Mag.rer.soc.oec. Dr.techn. Stefan Rath Bakk.techn.

MIP Models for SSA-Form-Based Register Allocation in the Java HotSpot™ Server Compiler

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Masterstudium Telematik

eingereicht an der

Technischen Universität Graz

Betreuer

Ass.-Prof. Dipl.-Ing. Dr.techn. Christian Steger
Institut für Technische Informatik

Dipl.-Ing. Dr.techn. Christian Wimmer
Secure Systems and Software Laboratory, UC Irvine

Graz, April 2015

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Datum

Unterschrift

Abstract

This thesis deals with the problem of register allocation in a just-in-time (JIT) compiler. In JIT compilers, compilation time is part of the runtime. Therefore, it is important to balance the minimization of runtime due to code optimization and compilation time. JIT compilers are widely included in virtual machines. The compiler used in this thesis is the Java HotSpot™ Server Compiler. The virtual machine detects the so-called hot spots of a program, which are methods that are called very often. These methods will be compiled, while the rest of the code is interpreted. Register allocation is very often the bottleneck of runtime minimization. Registers can be accessed much faster than the memory. However, there is only a limited number of registers available. Values that cannot be stored at registers because of capacity restrictions, have to be moved to the memory. Movements between the memory and the processor have to be minimized, because memory bandwidth is scarce and therefore time-costly.

The intermediate code representation in the compiler is in static single assignment (SSA) form. This representation is used for code optimization and can be exploited also for register allocation. The structure of the intermediate representation of the server compiler has been analyzed and a simple feasible register allocator has been implemented.

The contribution of this master's thesis is as follows. First, a novel mixed integer program (MIP) is proposed to model the task of optimal register allocation on a SSA-form based intermediate representation for the IA-32 architecture. Furthermore, a model is presented, that takes only spilling decisions into account and can be combined with existing approaches for spill-free register allocation of programs in SSA-form. Third, different abstractions for register allocation are compared and analyzed.

Kurzfassung

Diese Arbeit beschäftigt sich mit dem Problem der Register Allokation in einem Just-In-Time(JIT)-Compiler. In JIT-Compilern ist auch die Kompilierzeit Teil der gesamten Laufzeit. Daher ist es wichtig, sowohl die Laufzeit des Programms durch Optimierungen im Compiler zu minimieren, als auch die Zeit die für die Kompilierung benötigt wird. Diese Ziele sind konfliktär. JIT-Compiler sind meist Bestandteil von virtuellen Maschinen. Der Compiler, der in dieser Arbeit verwendet wird ist der so genannte Java HotSpot™ Server Compiler. Die virtuelle Maschine erkennt die so genannten Hot Spots eines Programms, das sind die Methoden, die sehr häufig aufgerufen werden. Diese Methoden werden kompiliert, während der Rest des Codes interpretiert wird.

Register Allokation ist sehr oft der Engpass der Laufzeitminimierung. Auf Register kann viel schneller zugegriffen werden als auf den Hauptspeicher. Allerdings steht auch nur eine begrenzte Anzahl von Registern zur Verfügung. Die Werte, die nicht in den Registern gespeichert werden können, weil nicht genügend Register zur Verfügung stehen, müssen in den Hauptspeicher verschoben werden. Die Anzahl der Verschiebungen zwischen dem Speicher und dem Prozessor soll minimiert werden, da die Speicherbandbreite gering und Verschiebungen dadurch zeitintensiv sind. Der Zwischencode des Compilers ist in Static-Single-Assignment (SSA)-Darstellung. Diese wird genutzt um Optimierungen im Compiler durchzuführen. Die SSA-Darstellung kann auch für die Register Allokation verwendet werden. Die Struktur des Zwischencodes des Servercompilers wurde analysiert und ein einfacher zulässiger Registerallokator wurde implementiert.

In dieser Arbeit wird ein neues gemischt ganzzahliges lineares (MIP) Modell präsentiert, das die optimale Registerallokation in SSA-Form für die IA-32 Architektur löst. Außerdem wird ein Modell präsentiert, das nur die Spilling-Entscheidung berücksichtigt. Dieses Modell kann mit existierenden Lösungsmeth-

oden für spillingfreie Registerallokation von Programmen in SSA-Form kombiniert werden. Schließlich werden verschiedene Abstraktionsmodelle für Registerallokation verglichen und analysiert.

