbound number of virtual registers—represented by the intervals—is mapped to the small set of physical registers. All intervals are sorted in the order of increasing start points and traversed in this order. The interval currently processed is called the current interval. The starting position of this interval, i.e. the from field of its first range, divides the remaining intervals into the following four sets:

- The unhandled set contains all intervals starting after position.
- The intervals in the active set cover position and have a register assigned.
- The inactive intervals start before position and end after position, but do not cover it because position is in a lifetime hole.
- All handled intervals end before position or were spilled to memory. These intervals are not processed any more, so it is not necessary to store them in a set.

If an interval is handled because it is spilled to memory, then it must not contain a use position that requires a register. When an interval with such a use position is spilled, the part containing the use position is split off and moved to the unhandled set to get a register assigned.

5.7.1 Walking Intervals

The main allocation loop processes all intervals of the sorted unhandled set, initialized with all intervals created in the steps before. In each iteration, the interval with the lowest starting position is removed from unhandled and processed by the allocator. During allocation, new intervals may be sorted into the unhandled set: When an interval is split, the split child is not processed immediately but instead added to unhandled. This postpones the decision what to do with split children until the allocator reaches their starting position.

Algorithm 5.7 shows how the active and inactive sets are adjusted before a register is searched for current. Intervals from active that do not cover the current position are either moved to handled if they end before position or moved to inactive. Similarly, intervals from inactive are moved to handled or active.

```
WALK_INTERVALS
    unhandled = list of intervals sorted by increasing start point
    active = { }
    inactive = { }

    // note: new intervals may be sorted into the unhandled list during
    // allocation when intervals are split
    while unhandled ≠ { } do
        current = pick and remove first interval from unhandled
        position = current.first_range.from

        // check for intervals in active that are expired or inactive
        for each interval it in active do
            if it.last_range.to < position then
                move it from active to handled
            else if not it.covers(position) then
                move it from active to inactive
            end if
        end for

        // check for intervals in inactive that are expired or active
        for each interval it in inactive do
            if it.last_range.to < position then
                move it from inactive to handled
            else if it.covers(position) then
                move it from inactive to active
            end if
        end for

        // find a register for current
        TRY_ALLOCATE_FREE_REG
        if allocation failed then
            ALLOCATE_BLOCKED_REG
        end if

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

Algorithm 5.7: Walk intervals for allocation

The method try_allocate_free_reg tries to find a register for current without spilling an interval. In the best case, a register is free for the entire lifetime of the interval, but it is also sufficient if a free register is found only for the beginning of the interval. If the allocation without spilling fails, then allocate_blocked_reg tries harder to find a register by spilling some intervals. The spilling decision is based on the use positions; the interval that is not used for the longest time is spilled. It is also possible that current itself is spilled if it is used later than all other intervals that block the registers.

5.7.2 Selection Strategy for Registers

Algorithm 5.8 is used to select a register for the current interval that is not occupied by any other interval. The algorithm inspects the active and inactive intervals, but does not split or change the assigned register of any interval but current.

```
TRY_ALLOCATE_FREE_REG
    set free_pos of all physical registers to max_int

    for each interval it in active do
        set_free_pos(it, 0)
    end for

    for each interval it in inactive intersecting with current do
        set_free_pos(it, next intersection of it with current)
    end for

    reg = register with highest free_pos
    if free_pos[reg] = 0 then
        // allocation failed, no register available without spilling
        return false
    else if free_pos[reg] > current.last_range.to then
        // register available for whole current
        assign register reg to interval current
    else
        // register available for first part of current
        assign register reg to interval current
        split current at optimal position before free_pos[reg]
    end if
```

Algorithm 5.8: Allocate register without spilling

All intervals of the active and inactive sets can possibly affect the allocation decision. They are used to fill the array free_pos: Each register is available for allocation until its free_pos. Before the active and inactive sets are processed, all physical registers are marked as entirely free by setting the free_pos to a high number (the maximum integer number is used). Then, the intervals are processed:

- All registers used by active intervals must be completely excluded from the allocation decision, so their free_pos is set to 0.
- Inactive intervals that do not intersect with current can be completely ignored because they do not disturb each other.
- The free_pos for registers of inactive intervals intersecting with current is set to the next intersection point, i.e. the register is available at the beginning because position is in a lifetime hole of the inactive interval, but it is not available for the whole lifetime of current.

The method set_free_pos modifies the free_pos for the register assigned to the given interval. If the position for one register is set multiple times—this can happen when many inactive intervals have the same register assigned—the minimum of all positions is used. The register with the highest free_pos is searched and used for allocation. Three cases can be distinguished:

- If the highest free_pos is 0, then all registers are occupied by active intervals. No free register is available for allocation. It is not possible to allocate a register without spilling, so Algorithm 5.9 on page 70 is used for allocation with spilling.
- If free_pos is higher than the end position of current, the register is available for the entire lifetime of the interval. This is the best case: The register is simply assigned to current, and the allocation completes successfully.

- If free_pos lies somewhere in the middle of current, the register is available for the first part of the interval only. The register is assigned to current, but current is split at free_pos (or even before). The split child is sorted into the unhandled list and will be processed later. A move operation between the two intervals is inserted at the split position.

Assigning a register only for the first part of the interval is an important optimization. It guarantees a good register utilization even if many long intervals with complex lifetime holes compete for registers. Long intervals can switch between different registers, so the probability for spilling is lower.

In the current implementation, allocating partial intervals is also necessary for method calls: Since all registers are blocked at a call by adding a short range to the fixed intervals, an interval live at a call site can never get a register for the 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.

Figure 5.10 illustrates the three different cases, in which the intervals 41 and 42 are slightly different. Assume that only two physical registers `[r1]` and `[r2]` are available for allocation. The intervals 40 and 41 have already the registers `[r1]` and `[r2]` assigned, respectively. Interval 42 is the current interval for allocation (printed as a light grey bar).

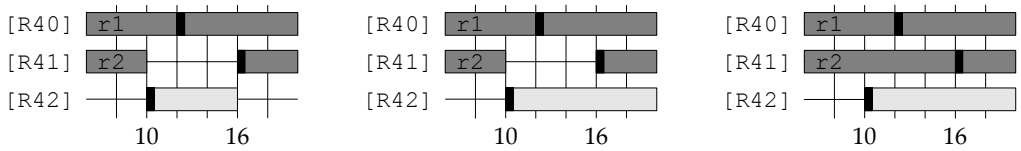


Figure 5.10: Example of allocation without spilling

The free_pos of register `[r1]` is 0 in all three examples because interval 40 is always active, so `[r1]` is never available for allocation. The free_pos of `[r2]` is different:

- In Figure 5.10 a), interval 41 does not intersect with interval 42, so it does not disturb the allocation. The free_pos of register `[r2]` has its initial high value, so `[r2]` is assigned to interval 42 without any splitting necessary.
- In Figure 5.10 b), the intervals 41 and 42 intersect; the free_pos of register `[r2]` is 16. Interval 42 gets the register `[r2]` assigned, but must be split before position 16. The split child is appended to the unhandled list and processed later, where it has the chance to get another register.
- In Figure 5.10 c), both registers are blocked because the intervals 40 and 41 are active. Since the free_pos of both registers is 0, the allocation fails. At least one interval must be spilled, as described in the next chapter.

5.7.3 Spilling of Intervals

When more intervals are simultaneously live than physical registers are available, spilling is inevitable. Finding the optimal interval for spilling with the smallest negative impact on the overall performance is too complicated. Instead a simple heuristic based on the use

positions is applied: The interval that is not used for the longest time is spilled because this frees a register as long as possible.

The heuristic also estimates the weights for intervals used by other register allocation algorithms: If an interval is used often and would therefore have a high weight assigned, then there is probably also a near use position, so the interval is not spilled. The heuristic also works well for intervals with changing utilization: If an interval is heavily used at first and then remains unused for a long time, then it is not spilled before the unused part.

Algorithm 5.9 shows the code that selects the spilled interval and assigns the freed register to the current interval.

```
ALLOCATE_BLOCKED_REG
    set use_pos and block_pos of all physical registers to max_int

    for each non-fixed interval it in active do
        set_use_pos(it, next usage of it after current.first_range.from)
    end for

    for each non-fixed interval it in inactive intersecting with current do
        set_use_pos(it, next usage of it after current.first_range.from)
    end for

    for each fixed interval it in active do
        set_block_pos(it, 0)
    end for

    for each fixed interval it in inactive intersecting with current do
        set_block_pos(it, next intersection of it with current)
    end for

    reg = register with highest use_pos
    if use_pos[reg] < first usage of current then
        // all active and inactive intervals are used before current, so it is best to spill current itself
        assign spill slot to current
        split current at optimal position before first use position that requires a register
    else if block_pos[reg] > current.last_range.to then
        // spilling made a register free for whole current
        assign register reg to interval current
        split and spill intersecting active and inactive intervals for reg
    else
        // spilling made a register free for first part of current
        assign register reg to interval current
        split current at optimal position before block_pos[reg]
        split and spill intersecting active and inactive intervals for reg
    end if
```

Algorithm 5.9: Allocate register with spilling

Again, all intervals of the active and inactive list can possibly affect the decision. The algorithm collects two numbers for each physical register based on these intervals:

- use_pos[reg] stores the position where an interval with the register reg assigned is used next. If more than one position is available, then the minimum is used. This number is used for the heuristic selecting the spill candidate. It is calculated by iterating all non-fixed active and inactive intervals and searching their next use position after the start position of current.

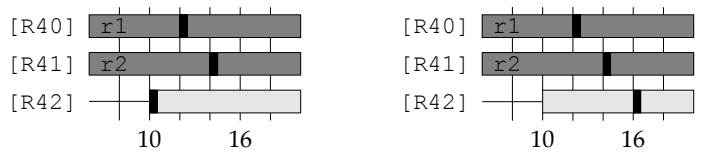
- block_pos[reg] stores a hard limit for each register where the register cannot be freed by spilling. This position is set by the fixed active and inactive intervals that model the operations requiring operands in fixed registers. The register allocator must adhere to these constraints since fixed intervals can never be spilled. The method set_block_pos implicitly sets the use_pos, so use_pos of a register is never higher than block_pos.

The register with the highest use_pos is selected as the best candidate for the current interval. Based on the collected positions, three cases can be distinguished:

- If the first use position of the current interval is found after the highest use_pos, it is better to spill current. It is split before its first use position where it must be reloaded. The active and inactive intervals are not changed and remain in their old locations. This case is also taken if all registers are blocked at a call site: All fixed registers are active at call sites, therefore the block_pos and the use_pos of all registers is 0.
- Otherwise, current gets the selected register assigned. All active and inactive intervals for this register intersecting with current are split before the start of current and spilled to the stack. These split children are not considered during allocation any more because they do not have a register assigned. If they have a use positions requiring a register, however, they must be reloaded again to a register later on. Therefore, they are split a second time before these use positions, and the second split children are sorted into the unhandled list. They get a register assigned when the allocator advances to the start position of these intervals.
- The third case is an extension of the second: If the selected register has a block_pos somewhere in the middle of current, then the register is not available for the whole lifetime. So current is split before block_pos, and the split child is sorted into the unhandled list.

This algorithm guarantees that current either gets a register assigned or is spilled itself. Many new split children may be created and sorted into the unhandled list, but all split children have a starting position after the start of current. This guarantees that the allocator can always advance. The allocation pass is guaranteed to terminate; endless loops of splitting and spilling cannot occur.

Figure 5.11 shows an example with two slightly different variations of the same intervals. The ranges are equal, only the use position of interval 42 is different. Assume that only two physical registers `[r1]` and `[r2]` are available for allocation, and that all use positions require a register. Normally, each interval starts with a use position as in Figure 5.11 a), but situations like Figure 5.11 b) can occur for split children.



a) splitting of intersecting interval b) splitting of current interval

Figure 5.11: Example of spilling intervals—before allocation of interval 42

The intervals 40 and 41 were already allocated before: Interval 40 got the register [r1] assigned, interval 41 the register [r2]. Interval 42 is the current interval for allocation (printed as a light grey bar in the figure). Table 5.3 shows the use_pos and block_pos for both registers.

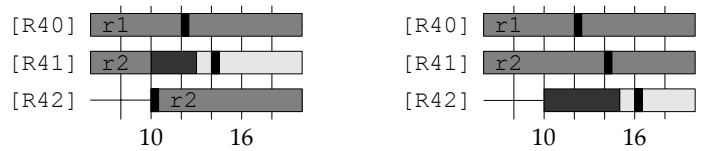
	use_pos	block_pos
[r1]	12	max_int
[r2]	14	max_int

Table 5.3: Registers state for spilling

The allocation result is different for case a) and b). In Figure 5.11 a), the maximum use_pos of the two intervals is 14, so [r2] is the best candidate for allocation. Because the first use position of interval 42 at position 10 is located before 14, interval 41 is split at the following positions: The first split is at position 10, the split child is spilled. Because of the use position at 14, the interval is split again before 14 (assume at position 13). This split child is appended to unhandled and processed when the allocator advances to position 13.

In Figure 5.11 b), the first use of the current interval 42 is not at position 10, but at 16. Since it is higher than the maximum use_pos of [r2], current is spilled itself. It is split at position 15, and the split child with the use position 16 will get a register later when the allocator processes this interval.

Figure 5.12 shows the result after processing interval 42: The spilled split child is printed as a dark grey bar; the split child with the use position that is sorted into unhandled is printed as a light grey bar. The spilled part is as long as possible in both cases, i.e. the new interval sorted into the unhandled list starts as late as possible.



a) splitting of intersecting interval b) splitting of current interval

Figure 5.12: Example of spilling intervals—after allocation of interval 42

5.7.4 Optimal Split Position for Intervals

Even during the allocation of medium-sized methods, many intervals must be split. The six general-purpose registers available for allocation on the IA-32 platform barely suffice for the local variables and temporaries of any method. Additionally, all intervals must be split and spilled for each call site because no callee-saved registers are available on Intel platforms.

The negative impact of spilled intervals can be reduced by searching the optimal position for splitting. In most cases, the split position is not fixed to a single position, but can be chosen from a range. In general, the position where an interval is spilled or reloaded can be moved to a lower position, while the upper bound is specified by the position calculated by the algorithms. The following three rules are used to find an optimal split position:

- *Move split position out of loops*: Loop blocks are executed far more often than blocks of a sequential control flow. Therefore, spilling or reloading inside a loop leads to a higher number of memory accesses than spilling before or after a loop. The loop information calculated during block ordering is used to move the split position to the latest position with the lowest loop depth.
- *Move split position to block boundaries*: When an interval is spilled, a move operation from the old to the new location must be inserted. If a split position is moved to a block boundary, the algorithm for resolving the data flow takes care of inserting the move. It is also possible that no move is necessary at all because of the control flow.
- *Move split position to odd positions*: If the split position cannot be moved to a block boundary, then the interval is split at an odd position. Normal operations have only even positions assigned, so all odd positions are available for spill moves.

Although the algorithm for searching the optimal split position is only a heuristic, it accounts much to the overall quality of the register allocation.

5.8 Resolving the Data Flow

As described earlier, the linear scan approach to register allocation simplifies the control flow graph to a linear list of basic blocks before allocation. The lifetime intervals hold the information how long a virtual register contains a useful value. This data is correct as long as no intervals are split. When an interval is split, a move operation is inserted from the old to the new location at the split position, so the data flow is correct in the basic block where the split occurred. But the linear block list models the control flow incompletely, so an additional resolving step is necessary.

Figure 5.13 shows a simple example of an interval where resolving is necessary: The four blocks of an if-then-else statement have been sorted into a linear order. Assume that the interval was split in the middle of block `B3` at the position 32, so it is in a register before 32 and spilled to the stack after 32.

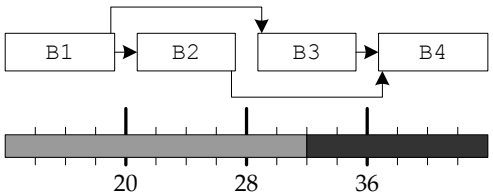


Figure 5.13: Example where resolving of data flow is necessary

If the control flow `B1` → `B3` → `B4` is taken, everything is correct because a move was inserted before operation 32. But in the alternative control flow `B1` → `B2` → `B4`, the interval is still in the register at the end of `B2`, but expected on the stack at the beginning of `B4`. A resolving move from the register to the stack must be inserted between `B2` and `B4`. Conceptually, the move is inserted at the edge between the blocks `B2` and `B4`, but actually it is inserted at the end of `B2`.

Algorithm 5.10 shows the code that inserts resolving moves between basic blocks. Each edge between basic blocks is processed separately. The live_in set of the successor block is used to iterate through all virtual registers that are live at the edge.