Acknowledgments

This thesis was conducted in collaboration with the Secure Systems and Software Laboratory at the University of California, Irvine and the Institue for Technical Informatics at Graz University of Technology. I am very grateful to the Austrian Marshall Plan Foundation for the financial support.

I want to thank my advisor Prof. Steger. Furthermore, I would like to thank Prof. Franz for giving me the opportunity to work on this thesis in his lab at UC Irvine. I am also very thankful to Dr. Christian Wimmer for his expert guidance. Furthermore, I want to thank my colleagues from UC Irvine for interesting conversations of all kinds. Finally, I thank Vera, my family and my friends for their support and patience during my studies.

Contents

1. Introduction

This thesis deals with register allocation for the Java HotSpot™ Server Compiler. The Java HotSpot™ Server Compiler is a just-in-time (JIT) compiler embedded in the Java HotSpot™ Virtual Machine. While traditional compilers translate code from a high-level language to machine code, before the program is executed, a JIT compiler translates code from a high level language to machine code during the execution of a program. Running a program written in a high-level language on a target machine requires either translation to machine code or an (software) interpreter. Both options have well known advantages and drawbacks. Executing machine code is usually faster than interpreting, but requires prior compilation and therefore reduces portability. JIT compilation is an approach to compromise both options.

It is possible that the JIT compiler compiles only part of the executed code or everything that needs to be executed. In the first option the remaining executed code needs to be interpreted. In the second option machine code is generated by the JIT compiler, when a routine (method) is called and no machine code exists. The advantage is, that only the routines that are actually used, need to be compiled. However in such a setting, lots of code needs to be compiled and compilation time increases the total execution time of the program significantly. Furthermore, sophisticated compilation optimizations lead to faster code, but need more time in the compilation process. Due to the fact that only a small portion of code is performance critical (Pareto Principle), only the performance critical code is worth being optimized.

Therefore, many JIT compilers only compile the performance critical code, in order to significantly improve the performance of the overall program. The idea is that the performance critical part of a program is executed more often. Therefore, it pays off to spend time in generating optimized code, while optimizing rarely

executed code will probably increase the overall runtime.

JIT compilers, which compile only promising parts of the code, are widely used for Virtual machines (VM) and scripting languages such as Java and JavaScript engines in various Browsers. VMs without integrated JIT compilers usually interpret high-level code, which usually leads to a poorer performance in terms of execution speed compared to code compiled by static (optimizing) compilers. The integration of a JIT compiler in a virtual machine is used to overcome this drawback. First the code is interpreted and the performance critical parts are detected during execution. These parts are compiled by the JIT compiler and the generated machine code is used when available. Summig up, JIT compilers in a VM are optimizing compilers, because the goal is to produce faster and more efficient code than the code produced by the default interpreter of the VM. Contrary to many other optimizing compilers, compilation time is a major issue for JIT compilers, because compilation time is obviously part of the total runtime of the program. There is a trade off between code optimizations and compilation speed in JIT-compilers. The optimization steps are time consuming and the compilation takes a substantial amount of time. More time spent in code optimization reduces the execution time of the code, but leads to an increase in compilation time.

Register allocation is one of the most important tasks in an optimizing compiler. Register allocation is placed in the back end of a compiler. The input to the allocator is usually an intermediate representation (IR) of the program, that uses an unlimited number of (virtual) registers. The task of the allocator is to replace the virtual registers by real registers of a certain architecture. The number of real registers is limited and register constraints need to be considered. For certain instructions only certain register can be feasible. The allocator might need to add additional instructions to the IR, which move variables to and from memory in order to find a feasible allocation. Allocating registers is usually one of the last steps of a compiler. Hence, the translation from IR, which is returned from the allocator, to machine code is usually a straightforward task. Registers are the part of the memory that can be accessed extremely fast. However, there is only a limited number of registers available. It is important to keep the number of movements between the memory and the registers as low as possible.

Pereira (2008) state that the code produced by an optimal allocator is over 250% faster than the code produced by a naive algorithm. This shows the high potential for optimization in register allocation. Register allocation is NP-complete, which was shown by Farach and Liberatore (1998) and Sethi (1975). Therefore, optimal or good allocation of the registers is mostly time consuming.