```
RESOLVE_DATA_FLOW
    MoveResolver resolver          // used for ordering and inserting moves into the LIR

    for each block from in blocks do
        for each successor to of from do
            // collect all resolving moves necessary between the blocks from and to
            for each operand opr in to.live_in do
                Interval parent_interval = intervals[opr]

                Interval from_interval = parent_interval.child_at(from.last_op.id)
                Interval to_interval = parent_interval.child_at(to.first_op.id)

                if from_interval ≠ to_interval then
                    // interval was split at the edge between the blocks from and to
                    resolver.add_mapping(from_interval, to_interval)
                end if
            end for

            // the moves are inserted either at the end of block from or at the beginning of block to,
            // depending on the control flow
            resolver.find_insert_position(from, to)

            // insert all moves in correct order (without overwriting registers that are used later)
            resolver.resolve_mappings()
        end for
    end for
```

Algorithm 5.10: Resolving the data flow

For each operand that is live at the currently processed edge from → to, an interval was created during the building of intervals. This interval is the split parent of all split children created during allocation. The function child_at(id) returns the split child covering the operation id (or the interval itself if it has not been split). For resolving, two intervals are searched:

- from_interval stores the location of the virtual register at the end of block from.
- to_interval stores the location of the virtual register at the beginning of block to.

If these two locations are different, the interval was split somewhere in between, so a resolving move must be inserted. When multiple moves must be inserted at one edge, then the order of the moves is important because the same register can occur as the source of one move and the destination of another move. The moves must be ordered such that a register is saved first before it is overwritten. The ordering of moves and the insertion into the LIR is implemented in the class MoveResolver.

5.9 Assignment of Register Numbers

After allocation, the lifetime intervals hold the mapping from virtual registers to physical registers. In the last step, this information is propagated back to the LIR operations. Algorithm 5.11 shows the straightforward iteration over all blocks, operations and operands.

```
ASSIGN_REG_NUM
    LIR_OpVisitState visitor          // used for collecting all operands of an operation

    for each block b in blocks do
        for each operation op in b.operations do
            visitor.visit(op)

            // process input, temporary and output operands
            for each virtual register v_opr in visitor.oprs do
                // calculate new operand based on the register assigned to the interval
                r_opr = intervals[v_opr].child_at(op.id).assigned_opr

                // store the new operand back to the operation
                visitor.set_opr(r_opr)
            end for

            if op is move with equal source and target then
                // remove useless moves from the list of LIR operations
                remove op from b.operations.
            end if
        end for
    end for
```

Algorithm 5.11: Assign register numbers

All operands are handled equally; there is no difference between input, temporary and output operands. The list of intervals and the split children are used to search the physical location of a virtual register:

- First, the original interval created during the building of intervals for the virtual register is searched.
- The function child_at returns the split child for the currently processed operation id.
- The split child either has a register or a stack slot assigned. A corresponding LIR-operand is created for the assigned location.
- The virtual register is replaced by the new operand using the visitor.

During the construction of the LIR, many move operations are inserted for technical reasons: They are needed for resolving phi functions, to ensure the two-operand form of the LIR and for moves from and to fixed registers. Many of these moves are unnecessary after register allocation because the allocator succeeded to put the source and the target operand in the same register. Such moves are removed from the LIR.

After the register assignment, the LIR does not contain references to virtual registers any more. The LIR is ready for code generation: All constraints of LIR operations requiring either arbitrary or fixed registers are met. Only the handling of the FPU register stack requires additional work, as described in Chapter 6.

5.10 Move Optimizations

About 50% of all LIR operations are move operations, and most moves access the memory. Reducing the number of moves also reduces the number of memory accesses, which is the original goal of register allocation. Several optimizations try to avoid or eliminate unnecessary moves.

5.10.1 Register Hints

In the linear scan algorithm described above, the selection strategy for registers is based on the intersection and use position of the intervals only. But there are some cases where an interval should get a certain register assigned: If an interval is defined by a move operation where both the source and the target operand are virtual registers, then the target should get the same register assigned as the source. In this case, the move operation is unnecessary and therefore deleted during the assignment of register numbers.

To model such dependencies between intervals, *register hints* are used. Each interval has a field register_hint filled by the following algorithm:

```
for each move operation move in all operations of all blocks do
    intervals[move.target].register_hint = intervals[move.source]
end for
```

The register hint is later used when a register is processed during allocation: If possible, the interval gets the same register as the hint interval, even if this register is not optimal according to the previously explained algorithm, i.e. even if it is not the register with the highest free_pos in Algorithm 5.8.

Selecting non-optimal registers could result in a higher number of intervals that must be spilled later on, but measurements show that register hints have an overall positive effect on the quality of register allocation, the total number of move operations in the LIR decreases.

5.10.2 Spill Optimization

When an interval is split, a move operation from the old to the new location is inserted into the LIR. The move can either be a move from a register to the stack, or from the stack to a register. Moves from the stack to a register are always necessary, but moves from a register to the stack can be eliminated in certain cases: When the stack slot is known to be correct, i.e. when it can be proven statically that the stack slot already contains the same value as the register, the move can be deleted. Normally this is difficult to prove, but there are two special cases where it is easy:

- Method parameters are passed on the stack and loaded from there when they are required in a register. When such an interval is spilled later, then no store to the stack slot is necessary because parameters never change their value.
- Most intervals have only one operation that defines the value that is used multiple times later on. If such an interval is spilled and reloaded several times, then a spill move is inserted directly after the definition. This guarantees that the stack slot is correct in all possible code paths, so all further moves to this stack slot can be eliminated.

Spill optimization reduces the number of stores to spill slots significantly. Especially the this-pointer of a method, passed as the first method parameter, is frequently used in a method and therefore often spilled and reloaded. Because the this-pointer never changes, all spill stores to its stack slot are unnecessary.

5.10.3 Merging Moves

If all predecessors of a block end with the same sequence of move operations, then these moves can be placed once at the begin of the successor instead of multiple times at the end of the predecessors. This optimization is performed after the assignment of register numbers because moves with equal physical operands can be merged, even if they originate from different virtual registers. Similarly, equal moves at the beginning of all successors can be placed at the end of the predecessor.

This optimization especially processes moves that were inserted during the resolution of phi functions (see Chapter 4.8.1) and during data flow resolution (see Chapter 5.8). The number of moves executed dynamically at runtime is not changed, but nevertheless the optimization has two positive effects:

- The static number of moves is reduced, so the native code of the method is smaller.
- Many blocks that originally contain only moves become empty by the optimization and can be deleted entirely, reducing the number of jumps executed at runtime.

Figure 5.14 shows an example where moves are merged: The last two moves of the blocks `B1` and `B2` are equal, so they can be merged and placed at the beginning of `B3`. The first move of `B1` is not present in `B2`, so it remains in `B1`. But block `B2` is empty after the optimization, so it will be deleted.

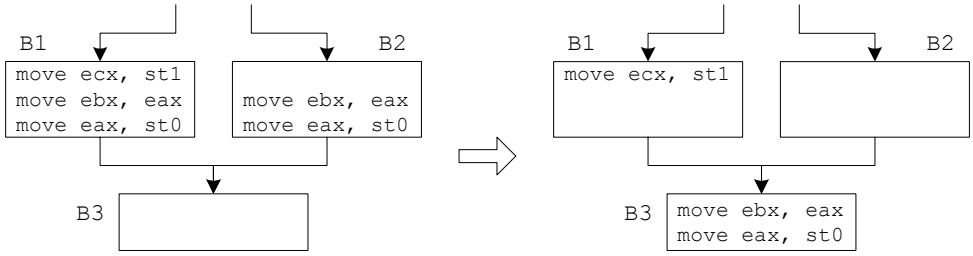


Figure 5.14: Example before and after merging moves

Handling of Floating Point Values

This chapter presents the architecture of the IA-32 floating point unit (FPU). Its floating point registers are organized as a stack, requiring additional work for the register allocator because register numbers must be translated to stack indices. Additionally, the data format of FPU registers does not match the IEEE standard used by Java, so explicit rounding is necessary. Finally, the SSE and SSE2 extensions requiring no special handling are presented as an alternative to the FPU for modern processors.

In the IA-32 architecture, floating point instructions are executed in the floating point unit (FPU, [Intel1]). Historically, the FPU was separated from the main processor and located in a coprocessor. Since the time of the Intel486 processor, the FPU is integrated in all processors and therefore generally available. But the separation is still visible in the instruction set and the instruction format of floating point instructions. They use a completely different paradigm that 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 to the IEEE 754 standard for floating point arithmetic [IEEE754] required by the Java specification. The FPU has a higher precision than specified, so explicit rounding is necessary.
- 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 simulation of the FPU stack.

The SSE and SSE2 extensions offer single-instruction multiple-data (SIMD) instructions for floating point values. The SSE instructions operate on four single-precision floating point values, whereas the SSE2 instructions operate on two double-precision floating point values. Both extensions were also designed as a complete replacement for the FPU. All SSE and SSE2 instructions are also available in a scalar form operating only on one value.

These instructions adhere to the IEEE standard and allow a direct addressing of registers, so they are much easier to handle in the compiler. If the SSE2 extensions are available on the processor, then the compiler creates code that uses SSE2 instructions instead of FPU instructions. Usually, this code executes faster because no rounding is necessary. The SSE extensions alone are not sufficient because they contain only instructions for single-precision values. The SSE2 extensions add support for double-precision values.

6.1 Intel FPU Architecture

The execution environment of the FPU consists of 8 data registers and several control and status registers. Each data register has a width of 80 bits and stores a floating point value in the so-called double extended-precision floating point format. The registers are organized as a stack that grows downwards. The register number of the current stack top, stored in a status register, is decremented by load instructions (equivalent to a push on the stack) and incremented by store instructions (equivalent to a pop). There is no possibility to address a register by its register number; all addressing is done relative to the stack top. Figure 6.1 shows the FPU register stack.

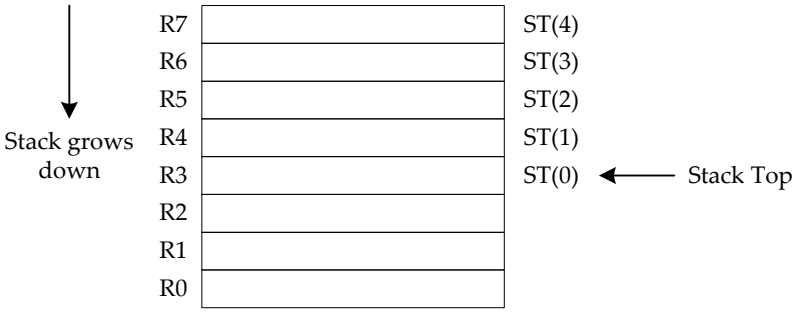


Figure 6.1: FPU register stack

6.1.1 Instruction Set

The FPU offers the usual arithmetic instructions for floating point values, together with instructions for loading, storing and comparing values and instructions for manipulating the register stack. As a convention, all FPU instructions start with the prefix "`f`" [Intel2A].

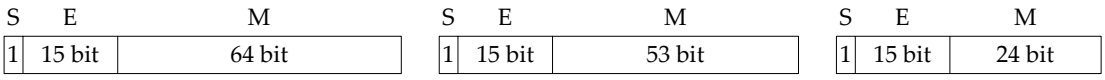
- FPU registers can be loaded from (instruction `fld`) and stored to (instruction `fst`) memory in various formats, including 32, 64 and 80-bit floating point formats and 32-bit integer numbers. All formats are automatically converted from and to the internal 80-bit floating point format.
- Arbitrary moves between floating point registers are not possible. Instead, a register can be loaded to the stack top (`fld`), the stack top can be stored in another register (`fst`), or the stack top can be exchanged with another register (`fxchg`). A raw stack pop can be simulated by incrementing the stack top pointer. All these instructions are directly available as LIR operations in the compiler to allow FPU stack manipulations in the LIR.
- In addition to the basic arithmetic instructions `fadd`, `fsub`, `fmul` `fdiv` and `frem`, trigonometric functions like `fsin` and `fcos` and transcendent functions like `fsqrt` are available.

All instructions require at least one operand at the top of the FPU stack. Only the second operand of binary instructions can be an arbitrary stack index. If none of the two operands is on the top of the stack, an `fxchg` is necessary prior to the actual instruction to bring one operand to the top.

Instructions reading an operand from the stack top but not writing the result to it are available in two variants: The normal variant does not change the stack top, while the variant with a trailing "p" in the instruction name pops the stack top. These instructions can be used to remove obsolete operands from the register stack without the need for an explicit pop instruction.

6.1.2 Precision Control

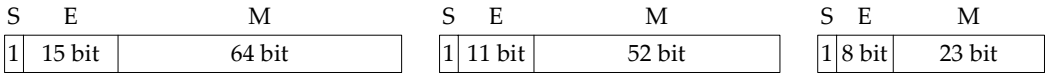
All floating point registers have a width of 80 bits, so register values are internally always stored in the double extended-precision format. However, the precision of calculations can be limited by setting a precision control flag. Figure 6.2 shows the three different precisions available for calculations. The formats differ only in the bits used for the mantissa; the exponent always has a width of 15 bits.



a) Double extended-precision b) Double-precision c) Single-precision

Figure 6.2: Available precisions for calculations

The formats for loading and storing floating point values from and to memory are independent from the calculation formats. Again, three formats are available, but with slightly different bit widths. Figure 6.3 shows the formats available for memory access. The single- and double-precision formats exactly match the IEEE 754 standard for single- and double-precision floating point numbers.



a) Double extended-precision b) Double-precision c) Single-precision

Figure 6.3: Available precisions for memory access

6.2 Rounding of FPU Registers

The two floating point types of the Java programming language, `float` and `double`, are conceptually associated with the single-precision 32-bit and double-precision 64-bit format of the IEEE 754 standard. These are the formats shown in Figure 6.3 c) and b), respectively. The value sets for Java are strictly defined in the specification [Gosling00]. All virtual machines must enforce them to guarantee the platform-independent semantics of floating point calculations.

Because the Intel FPU does not adhere to the IEEE 754 standard, it is difficult to implement the Java semantics on Intel processors. The internal formats for calculation have a too high precision that must be explicitly rounded to a lower precision. The Java language specification does not allow a higher precision of floating point values than defined in the standard. The only exceptions are values on the Java operand stack, as explained later. The compiler uses the following settings and operations to meet the Java specification.

- The precision control of the FPU is set to double-precision, so the format shown in Figure 6.2 b) is used internally by the processor for all floating point calculations. This precision is high enough for both `float` and `double` calculations of the Java programming language.
- For calculations using the type `double`, the internal format has the correct bit width of the mantissa, but a too wide exponent. This format is allowed for values on the operand stack by the Java language specification, but not for values stored in local variables and object fields. These values must be rounded explicitly.
- For calculations using the type `float`, the internal format is far too precise. After each instruction, the result must be rounded explicitly.

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 prevents the generation of effective floating point code. It is not allowed to hold a floating point value in a register between its definition and use, even if a register would be free. In the compiler, the rounding of floating point values needs special handling in all phases of the compilation:

- The HIR instruction RoundFP rounds the result of a computation when it would be saved in a local variable.
- The LIR operation lir_roundfp rounds a floating point register in the LIR.
- During linear scan register allocation, the output operand of lir_roundfp is always assigned a spill slot. So the rounding operation of the LIR is converted to a store to memory in native code.

6.3 FPU Stack Allocation

The second issue in conjunction with the Intel FPU is the stack-based handling of floating point registers. The linear scan algorithm for register allocation is not capable of handling stack indices. Register allocation for the floating point registers ignores the stack-based handling and emits code that directly addresses all eight registers. The *FPU stack allocation* converts the register numbers to stack indices. Algorithm 6.1 shows the main steps necessary for FPU stack allocation

```
ALLOCATE_FPU_STACK
    for each block b in blocks do
        load initial FPU stack state of b to simulator

        for each floating point operation op in b.operations do
            bring input operands of op on top of FPU stack if necessary
            simulate effect of op on FPU stack
            replace register numbers with stack indices
        end for

        for each successor sux of b do
            PROCESS_EDGE(b, sux)
        end for
    end for
```

Algorithm 6.1: Allocate FPU stack

The FPU stack allocation operates on the LIR where virtual registers were already replaced by physical registers. All operations with floating point operands are processed in a FPU stack simulator. The simulator provides a mapping from register numbers to stack indices. The simulation state is updated when an operation modifies the FPU stack, e.g. when items are pushed, popped or exchanged.

Most FPU operations require at least one operand at the top of the stack. If no operand is on top before the operation, then an exchange operation is inserted in the LIR before the actual operation. If an input register is no longer required after an operation, i.e. when the operation is the last use position of the according interval, then the register should be removed from the stack. If this register is on top of the stack, then the special variant of the instruction that pops the stack top is used. If the register is not on top of the stack, the register is left as a dead value on the stack.

When a block has more than one predecessor, the stack states of the predecessors must be merged because each predecessor has a different register order on the stack. The first predecessor defines the initial state of the block, all other predecessors are merged with this state. During merging, operations that modify the stack are inserted at the end of the predecessor.

6.3.1 FPU Stack Simulation

In the stack simulator, the FPU stack is represented as an array of registers. Figure 6.4 shows a simple example of a stack containing the five register `R2`, `R0`, `R4`, `R3` and `R7` (in this order), where `R7` is the current top of the stack. Assume that the register `R0` should be stored to the memory location `mem`. Because the value is not needed later any more, the register should also be removed from the stack.

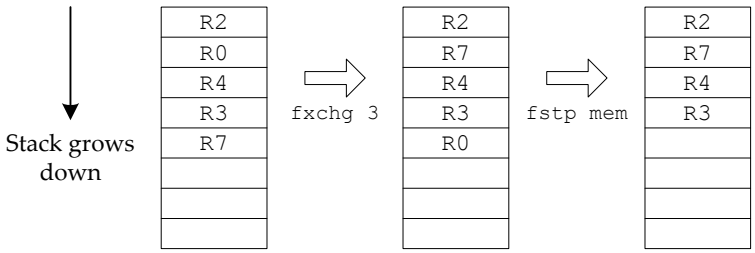


Figure 6.4: FPU stack simulation

First, `R0` must be moved to the top of the stack. The registers `R0` and `R7` are exchanged by the instruction `fxchg 3` because `R0` has the index 3, counted from the top of the stack. Then the variant of the store instruction is used that pops the argument from the stack and stores it to the memory location. This avoids an explicit pop instruction. The store instruction does not need a stack index as a parameter since it always operates on the stack top.

Each LIR operation affecting the FPU stack is processed by the simulator. The following list summarizes the actions for different classes of operations.

- Arithmetic operations for floating point operands require one operand on top of the stack. If neither the left operand nor the right operand is the current top, one

operand is moved to top with an `fxchg` operation. The register numbers of the original operation are replaced by the according stack indices.

- Move operations from memory or from one register to another register are replaced by an `fld` operation. The result is always on top of the stack.
- Move operations from a register to the memory are replaced by an `fst` operation. The source operand is brought to the top of the stack before.
- Return operations require the stack to be empty, so all dead registers that are still present on the stack must be popped. If the method returns a floating point value, then the result is passed to the caller as the only value on the FPU stack.
- When another method is called, the FPU stack must be empty too. The register allocator guarantees that no register is live at the call; all registers that are still present on the stack are dead registers and must be popped.
- All other operations with no floating point operands can be ignored.

6.3.2 Merging FPU Stacks

Before a block is processed, the initial state of the FPU stack at the beginning of the block is loaded into the simulator. This state is provided by the predecessor blocks: Because the state does not change at control flow edges, the state at the beginning of the block is equal to the state at the end of its predecessors. Algorithm 6.2 shows the actions necessary for processing a control flow edge between the blocks from_block and to_block. The current state refers to the state at the end of from_block because this block was simulated last.

```
PROCESS_EDGE(BlockBegin from_block, to_block)
    if to_block has only one predecessor then
        copy current state to to_block
    else if from_block is first predecessor of to_block then
        cleanup current state
        set initial state of to_block to current state
    else
        // initial state of to_block already present
        merge current state with initial state of to_block
    end if
```

Algorithm 6.2: Process a control flow edge

If a block has only one incoming edge, i.e. one predecessor, then no special handling is needed for this edge. The initial state is simply a copy of the state at the end of the predecessor. Dead registers remain on the stack because they do not disturb the further processing.

If a block has more than one predecessor, all predecessors must end with the same state. The first predecessor defines the initial state of the block. This state is arbitrary—no fixed ordering of the registers is required—but it must not contain any dead values. The cleanup removes all dead registers from the stack by exchanging them to the stack top and popping them. The cleanup code is placed at the end of the predecessor.

All other predecessors are merged with the initial state set by the first predecessor. The code inserted for merging at the end of the predecessor exchanges registers of the stack until it matches the initial state. Because all dead registers were removed from the initial state, it is guaranteed that the stack size does not increase when stacks are merged.

Figure 6.5 illustrates step by step when a copy, cleanup or merge is necessary. The control flow graph consists of 4 blocks `B1`, `B2`, `B3` and `B4` that are also processed in this order.

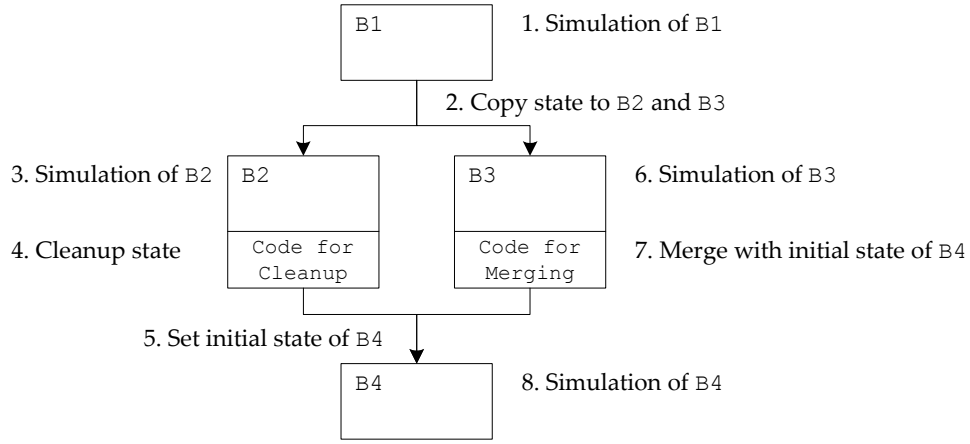


Figure 6.5: Copy, cleanup and merge of FPU stack state

The following operations are performed:

1. Block `B1` is simulated. When all operations are processed, the current state reflects the FPU stack state at the end of `B1`.
2. The successors `B2` and `B3` of `B1` have only one predecessor. So the current state is copied to `B2` and `B3`, so both blocks have an initial state equal to the state at the end of `B1`.
3. Block `B2` is simulated, starting with the initial state defined in step 2. When all operations are processed, the current state might contain dead registers.
4. The successor `B4` of `B2` has two predecessors, so merging is necessary for this block. Because `B2` is the first predecessor of `B4`, the current state is cleaned so that it does not contain dead registers. The operations for cleanup are inserted at the end of `B2`.
5. The cleaned current state is set as the initial state of `B4`.
6. Block `B3` is simulated, starting with the initial state defined in step 2. The simulation works completely independent from any state of step 3 to 5.
7. The successor `B4` of `B3` has already an initial state defined. Therefore, the current state is merged with the initial state of `B3`. The operations for merging are inserted at the end of `B3`.
8. Block `B4` is simulated, starting with the initial state defined in step 5.

6.3.3 Algorithm for Stack Cleanup

When a state is cleaned or merged, only exchange and pop operations are inserted. Other operations are not required. The algorithm for cleanup is very simple: as long as the state contains dead registers, they are exchanged with the stack top and popped. Figure 6.6 shows the details for the cleanup at the end of `B2` in Figure 6.5. Assume that the registers `R1` and `R5` are dead, illustrated by grey rectangles. The left side shows the original stack before cleanup, the right side the final state after cleanup.

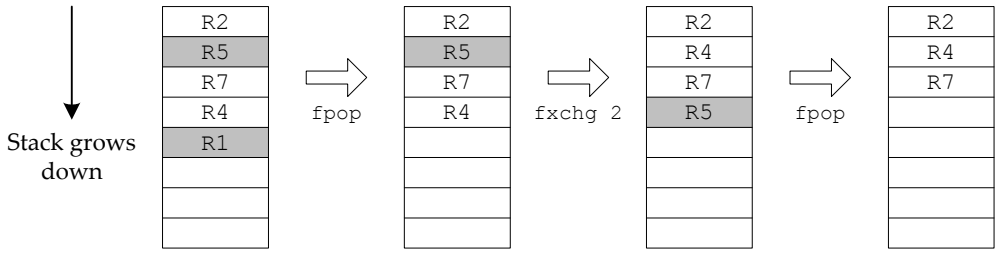


Figure 6.6: Example of stack cleanup

The current stack top `R1` is dead and is therefore popped without an exchange necessary. The next dead register is `R5` with the stack index 2, so it is exchanged with the stack top. This moves `R4` down to the old location of `R5`. Finally, the stack top `R5` can be popped from the stack and all dead registers are removed. The final code for cleanup is:

```
fpop
fxchg 2
fpop
```

This code is inserted at the end of `B2`. The stack without dead registers is saved as the initial state for the succeeding block `B4`. The simulation of `B4` could start now, even if the other predecessor was not processed yet.

6.3.4 Algorithm for Stack Merging

For the second and all further predecessors of a block, the stack at the end of the predecessor must be merged with the initial state set by the first predecessor. The following rules are applied until the stack is correct:

1. As long as the current stack top is not at the right location, i.e. it should not be on the stack top, it is exchanged with the right location.
2. If the stack top is correct, but the remaining stack is not ordered properly, then the stack top is exchanged with some other register that is not yet in place to get another value on the top. Then the algorithm continues with the first step.
3. If a dead value is on the stack top, it is popped from the stack.

Figure 6.7 shows the details for the cleanup at the end of `B3` in Figure 6.5. The left side shows the original stack before merging, the right side the result of the merging, which must be equal to the initial state of `B4` shown on the right side of Figure 6.6.

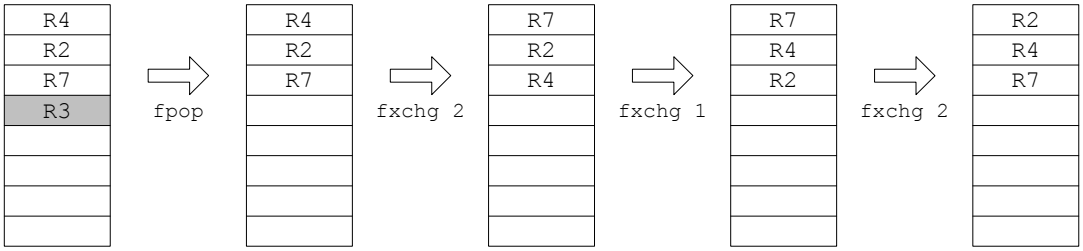


Figure 6.7: Example of stack merging

The register `R3` must be dead because it is not present in the initial state, so it is popped from the stack. The new stack top `R7` would be at the correct location, but the remaining stack is not correct. So `R7` must be exchanged with a register somewhere down the stack that is not yet in place. Assume that `R7` is exchanged with `R4`. Then `R4` can be put to its correct location by exchanging it with `R2`. After the last exchange of `R2` and `R7`, the stack matches the initial state and the merging is complete. The final code for cleanup is:

```
fpop
fxchg 2
fxchg 1
fxchg 2
```

This code is inserted at the end of `B3`. Exchange and pop operations inserted for stack merging are usually no performance bottleneck because the processor executes them very fast. In contrast to rounding registers, no memory access is necessary. Only the code size increases slightly.

6.4 Intel SSE2 Architecture

The SSE and SSE2 extensions of the IA-32 architecture were introduced with the Intel Pentium III and Pentium 4 processor, respectively. They allow the processor to perform single-instruction multiple-data (SIMD) operations for floating point operands. The execution unit works completely separated from the FPU. The instructions operate on 8 XMM registers, where each register has a width of 128 bits and can store four single-precision or two double-precision floating point values. The data types adhere to the IEEE 754 standard, so they also fulfill the Java specification.

The extensions are designed to work on two or four floating point values in parallel, but all instructions are also available in a scalar form. These instructions use only the low 32 or 64 bits of the XMM registers. All usual arithmetic instructions are available, so the scalar instructions of the SSE and SSE2 extensions can serve as a complete replacement for the FPU.

The compiler uses the SSE2 extensions to overcome the problems with the FPU stack. Because SSE2 instructions adhere to the IEEE 754 standard and the XMM registers are not stack-based, the special handling described in the previous parts of this chapter is not necessary. The XMM registers can be allocated with the same algorithms as the general-purpose registers.

Only the scalar versions of the instructions are used, no parallelization is performed. Nevertheless, floating point applications run faster when the SSE2 extensions are used

since no rounding is necessary. This is an obvious result of the benchmarks presented in the next chapter.

Nevertheless, the code generation algorithms for the FPU cannot be removed from the compiler because the SSE2 extensions are only available in modern processors, while the virtual machine must work on all Intel processors. But the high overhead of the FPU is alleviated by the fact that all new Intel processors implement the SSE2 extensions, so the percentage of systems where FPU stack allocation is necessary decreases.

Evaluation

This chapter evaluates the performance of the linear scan algorithm. Both the speed of the compilation and the speed of the generated code are measured and compared with the product version of the client compiler shipped with the Sun JDK 1.4.2. The compilation speed of linear scan is slightly lower compared to the product version, but the speed of the generated code is significantly higher: The speed of applications that profit largely from register allocation, such as parts of the SciMark 2.0 benchmark for numerical computations, can even double. But also the average speed of real-world applications, as measured with SPECjvm98, is about 30% higher compared to the JDK 1.4.2.

The original design goal of the Java HotSpot client compiler was to provide a high compilation speed at the cost of peak performance [Griesemer00]. This goal was achieved by omitting time-consuming optimizations. For this reason, the product version of the client compiler implements a heuristic for register allocation only. The main goals of this master thesis was the implementation of a global register allocation algorithm that leads to faster executing code, but without a significant compile time increase. The measurements of this chapter prove that this goal was achieved.

Additionally, the measurements show the large difference between the code using the FPU and the SSE2 extensions for floating point computations. Whereas numeric applications using the FPU show a speedup below average when the linear scan algorithm is compared with the old heuristic for register allocation, the speedup is above average when the SSE2 extensions are enabled.

7.1 Compared Configurations

To evaluate the quality of the linear scan algorithm, the research version of the Java HotSpot client compiler is compared with the product version of the client compiler shipped with the Sun JDK 1.4.2. In particular, the following three configurations are compared.

- The Java HotSpot client compiler of the Sun JDK 1.4.2_05, as described in Chapter 3.2.4 on page 23, using a local heuristic for register allocation.

- The research version of the client compiler with SSA form for the HIR and the linear scan algorithm for register allocation. The Intel FPU is used for floating point operations.
- The research version of the client compiler, using the SSE2 extensions for floating point operations.

The runtime library of the JDK 1.4.2_05 is used for all three compilers. All benchmarks are measured on an Intel Pentium 4 processor with 2.5 GHz, 512 KByte L2-Cache, 1 GByte of main memory and an Asus P4G8X motherboard with the Intel 7205 chipset, running Microsoft Windows XP Professional. The Pentium 4 processor implements the SSE2 extensions, so direct comparisons between FPU and SSE2 code are possible.

The upcoming Sun JDK 1.5 was available only as a beta version at the time when the benchmarks were run in August 2004, so the JDK 1.4.2 serves as the reference for all comparisons. Tests with the latest version JDK 1.5.0_beta2 did not show a big difference to the JDK 1.4.2_05, only some benchmarks were slightly faster.

The usage of SSE2 extensions can be configured with a startup flag of the research compiler: The flag `-XX:UseSSE=0` disables the extensions, the flag `-XX:UseSSE=2` enables them. To reduce the influence of garbage collection and memory management, the flags `-Xms800M -Xmx800M` of the HotSpot VM were used to fix a large heap size of 800 MB. No other flags were set, so the default values for all internal configuration parameters of the VM were used.

7.2 Compile Time

The compile time of methods can be easily measured via internal timers of the HotSpot virtual machine. To get a reasonable set of methods compiled, the SPECjvm98 benchmark was used. In a typical run, about 1200 methods are compiled (the exact number varies slightly because of a different inlining of methods). Table 7.1 summarizes some statistical data about the compilation. The numbers are accumulated over all compiled methods.

	JDK 1.4.2 Client	**Linear Scan**
Compiled bytes	232,918 bytes	240,083 bytes
Code size	843,232 bytes	987,087 bytes
Total size	2,192,390 bytes	2,370,319 bytes
Compilation time	1.143 sec.	1.273 sec.
Compilation speed	**204,329 bytes/sec.**	**189,750 bytes/sec.**

Table 7.1: Comparision of compile time

All of these numbers use the physical size of the methods in bytes, not the number of bytecodes (typical bytecodes have a size from one to three bytes). *Compiled bytes* accumulates the size of all methods compiled, i.e. the number of bytes parsed by the compiler. *Code size* is the size of the native code generated by the compiler. *Total size* is the total memory size allocated to store the compilation result. This number includes the *code size*, but is much bigger because it also includes the size of the meta data like debug information and oop maps that require a large amount of space. *Compilation time* is the total time spent in the compiler

The most important number is the *compilation speed*, calculated as the quotient of *compiled bytes* and *compilation time*. The higher the compilation speed is, the less time is spent in the compiler and the shorter are the pauses when a method is compiled. The compilation speed of the research compiler is only 7% lower than the speed of the product compiler.

7.2.1 Compilation Phases

Table 7.2 shows the distribution of the total compile time on the different phases of the research compiler. About half of the time is spent in the front end for HIR generation. This time includes parsing the bytecodes and optimizations performed on the HIR. The generation of the LIR from the HIR takes about 10% of the time, the generation of the native code from the LIR about 14%.

	Absolute	**Relative**
HIR Generation	0.592 sec.	46.5%
LIR Generation	0.130 sec.	10.2%
Linear Scan	0.341 sec.	26.8%
Code Generation	0.175 sec.	13.7%
Other	0.035 sec.	2.7%
Total	1.273 sec.	100.0%

Table 7.2: Distribution of total compile time

About one fourth of the total compilation time is spent in the linear scan algorithm. This is a considerable amount of time, but not unusually much. For example, the Java HotSpot server compiler, which is overall much slower than the client compiler, spends nearly half of the compilation time in its graph coloring register allocator [Paleczny01]. Table 7.3 breaks the time spent in the linear scan algorithm down into the basic steps of the algorithm, as presented in Algorithm 5.1 on page 52.

	Absolute	**Relative**
Number Operations	0.005 sec.	1.5%
Compute Local Live Sets	0.033 sec.	9.7%
Compute Global Live Sets	0.006 sec.	1.7%
Build Intervals	0.075 sec.	22.0%
Walk Intervals	0.096 sec.	28.1%
Resolve Data Flow	0.020 sec.	5.9%
Assign Register Numbers	0.027 sec.	7.9%
Construct Debug Information	0.059 sec.	17.3%
Other	0.020 sec.	5.9%
Total	0.341 sec.	100.0%

Table 7.3: Distribution of linear scan time

The three main parts of the algorithm require approximately one third of the total time:

- The pre-work for building the intervals, including the computation of the live sets and the data flow analysis.
- The actual allocation using the intervals, and the succeeding data flow resolution.
- The post-work for assigning the register numbers back to the LIR, including the construction of the debug information and the oop maps.

The parts of the algorithm that are known to be non-linear, such as the computation of the global live sets and the data flow resolution, require only a minor part of the total allocation time. So the asymptotic complexity of the implemented algorithm is higher than O(n), but nevertheless nearly linear in practice. This is also confirmed by the measurements of the next chapter.

7.2.2 Allocation Time for Large Methods

The time needed for compiling a method mainly depends on the size of the method in bytecodes. Most methods are very small and have a size less than 200 bytes. Only about 100 of the 1200 methods compiled during SPECjvm98 are larger. The number of LIR operations generated for a method does not only depend on the size of the method, but also on the size of inlined methods. The linear scan algorithm operates on the LIR, so the time needed for compilation is proportional to the number of LIR operations. Therefore, it is best to show the time used for register allocation depending on the number of LIR operations of the method. Figure 7.1 shows the 100 methods with the highest allocation time of all 1200 methods.

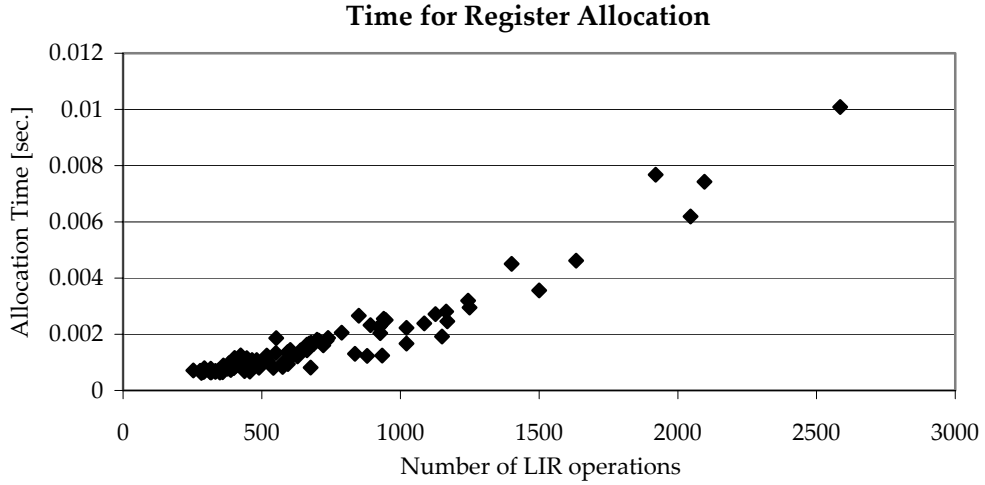


Figure 7.1: Time for register allocation—100 slowest methods out of 1200

The absolute compilation time is largely dependent on the system where the benchmark is run, so only the relative time is of interest. Figure 7.1 indicates that the linear scan algorithm nearly has a linear time behavior. It remains efficient even for large methods with many thousand LIR operations.

7.3 Run Time

The impact of the linear scan algorithm on the runtime of an application is difficult to measure. The total runtime of a Java application depends not only on the quality of the generated native code, but also on many other factors. A benchmark measures also the following components of the virtual machine:

- All classes used by a benchmark must be loaded by the class loader and verified before the execution starts.
- Before a method is compiled, it is executed in the interpreter for a while. Most methods are executed very infrequently and never get compiled. A good compilation policy ensures that the time spent in the interpreter is not significant for a long running benchmark.
- Occasional runs of the garbage collector are necessary to free unused objects. If the heap is full even after garbage collection, it must be enlarged. The overhead for memory management can be reduced by executing the benchmark with a large, fixed heap size.
- Several parts of the native code, e.g. the correct handling of exceptions, need the help of the virtual machine. The time spent in runtime calls does not depend on the quality of the native code generated by the JIT compiler.
- The I/O time for loading and storing files mainly depends on the speed of the external devices and the caches of the operating system.

Two benchmarks are used to evaluate the linear scan register allocation: SciMark 2.0 and SPECjvm98. Both benchmarks show a significant speedup, especially when the SSE2 extensions are enabled.

7.3.1 SciMark 2.0

SciMark 2.0, which is available for free at [SciMark2], is a benchmark for scientific and numerical computing. It executes and measures five computational kernels and reports a score in Mflops. The benchmark has the following characteristics:

- All kernels perform a large number of floating point operations.
- Each kernel consists only of one or two methods, executing long running but small nested loops.
- A slow start is performed, so all methods are already compiled for the finally measured run.
- No objects are allocated, so no garbage collection is necessary during the benchmark.
- No files are accessed; the kernels operate on randomly generated data.

Because of these characteristics, the benchmark is suitable for showing the difference between the Intel FPU and the SSE2 extensions. As presented in Chapter 6.2, native code for the FPU requires explicit rounding of floating point values by storing and reloading them to memory, so there is a big difference between the FPU and SSE2 code of the research compiler. The product version of the JDK 1.4.2 has no SSE2 support, so floating point operations are always executed in the FPU.

Table 7.4 and Figure 7.2 show the results of SciMark 2.0 for the three measured configurations. The relative numbers represent the speedup when the linear scan results are compared with the product compiler of the JDK 1.4.2.

	JDK 1.4.2 Client	Linear Scan FPU		Linear Scan SSE2	
		absolute	relative	absolute	relative
Fast Fourier Transformation (FFT)	84.1	156.8	1.87	226.9	2.70
Jacobi Succ. Over-relaxation (SOR)	354.5	344.5	0.97	383.1	1.08
Sparse matrix multiply (SMM)	131.1	198.8	1.52	281.5	2.15
Dense LU matrix factorization (LU)	383.3	393.7	1.03	491.0	1.28
Monte Carlo integration (MC)	44.5	49.1	1.10	44.5	1.00
Arithmetic Mean	199.5	228.6	**1.15**	285.4	**1.43**

Table 7.4: Results of SciMark 2.0 in Mflops (higher is better)

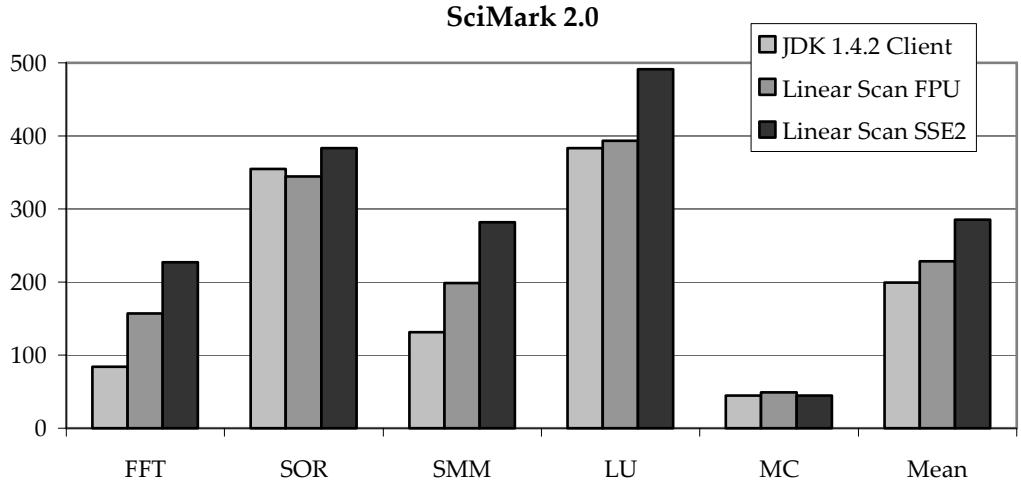


Figure 7.2: Results of SciMark 2.0 (higher is better)

The relative speedup of the five kernels varies from no improvement up to a nearly triple speed, so a closer look at the kernels is necessary:

- *FFT* performs a Fast Fourier Transformation on a one-dimensional array of complex numbers. This is the kernel with the highest speedup because it requires extensive computations where many temporary values are stored in local variables. The register allocator succeeds to put most local variables into registers, so the number of loads and stores to memory is greatly reduced.
- *SOR* operates on a two-dimensional grid of numbers. The innermost loop of this benchmark computes a new value for each grid point by averaging the four nearest neighbors. It operates directly on array elements without using local variables. Loading and storing array elements dominates the runtime, the register allocator is quite useless.
- *SMM* multiplies a sparse matrix with a vector. This is the second benchmark with a high speedup. Again most local variables can be held in registers.
- *LU* computes the factorization of a dense matrix. Only a moderate speedup is possible because of a high number of array accesses.
- *MC* approximates Pi using the Monte Carlo integration. Two random numbers are generated in each loop iteration. The random number generator is placed in its own

synchronized method, so the overhead for the method call and the synchronization countervails the optimizations of the register allocator.

The high speedup for the kernels *FFT* and *SMM* show the potential of a good register allocation algorithm for numerical computations. But the arithmetic mean of all five kernels is impressive too: The linear scan algorithm using the SSE2 extensions generates code running 43% faster than the product compiler of the JDK 1.4.2, without a significant increase of the compilation time.

The benchmark also shows the importance of the Intel SSE2 extensions for the Java programming language: The necessity for explicit rounding in the FPU mode prohibits a good register allocation for floating point numbers and leads to a significantly slower execution of numeric applications.

7.3.2 SPECjvm98

The SPECjvm98 benchmark [Spec98] is commonly used to assess the performance of Java runtime environments. It consists of seven programs derived from real-world applications that cover a broad range of scenarios where Java applications are deployed in real life. In contrast to SciMark 2.0, SPECjvm98 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 major change in the execution time occurs. A score is then calculated for the slowest and the fastest run, where a higher score is better.

- The slowest run is usually the first run of the benchmark where the classes must be loaded and execution starts in the interpreter. During the first run, the hot methods of the program are compiled, so the compilation time is also included in the slowest run. Therefore, this number is an indication of the startup speed of the JVM: A higher score of the slowest run denotes a faster startup of applications.
- The fastest run is usually the last run of the benchmark. All hot methods were already compiled during previous runs; the program has reached a fixpoint of execution time. This number measures the quality of the generated code: A higher score of the fastest run denotes a higher peak performance of the JVM.

The HotSpot client compiler is optimized for a fast startup, possibly at the cost of peak performance. So the difference between the slowest and the fastest run is usually very small. Table 7.5 shows the absolute results of SPECjvm98 for the three configurations. The relative numbers of Table 7.6 represent the speedup when comparing the linear scan results with the product compiler of the JDK 1.4.2. Figure 7.3 shows a diagram of the fastest runs for the three configurations.

The SPECjvm98 benchmark defines strict run rules that must be enforced for official results. These rules are negligibly violated by the configuration used because the benchmark was not run as an applet, but as a stand-alone application. The numbers given in Table 7.5 should not be compared with other published SPECjvm98 metrics. Nevertheless, the relative comparisons for a single system made in this chapter are correct.

	JDK 1.4.2 Client		Linear Scan FPU		Linear Scan SSE2	
	slowest	fastest	slowest	fastest	slowest	fastest
_227_mtrt	228.0	261.0	245.0	283.0	307.0	382.0
_202_jess	134.0	151.0	165.0	196.0	168.0	199.0
_201_compress	161.0	162.0	203.0	205.0	202.0	206.0
_209_db	33.2	35.0	34.4	36.3	33.0	36.9
_222_mpegaudio	193.0	203.0	205.0	216.0	327.0	354.0
_228_jack	162.0	183.0	180.0	211.0	180.0	211.0
_213_javac	76.4	95.4	83.4	109.0	83.7	110.0
Geometric Mean	121.0	**134.0**	136.0	**154.0**	150.0	**174.0**

Table 7.5: Absolute results of SPECjvm98 (higher is better)

	Linear Scan FPU		Linear Scan SSE2	
	slowest	fastest	slowest	fastest
_227_mtrt	1.07	1.08	1.35	1.46
_202_jess	1.23	1.30	1.25	1.32
_201_compress	1.26	1.27	1.25	1.27
_209_db	1.04	1.04	1.00	1.05
_222_mpegaudio	1.06	1.06	1.69	1.74
_228_jack	1.11	1.15	1.11	1.15
_213_javac	1.09	1.14	1.10	1.15
Geometric Mean	1.12	**1.15**	1.24	**1.30**

Table 7.6: Relative results of SPECjvm98 compared with JDK 1.4.2 Client

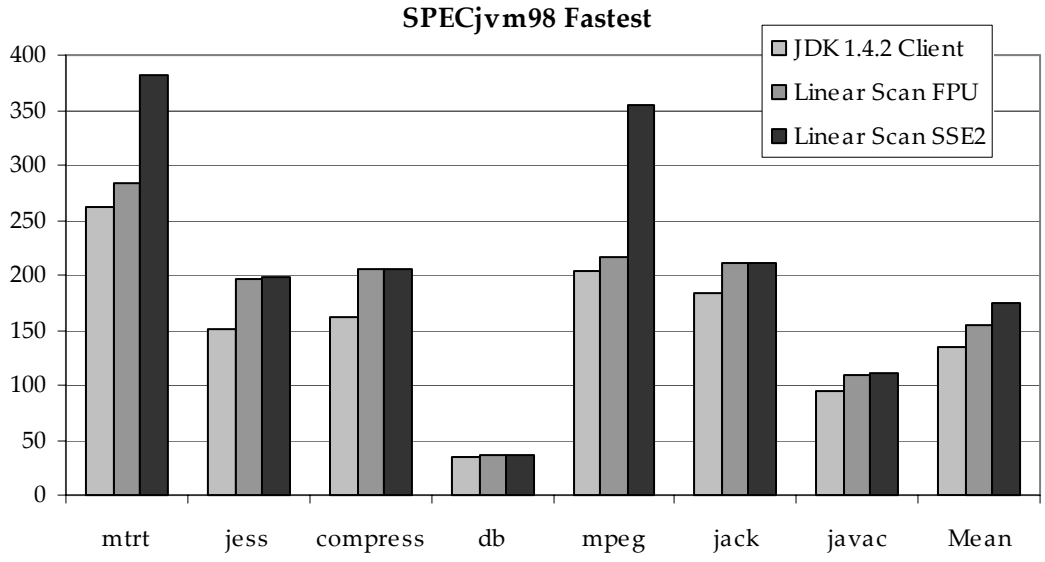


Figure 7.3: SPECjvm98 fastest run (higher is better)

The geometric mean can be considered as a realistic measure for the overall performance of a JVM. The speedup results of Table 7.6 are not synthetic numbers for special areas of applications, but represent the speedup that can be expected in everyday applications. The following results can be deduced from the table:

- The difference between the FPU and the SSE2 extensions affects not only numerical applications, but also common client applications like the decoding of mpeg audio files. The decision to extend the compiler to generate SSE2 code was therefore undoubtedly correct.
- Even on old processors with no SSE2 extensions, floating point applications are measurably faster. Applications with no floating point computations show an equally high speedup in both configurations.
- Linear scan outperforms the product compiler of the JDK 1.4.2 by 30%, i.e. the peak performance is 30% higher. This comes at no cost of the startup time because even the slowest run—which includes all compilations—is 24% faster.

The speedups of Table 7.6 correspond with the expected results based on the characteristics of the benchmark programs:

- *_227_mtrt* implements a ray-tracing algorithm that makes heavy use of floating point operations. This explains the modest speedup for FPU code and the high speedup for SSE2 code, similar to the results of SciMark 2.0.
- *_202_jess* is the Java Expert Shell System, which applies rules to a set of data to solve a set of puzzles. No floating point operations are performed, so the speed of FPU and SSE2 code is equal.
- *_201_compress* compresses and decompresses data using the LZW algorithm. It works only with integers and characters, so the speedup of FPU and SSE2 code is equally high.
- *_209_db* performs database functions on a memory-resident address database. The most time is spent in the sorting algorithm that offers no possibility for optimizations by linear scan. Additionally, all data is stored using the class Vector of the runtime library where all accesses are synchronized, which leads to a high overhead.
- *_222_mpegaudio* decompresses audio files that conform to the ISO MPEG Layer-3 audio specification. The transformation uses floating point operations, so there is a low speedup for the FPU code and a high speedup for the SSE2 code.
- *_228_jack* is a parser generator generating Java code from a specification file. The implementation makes heavy use of exception handling; exceptions are used for the modeling of the normal control flow. Exception handling is complicated by register allocation because local variables are not in fixed locations, but the benchmark shows that this is no performance bottleneck.
- *_213_javac* is the Java compiler from the JDK 1.0.2. It operates mostly on strings, so there is no difference between FPU and SSE2 code.

Summary

This final chapter summarizes the basic principles of the linear scan register allocator implemented for this master thesis. A short outlook on ongoing and planned future work of the project is given: Global optimizations that are known to be effective will be implemented. Sun Microsystems plans to integrate the research compiler in a future version of their JDK.

This master thesis presented the detailed algorithms of a register allocator using the linear scan algorithm. The algorithm is implemented in a research version of the Java HotSpot client compiler of Sun Microsystems. The research compiler is the result of a long lasting and ongoing research collaboration between Sun Microsystems and the Institute for System Software at the Johannes Kepler University Linz.

Compared with register allocation algorithms based on graph coloring, the linear scan algorithm is much simpler and faster. It is capable of allocating lifetime intervals in a single linear pass over all intervals. The simplest form of the algorithm has the drawback that it does not support lifetime holes of intervals. Also each interval is fixed to one location, either in a register or on the stack. Therefore, the algorithm implemented for this thesis is extended to support holes in intervals and splitting of intervals. When an interval should change its location, it is split into two independent intervals.

This optimization allows a much better utilization of the register set, especially for architectures with a low number of registers such as the Intel IA-32 architecture. This architecture additionally complicates the work of a register allocation by requiring fixed registers for some instructions and a complicated structure of the floating point registers.

The flexible algorithm for splitting intervals is one of the main results of this thesis. An interval can be split anywhere by the register allocator if this seems to be good for the overall performance of the method. The splitting position can be moved to the optimal position out of loops to minimize the spill moves in frequently executed parts of the method. The actual splitting position is calculated by a heuristic taking many parameters into account. The correct parameterization of the heuristic is crucial for the overall performance of the generated code.

Register constraints of the IA-32 architecture are modeled by fixed intervals, representing ranges where a physical register is not available for normal allocation. The allocation algorithm needs not handle operations that require operands in fixed registers, method calls that destroy all registers or other operations that have special demands on registers.

This simplifies the allocation algorithm considerably and avoids platform-dependent code in the allocator. Especially the automatic handling of calls is worth mentioning: At a call operation, all registers are marked as blocked by a short range in all fixed intervals. Therefore, the allocation algorithm cannot assign a physical register to any non-fixed interval at this position, and all values that are live at the call site are automatically spilled to the stack.

The implementation of the linear scan algorithm required some changes in other parts of the compiler, mostly because of the fact that a local variable has no fixed location on the stack any more, but remains in a register if possible:

- The computation of the debug information and oop maps, necessary for deoptimization and garbage collection, respectively, must be done during register allocation. Previously, this work was done when the native code was generated.
- The correct handling of exceptions is somewhat more complicated now because arbitrary registers can be in use when an exception is thrown. These registers must be preserved when the exception handler is searched. Currently, the debug information is used to match up the state of the local variables at the throwing instruction and the begin of the exception handler.

The benchmark results show the high capabilities of the linear scan algorithm: While the compilation is only 7% slower when compared to the product version of the JDK 1.4.2, the average application speed is about 30% better. So the linear scan algorithm is able to allocate the registers nearly as fast as the old local allocator, which uses only a simple heuristic for the optimization of loops, but the generated code is much better, i.e. the register allocator succeeds to reduce the number of moves from and to memory significantly.

8.1 Future Work

The research compiler extends the product compiler of the JDK 1.4.2 with an intermediate representation in static single assignment (SSA) form. This form allows the complete elimination of instructions accessing local variables, but requires phi functions when a block has more than one predecessor. The SSA form simplifies many optimizations, such as common subexpression elimination. This potential is currently unused.

The goal of the ongoing project is the implementation of global optimizations that can be applied fast and that are known to be effective in increasing the application speed. The first optimization will be the implementation of global common subexpression elimination, which is considerably simplified by the SSA form of the intermediate representation.

Another field of research in the project deals with escape analysis for the compiler: In his PhD thesis, Thomas Kotzmann implements an analysis that detects allocation sites that do not escape, i.e. the allocated objects are not installed into static fields or heap objects and are not returned to the caller. The following optimizations are currently under development:

- When an object is local to a single method, the allocation can be completely eliminated (scalar replacement). The fields of the object are treated equally to local variables; they are initially assigned a virtual register and then processed normally by the register allocator.
- An object passed as a parameter to another method can be allocated on the stack when it does not escape in the called method. The elimination of non-escaping allocations frees the garbage collector from processing many short living objects, such as temporary string buffers.
- When methods are synchronized on non-escaping objects, the synchronization is also unnecessary and can be safely removed.

The research compiler has meanwhile reached a very stable state. Many stress tests that are implemented in the virtual machine were successfully executed. So another goal is to bring the compiler to product quality that allows the deployment in the product version of a future version of the JDK. Sun Microsystems plans a possible replacement of the current client compiler with the research compiler, using the linear scan algorithm for register allocation.

Compilation Example

This chapter presents a complete example for the compilation of a method. First, the Java source code, the Java bytecodes and the two intermediate representations HIR and LIR are presented. Then, all steps necessary for linear scan register allocation are shown in detail. The chapter closes with the native code that is ready for execution.

The example described in this chapter calculates and prints all Fibonacci numbers below 10,000. The numbers are summed up and printed in a loop. Figure A.1 shows the Java source code of the calculation method. The standard iterative algorithm is used that saves the last but one number (called `lo`) and the last number (called `hi`) in local variables.

```
public static void fibonacci() {
  int lo = 0;
  int hi = 1;
  while (hi < 10000) {
    hi = hi + lo;
    lo = hi - lo;
    print(lo);
  }
}
```

Figure A.1: Java source code

Assume that the method `print` that is called in each iteration is a `static` method of the same class just prints the number and a trailing space to the standard output stream. Since this requires several other method calls that would complicate the example too much, the method `print` is not considered any more. Especially, it is not inlined during compilation. The following output is generated by the method:

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

Figure A.2 shows the machine-independent, stack-based bytecodes created by the Java compiler for this method. The bytecodes refer to the local variables by numbers: The local variable `lo` has the index 0, `hi` the index 1. The while loop is compiled to a conditional branch for the loop condition at the bytecode index (bci) 8 and an unconditional jump at the bci 23.

```
0:  iconst_0
1:  istore_0            // lo = 0
2:  iconst_1
3:  istore_1            // hi = 1
4:  iload_1
5:  sipush 10000
8:  if_icmpge 26        // while (hi < 10000)
11: iload_1
12: iload_0
13: iadd
14: istore_1            // hi = hi + lo
15: iload_1
16: iload_0
17: isub
18: istore_0            // lo = hi - lo
19: iload_0
20: invokestatic #12    // print(lo)
23: goto 4              // end of while-loop
26: return
```

Figure A.2: Java bytecodes

A.1 HIR

The bytecodes are the main input of the just-in-time compiler when the virtual machine decides to compile this method. The front end of the compiler generates the HIR by iterating through the bytecodes twice. In the first iteration, the boundaries of the basic blocks are determined. The following blocks are identified (the numbering of the blocks is arbitrary):

- `B0` (bci 0-3): Block before the loop that initializes the local variables `lo` and `hi`.
- `B3` (bci 4-8): Header block of the loop that checks the loop condition.
- `B2` (bci 11-23): Body of the loop that calculates the next number and calls `print`.
- `B1` (bci 26): Block that contains only the `return` bytecode.

Additionally, the compiler generates the header block `B4` for technical reasons. This block has no representation in the bytecodes. The second iteration of the bytecodes fills the blocks with HIR instructions. Figure A.3 shows the complete HIR: The first line of each block represents the BlockBegin instruction. The bytecode range and the predecessors and successors of the block are printed. The following lines represent the HIR instructions.

The instructions `i7` and `i8` are phi functions of block `B3` for the local variables `lo` and `hi`, respectively. They are necessary because `B3` has two predecessors. The instruction `i7` gets the value `i4` when Block `B0` was executed before block `B3` (i.e. for the first iteration of the loop) and the value `i12` when `B2` was executed before (i.e. for all other iterations). The instruction `i8` gets the value `i5` or `i11`, respectively.

```
B4 [0, 0] sux: B0
__bci__use__tid____instr_____________________________________
. 0    0     17    std entry B0

B0 [0, 3] pred: B4 sux: B3
__bci__use__tid____instr_____________________________________
. 0    1    i4     0
. 2    1    i5     1
. 3    0     6     goto B3

B3 [4, 8] pred: B0 B2 sux: B1 B2
Locals:
   0: i7 [ i4 i12 ]     // phi function for lo
   1: i8 [ i5 i11 ]     // phi function for hi
__bci__use__tid____instr_____________________________________
  5    1    i9     10000
. 8    0     10    if i8 >= i9 then B1 else B2

B2 [11, 23] pred: B3 sux: B3
__bci__use__tid____instr_____________________________________
. 13   2    i11    i8 + i7
. 17   2    i12    i11 - i7
. 20   0    v13    invokestatic(i12)
. 23   0     14    goto B3 (safepoint)

B1 [26, 26] pred: B3
__bci__use__tid____instr_____________________________________
. 26   0    v15    return
```

Figure A.3: HIR

A.2 LIR before Register Allocation

The back end of the compiler generates the LIR from the HIR. Figure A.4 on the next page shows the LIR before register allocation where most operands of the LIR operations are virtual registers. The structure of the basic blocks is equivalent to the HIR, so the first line of each block is equal to Figure A.3. The following lines represent the LIR operations.

Each block starts with a label that is used as the target for branches to this block. The last operation of each block is always an unconditional jump to a successor or a return operation because otherwise the control flow at the end of the block would be undefined. Most LIR operations are a direct result of the HIR instructions. Only the move operations are inserted for special reasons:

- The moves with the id 8, 10, 36 and 38 are resolving moves for the phi functions of the HIR.
- The moves with the id 24 and 28 are inserted because of the two-operand form required by the IA-32 architecture. For succeeding arithmetic operations, the left input operand and the result operand are equal.
- The move with the id 32 stores the parameter for the method call to the appropriate stack slot.

```
B4 [0, 0] sux: B0
__id__Operation_____________________________________________
   0  label [label:0x31a8904]
   2  std_entry [ecx|L]
   4  branch [AL] [B0]

B0 [0, 3] pred: B4 sux: B3
__id__Operation_____________________________________________
   6  label [label:0x978b8c]
   8  move [int:1|I] [R42|I]
  10  move [int:0|I] [R41|I]
  12  branch [AL] [B3]

B3 [4, 8] pred: B0 B2 sux: B1 B2
__id__Operation_____________________________________________
  14  label [label:0x31a81d4]
  16  cmp [R42|I] [int:10000|I]
  18  branch [GE] [B1]
  20  branch [AL] [B2]

B2 [11, 23] pred: B3 sux: B3
__id__Operation_____________________________________________
  22  label [label:0x31a80fc]
  24  move [R42|I] [R43|I]
  26  add  [R43|I] [R41|I] [R43|I]
  28  move [R43|I] [R44|I]
  30  sub  [R44|I] [R41|I] [R44|I]
  32  move [R44|I] [Base:[esp|I] Disp: 0|]
  34  static call: [bci:20]
  36  move [R43|I] [R42|I]
  38  move [R44|I] [R41|I]
  40  safepoint [bci:23]
  42  branch [AL] [B3]

B1 [26, 26] pred: B3
__id__Operation_____________________________________________
  44  label [label:0x978c64]
  46  return
```

Figure A.4: LIR before register allocation

A.3 Block Order

Before computing the final block order, the loops of the method are searched. The loop detection algorithm identifies block `B2` as the only loop end block of the loop with index 0 starting at block `B3`. The remaining blocks `B4`, `B0` and `B1` are not part of a loop.

The options for the block order algorithm are very limited: `B4`, `B0` and `B3` must be arranged in this order because of their sequential control flow. Only the order of `B1` and `B2` is not fixed by the control flow. `B2` is emitted before `B1` because it has a higher loop depth and therefore a higher weight. The final block order is `B4`, `B0`, `B3`, `B2`, `B1`. Note that the two loop blocks `B3` and `B2` are consecutive. The upper part of Figure A.5 shows the control flow graph with this block order.

A.4 Building Intervals

Figure A.5 shows the lifetime intervals for the example. Each virtual register of the LIR is represented by its own line. In order to deal with the fact that the call operation 34 destroys all registers, short ranges are added to all fixed intervals. For each physical register (the general-purpose registers `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi` and the 8 FPU registers), a fixed interval is created. Because all fixed intervals have exactly the same lifetime in this example, only one line is printed for all fixed intervals. The intervals 41 and 42 have a lifetime hole in `B2`, e.g. the virtual register `[R42]` is overwritten by the operation 36, so its value is not required between operation 24 and 36.

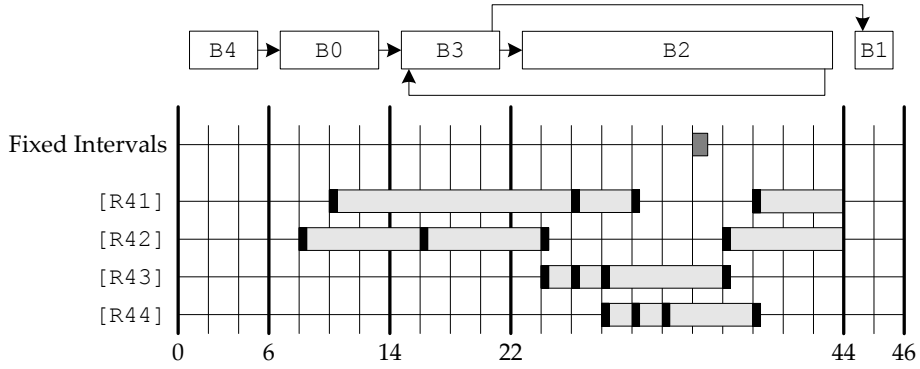


Figure A.5: Intervals before register allocation

The following shading of boxes is used for printing intervals:

- Unhandled intervals are printed as light grey bars. All non-fixed intervals of Figure A.5 are initially unhandled.
- Already processed intervals with a register assigned are printed as medium grey bars.
- Processed intervals that were spilled to memory are printed as dark grey bars.

A.5 Walking Intervals

The actual register allocation assigns a physical register to each interval. Because all intervals store integer values, one of the six general-purpose registers must be assigned to each interval. No floating point registers are used by the example.

The non-fixed intervals are sorted by increasing start position and traversed in this order. The unhandled set contains all intervals that were not processed yet. In each iteration, the interval with the lowest start position is removed from the unhandled set and processed. This interval is the current interval. The active and the inactive sets contain all intervals that have already a register assigned and that do not end before the start position of current.

The following sections show the content of the unhandled, active and inactive sets when the intervals are processed. The inactive intervals are further classified into the inactive intervals intersecting with current—these intervals are relevant for the allocation—and the inactive intervals not intersecting with current—these can be ignored.

A.5.1 Interval 42

- current: interval 42, starting at position 8
- unhandled: { 41, 43, 44 }
- active: { }
- inactive intersecting with current: { }
- inactive not intersecting with current: { all fixed intervals }

The active set is empty, and the fixed intervals of the inactive set do not intersect with current because current has a lifetime hole at the call operation with the id 34. Therefore, all registers are available for the whole lifetime of current. Assume `esi` is selected for allocation. Then interval 42 is added to the active set. Figure A.6 shows the intervals with `esi` assigned to `[R42]`.

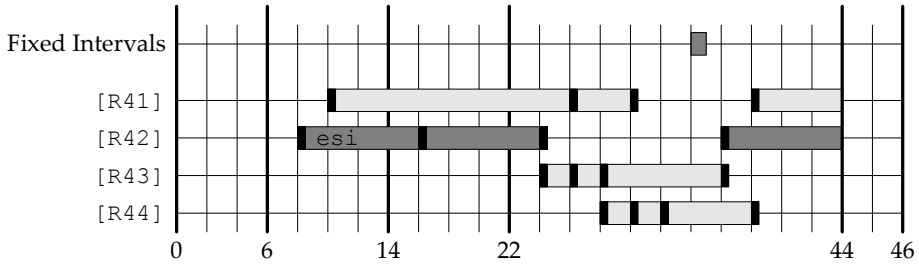


Figure A.6: Intervals after processing of interval 42

A.5.2 Interval 41

- current: interval 41, starting at position 10
- unhandled: { 43, 44 }
- active: { 42 }
- inactive intersecting with current: { }
- inactive not intersecting with current: { all fixed intervals }

Similarly to interval 42, current does not intersect with the fixed intervals. The register `esi` is blocked because interval 42 is active. All other registers are available for the whole lifetime of current. Assume `edi` is selected for allocation. Then interval 41 is added to the active set. Figure A.7 shows the intervals after processing of interval 41.

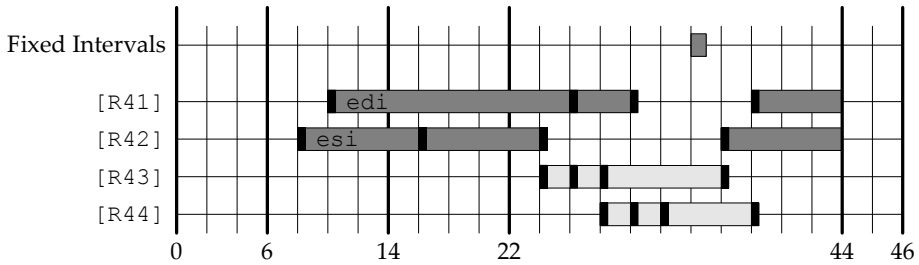


Figure A.7: Intervals after processing of interval 41

A.5.3 Interval 43

- current: interval 43, starting at position 24
- unhandled: { 44 }
- active: { 41 }
- inactive intersecting with current: { all fixed intervals }
- inactive not intersecting with current: { 42 }

Before the current interval 43 is processed, interval 42 is moved from the active set to the inactive set because of its lifetime hole starting at position 24. The interval 41 is still active. The fixed intervals intersect with current, so no register is available for the whole lifetime of current and current must be split. The splitting position cannot be moved to a block boundary or out of the loop, so current is split at position 33 before the call.

A new interval with the number 45 is created for the split child starting at position 33. This interval is sorted into the unhandled set and processed later when position 33 is reached by the allocator. The current interval 43 is shorter now and ends before the call, so a register is available. It gets the register `esi` assigned and is added to the active set.

Figure A.8 shows a snapshot of the intervals after processing interval 43: The intervals 41 and 42 have a register assigned for their whole lifetime. Interval 43 was split at position 33, so the part before 33 (interval number 43) has a register assigned, while the part after 33 (interval number 45) is still unhandled. Both intervals are printed in the same line of `[R43]` to emphasize that they represent the lifetime of the virtual register `[R43]` together. There is no virtual register `[R45]` present in the LIR. Interval 44 is still completely unhandled.

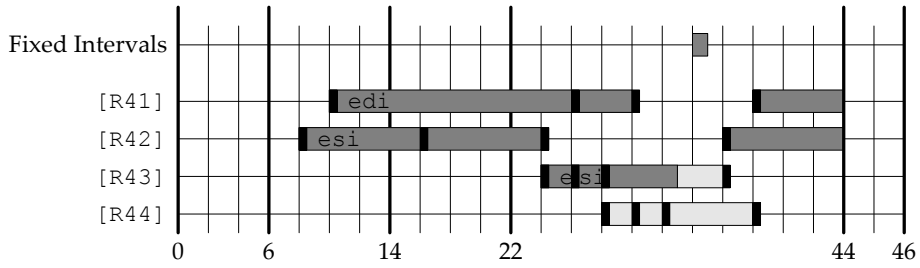


Figure A.8: Intervals after processing of interval 43

A.5.4 Interval 44

- current: 44, starting at position 28
- unhandled: { 45 }
- active: { 41, 43 }
- inactive intersecting with current: { 42, all fixed intervals }
- inactive not intersecting with current: { }

This interval is processed similarly to interval 43: Because it intersects with the fixed intervals, it is split at position 33. The first part gets `ebx` assigned because `esi` and `edi` are still blocked by the active intervals 41 and 43. A new interval with the number 45 is created for the split child starting at position 33. This interval is sorted into the unhandled set.

Figure A.9 shows the intervals after processing of interval 44: Interval 44 was split at position 33, so the part before 33 (interval number 44) has a register assigned, while the part after 33 (interval number 46) is still unhandled. Again, both intervals are printed in the line of the virtual register `[R44]`.

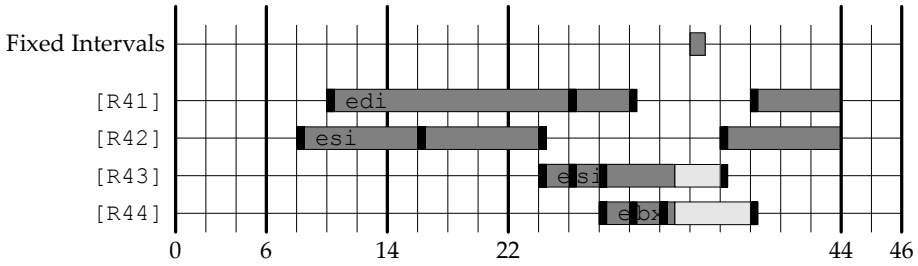


Figure A.9: Intervals after processing of interval 44

A.5.5 Interval 45

- current: 45, starting at position 33
- unhandled: { 46 }
- active: { }
- inactive intersecting with current: { all fixed intervals }
- inactive not intersecting with current: { 41, 42 }

This interval is the split child of the original interval 43. All registers are blocked by the fixed intervals at position 34, so no register is available for allocation and an interval must be spilled to the stack. Because fixed intervals must never be spilled, the only candidate for spilling is the current interval 45 itself. So current gets a new spill slot assigned, called `[stack:0]`. A move operation from the register `esi` to the spill slot is inserted into the LIR at position 33.

The use position at id 36 does not require a register because the interval is used as the input parameter of a move at id 36. The value needs not be reloaded in a register and interval 45 remains on the stack for its entire lifetime. No further splitting is necessary. Figure A.10 shows the intervals after spilling interval 45, representing the right part of the virtual register `[R43]`.

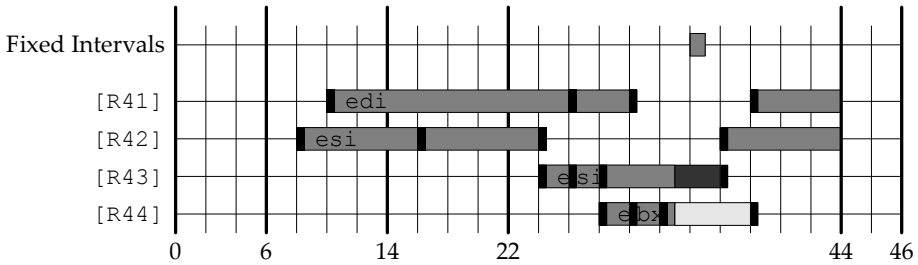


Figure A.10: Intervals after processing of interval 45

A.5.6 Interval 46

- current: 46, starting at position 33
- unhandled: { }
- active: { }
- inactive intersecting with current: { 42, all fixed intervals }
- inactive not intersecting with current: { 41 }

Similarly to interval 45, this interval is spilled to the stack because all registers are blocked by the fixed intervals. The use position at the id 38 requires no register, so the entire interval 45 is spilled to the spill slot `[stack:1]` and no further splitting is necessary. Another move operation is inserted into the LIR at position 33.

Now the unhandled set is empty, all intervals are processed and the algorithm stops. All intervals have either a register assigned or are spilled to the stack. Figure A.11 shows the intervals after register allocation. No unhandled intervals are present any more.

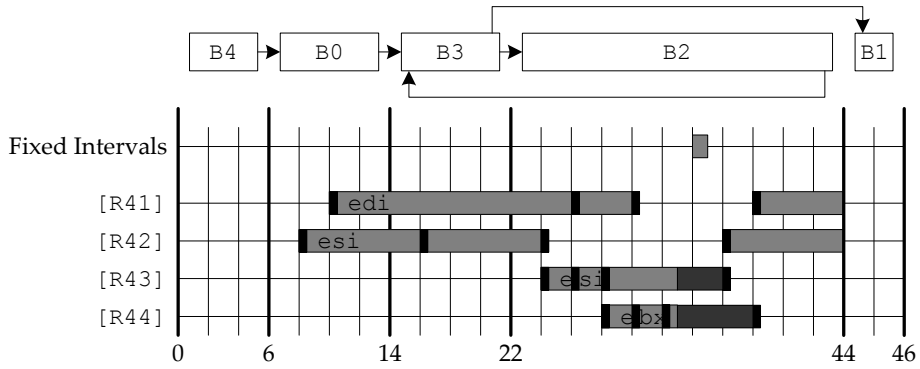


Figure A.11: Intervals after register allocation

It is easy to see that the allocation was done properly:

- Intervals with intersecting ranges have different registers assigned. Only the intervals 42 and 43 have the same register assigned because interval 43 fits into the lifetime hole of interval 42.
- All intervals that are live at the call operation with the id 33 are spilled. No value is destroyed when the called method overwrites the registers.

A.6 LIR after Register Allocation

No resolving of the data flow is necessary in this simple example: The intervals 41 and 42 that span multiple blocks are not split, so no additional move operations must be inserted into the LIR.

Figure A.12 shows the LIR where the virtual registers are replaced with the allocated physical registers. The following operations and operands are different when compared with the original LIR showed in Figure A.4.

- The virtual registers `[R41]` and `[R42]` are replaced by the physical register `esi` and `edi`, respectively.
- The virtual register `[R43]` is replaced with the physical register `esi`. Only the input operand of the operation 36 is replaced with the spill slot `[stack:0]` because this operation is covered by the split child 45.
- The virtual register `[R44]` is replaced with the physical register `ebx`. Only the input operand of the operation 38 is replaced with the spill slot `[stack:1]` because this operation is covered by the split child 46.
- Two new move operations are inserted at position 33. They save the registers `ebx` and `esi` before the method call to the stack slots `[stack:1]` and `[stack:0]`, respectively.
- The move operation at position 24 is deleted because the source operand `[R42]` and the target operand `[R43]` are allocated to the same register `esi`.

```
B4 [0, 0] sux: B0
__id__Operation_____________________________________________
   0  label [label:0x31a8904]
   2  std_entry [ecx|L]
   4  branch [AL] [B0]

B0 [0, 3] pred: B4 sux: B3
__id__Operation_____________________________________________
   6  label [label:0x978b6c]
   8  move [int:1|I] [esi|I]
  10  move [int:0|I] [edi|I]
  12  branch [AL] [B3]

B3 [4, 8] pred: B0 B2 sux: B1 B2
__id__Operation_____________________________________________
  14  label [label:0x31a81d4]
  16  cmp [esi|I] [int:10000|I]
  18  branch [GE] [B1]
  20  branch [AL] [B2]

B2 [11, 23] pred: B3 sux: B3
__id__Operation_____________________________________________
  22  label [label:0x31a80fc]
  26  add  [esi|I] [edi|I] [esi|I]
  28  move [esi|I] [ebx|I]
  30  sub  [ebx|I] [edi|I] [ebx|I]
  32  move [ebx|I] [Base:[esp|I] Disp: 0|]
  33  move [ebx|I] [stack:1|I]
  33  move [esi|I] [stack:0|I]
  34  static call: [bci:20]
  36  move [stack:0|I] [esi|I]
  38  move [stack:1|I] [edi|I]
  40  safepoint [bci:23]
  42  branch [AL] [B3]

B1 [26, 26] pred: B3
__id__Operation_____________________________________________
  44  label [label:0x978c44]
  46  return
```

Figure A.12: LIR after register allocation

A.7 Code Generation

The native code can be generated in a straightforward way from the LIR. In this example, all moves and arithmetic operations are converted to a single native instruction. The other operations are converted as follows:

- Labels are needed only for marking the target of jumps, so no native instruction is necessary for them.
- Unconditional jumps between succeeding blocks are unnecessary, so they are omitted. In this example, the jumps with the id 4, 12 and 20 are unnecessary.
- Spill slots are addressed relative to the base pointer `ebp`.
- The loop header is aligned at a 4-byte boundary for performance reasons.
- The safepoint operation is translated to a native instruction accessing a special, fixed memory address. This address is used by the runtime to stop the thread for garbage collection.
- The entry code for the method builds the stack frame by modifying `esp` and `ebp`. The very first instruction of the method checks for possible stack overflows in the near future.
- The return code removes the stack frame. Before the actual return instruction, another safepoint is inserted.

The native code for the method is shown in Figure A.13. The usual syntax for IA-32 assembler code is used, i.e. the target of moves and the result of arithmetic instructions is always the leftmost operand. This native code is then installed in the code cache of the virtual machine and is ready for execution.

```
00000000: mov   dword ptr [esp-3000h], eax
00000007: push  ebp
00000008: mov   ebp, esp
0000000a: sub   esp, 18h
0000000d: mov   esi, 1h
00000012: mov   edi, 0h
00000017: nop
00000018: cmp   esi, 2710h
0000001e: jge   00000049
00000024: add   esi, edi
00000026: mov   ebx, esi
00000028: sub   ebx, edi
0000002a: mov   dword ptr [esp], ebx
0000002d: mov   dword ptr [ebp-8h], ebx
00000030: mov   dword ptr [ebp-4h], esi
00000033: call  00a50d40
00000038: mov   esi, dword ptr [ebp-4h]
0000003b: mov   edi, dword ptr [ebp-8h]
0000003e: test  dword ptr [370000h], eax
00000044: jmp   00000018
00000049: mov   esp, ebp
0000004b: pop   ebp
0000004c: test  dword ptr [370000h], eax
00000052: ret
```

Figure A.13: Native code

List of Figures

List of Tables

List of Algorithms

Literature

[AdlTabatabai98] Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, James M. Stichnoth: *Fast, effective code generation in a just-in-time Java compiler*. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pp 280-290. ACM Press, 1998.
DOI: 10.1145/277650.277740

[Agesen99] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, Derek White: *An efficient meta-lock for implementing ubiquitous synchronization*. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp 207-222. ACM Press, 1999.
DOI: 10.1145/320384.320402

[Aho86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: *Compilers: principles, techniques and tools*. Addison-Wesley, 1986.

[Briggs89] Preston Briggs, Keith D. Cooper, Ken Kennedy, Linda Torczon: *Coloring heuristics for register allocation*. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pp 275-284. ACM Press, 1989.
DOI: 10.1145/73141.74843

[Burke99] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, John Whaley: *The Jalapeño dynamic optimizing compiler for Java*. In *Proceedings of the ACM 1999 conference on Java Grande*, pp 129-141. ACM Press, 1999.
DOI: 10.1145/304065.304113

[Chaitin81] Gregory J. Chaitin, Marc A. Auslander, Ashok. K. Chandra, John Cocke, Martin. E. Hopkins, Peter W. Markstein: *Register allocation via coloring*. In *Computer Languages, Volume 6 (1981)*, pp 47-57. Elsevier Ltd, 1981.
DOI: 10.1016/0096-0551(81)90048-5

[Chaitin82] Gregory J. Chaitin: *Register allocation & spilling via graph coloring*. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pp 98-101. ACM Press, 1982.

[Click95] Cliff Click, Michael Paleczny, *A simple graph-based intermediate representation*. In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, pp 35-49. ACM Press, 1995.
DOI: 10.1145/202529.202534

[Click02] Cliff Click, John Rose: *Fast subtype checking in the HotSpot JVM*. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pp 96-107. ACM Press, 2002.
DOI: 10.1145/583810.583821

[Cytron91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck: *Efficiently computing static single assignment form and the control dependence graph*. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Volume 13, Issue 4, pp 451-490. ACM Press, 1999.
DOI: 10.1145/115372.115320

[Farach98] Martin Farach, Vincenzo Liberatore: *On local register allocation*. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pp 564-573. Society for Industrial and Applied Mathematics, 1998.

[Gosling00] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[Griesemer99] Robert Griesemer: *Generation of virtual machine code at startup*. In *Proceedings of the OOPSLA'99 Workshop on Simplicity, Performance, and Portability in Virtual Machine Design*, 1999.

[Griesemer00] Robert Griesemer, Srdjan Mitrovic: *A Compiler for the Java HotSpot Virtual Machine*. In László Böszörményi, Jürg Gutknecht, Gustav Pomberger (editors): *The School of Niklaus Wirth: The Art of Simplicity*, pp 133-152. dpunkt.verlag, 2000.

[Hölzle91] Urs Hölzle, Craig Chambers, David Ungar: *Optimizing dynamically-typed object-oriented languages with polymorphic inline caches*. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91)*. Lecture Notes in Computer Science, Volume 512, pp 21-38. Springer-Verlag, 1991.

[Hölzle92] Urs Hölzle, Craig Chambers, David Ungar: *Debugging optimized code with dynamic deoptimization*. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pp 32-43. ACM Press, 1992.
DOI: 10.1145/143095.143114

[IEEE754] Institute of Electrical and Electronics Engineers: *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Standard 754-1985.

[Intel1] Intel Corporation: *The IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*. Order Number: 253665, 2004.

[Intel2A] Intel Corporation: *The IA-32 Intel Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M*. Order Number 253666, 2004.

[Intel2B] Intel Corporation: *The IA-32 Intel Architecture Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z*. Order Number 253667, 2004.

[Johansson02] Erik Johansson, Konstantinos Sagonas: *Linear Scan Register Allocation in a High-Performance Erlang Compiler*. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002)*. Lecture Notes in Computer Science, Volume 2257, pp 101-119. Springer-Verlag, 2002.

[Kotzmann02] Thomas Kotzmann: *Ein Just-in-Time-Compiler für Java*. Master thesis, Institute for Practical Computer Science, Johannes Kepler University Linz, 2002.

[Lindholm99] Tim Lindholm, Frank Yellin: *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[Mössenböck00] Hanspeter 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.

[Mössenböck02] Hanspeter Mössenböck, Michael Pfeiffer: *Linear scan register allocation in the context of SSA form and register constraints*. In *Proceedings of the International Conference on Compiler Construction (CC'02)*. Lecture Notes in Computer Science, Volume 2304, pp 229-246. Springer-Verlag, 2002.

[Muchnick97] Steven S. Muchnick: *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[Paleczny01] Michael Paleczny, Christopher Vick, Cliff Click: *The Java HotSpot Server Compiler*. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, 2001.

[Pelegri88] Eduardo Pelegrí-Llopart, Susan. L. Graham: *Optimal code generation for expression trees: an application BURS theory*. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp 294-308. ACM Press, 1988.
DOI: 10.1145/73560.73586

[Poletto97] Massimiliano Poletto, Dawson R. Engler, M. Frans 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*, pp 109-121. ACM Press, 1997.
DOI: 10.1145/258915.258926

[Poletto99] Massimiliano Poletto, Vivek Sarkar: *Linear scan register allocation*. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Volume 21, Issue 5, pp 895-913. ACM Press, 1999.
DOI: 10.1145/330249.330250

[SciMark2] Roldan Pozo, Bruce Miller: *SciMark 2.0*.
http://math.nist.gov/scimark2/

[Sethi73] Ravi Sethi*: Complete register allocation problems*. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pp 182-195. ACM Press, 1973.

[Spec98] Standard Performance Evaluation Corporation: *SPECjvm98*. http://www.spec.org/jvm98/

[Sun02] Sun Microsystems, Inc.: *The Java HotSpot Virtual Machine, v1.4.1*. Technical White Paper, 2002.

[Sun03] Sun Microsystems, Inc.: *Tuning Garbage Collection with the 1.4.2 Java Virtual Machine*. Documentation, 2003.

[Traub98] Omri Traub, Glenn Holloway, Michael D. Smith: *Quality and speed in linear-scan register allocation*. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pp 142-151. ACM Press, 1998.
DOI: 10.1145/277650.277714

When a digital object identifier (DOI) is available for a paper, additional information can be retrieved at the URL http://dx.doi.org/*[DOI]*

Curriculum Vitae

Personal Data:

Name:	Christian Wimmer
Date of birth:	June 12, 1981
Family status:	single
Citizenship:	Austrian
Parents:	Ing. Maximilian Wimmer Anna Wimmer
Siblings:	Dipl.-Ing. Peter Wimmer Isabella Wimmer

Education:

1987 – 1991	Primary school
1991 – 1999	Secondary school
June 1999	Matura passed with distinction
1999 – 2000	Military service
since 2000	Study of Computer Science Johannes Kepler University Linz
July 2002	First part of final examination passed with distinction