This thesis deals with register allocation for the Java HotSpot™ Server Compiler, which is a JIT compiler for the Java HotSpot™ Virtual Machine. The Java HotSpot™ Virtual Machine relies on the insight that the execution time of a program can be significantly reduced, when methods that are called often are compiled to machine code during runtime, instead of being only interpreted in the VM. More precisely, everything runs in interpreter mode first. Methods, which have been called more than a certain number of times are scheduled for compilation. These are the so-called hot spots where it pays off to use compilation. To sum it up, the idea is to speed up the process by focusing on the bottleneck methods that are called very often and therefore account substantially for the runtime. As mentioned above, the compilation is done during the execution of the program and therefore compilation time has to be added to the total execution time. Hence, low compilation time is crucial.

The Java HotSpot™ Virtual Machine has two compilers: a client and a server compiler. As the name implies, the client compiler is used for client machines, such as laptops and desktop computers, while the server compiler is used for server machines. However, these are only the default settings and the user can switch between the compilers depending on the application that she or he wants to run. The focus of the client compiler is a low startup and response time, whereas the goal of the server compiler is to get a good peak performance and startup time is less important. Therefore, compilation time is less important for the server compiler and more time can be spent in optimizing code during the compilation phase which will result in a faster execution time of the compiled method. As mentioned above, register allocation is a task in the compilation process, where a lot of time consuming code optimization can be done.

In the server compiler, a graph coloring based allocator is used while in the client compiler a much faster linear scan algorithm is used. The graph coloring based allocator in the server compiler is a so-called Chaitin - Briggs allocator

(Paleczny et al. (2001), Chaitin (1982), Briggs et al. (1994)) The register allocation for the client compiler has been subsequently improved by Mössenböck (2000), Mössenböck and Pfeiffer (2002) and Wimmer (2004), who started with a graph coloring based algorithm and then implemented and improved the linear scan algorithm.

In this thesis, an analysis of the intermediate representation of the server compiler is provided and a simple feasible register allocator has been implemented. Moreover, two novel integer linear programs (ILP) are proposed. The first is solving the problem of optimal register allocation on a SSA-form based intermediate representation for the IA-32 architecture. The second relies on a decomposition and tackles the aspect of spilling variables to memory. Finally, different abstractions for register allocation are described, compared and analyzed.

The remainder of the thesis is organized as follows. Chapter 2 deals with register allocation and related aspects. It describes the memory hierarchy and the registers that exist in the most common architecture, the IA-32. Moreover, there is a basic introduction on the Java virtual machine and the server compiler. The basics in register allocation are discussed. The chapter deals with programs, the static single assignment (SSA)-form, the translation out of SSA-form and finally with the intermediate representation. Chapter 3 describes different abstractions of register allocation, which show how the register allocation problem can be modeled. More precisely, there are solution methods based on graph coloring and on modeling the problem as multi-commodity network flow problem. Furthermore, there is the linear scan algorithm with second chance binpacking, the boolean quadratic programming approach and register allocation by puzzle solving. Finally, there are also ILP based approaches. Chapter 4 gives a detailed overview of the task of register allocation in the Java™ HotSpot server compiler for the IA-32 architecture and Chapter 5 presents mathematical programs for the register allocation problem. Finally, Chapter 6 concludes the thesis.

2. The Role of Register Allocation in the Compiler and Computer Architecture

This chapter describes the role of register allocation in compiler optimization and its interfaces to the computer architecture. First, the memory hierarchy is described, showing the crucial role of register allocation. Then the register set in the IA-32, the most common architecture at the moment, is presented. Then the Java HotSpot™ Virtual machine and the server compiler are described. Finally, the task of register allocation and programs in SSA-form are discussed.

2.1. Memory Hierarchy

In typical computer architectures there is a memory hierarchy which is organized in a pyramid. Figure 2.1 shows this memory hierarchy. The axes are the size of the available storage space and the increasing cost and decreasing access speed. On the top are the registers. There is only a limited amount available. The cost in terms of production cost and power consumption is quite high. According to Pereira (2008) reading and writing to registers can be done in one cycle of the CPU clock. The registers are followed by L1 and L2 cache, the main memory and finally the hard disk.

The register allocator can access directly the registers and the main memory. However, the L1 and L2 cache are used to speed up loads and stores from the main memory. They cannot be accessed directly, but they are used by the memory management in the background. Data that is loaded frequently from the main memory, will probably be cached to speed up the loading, but those operations

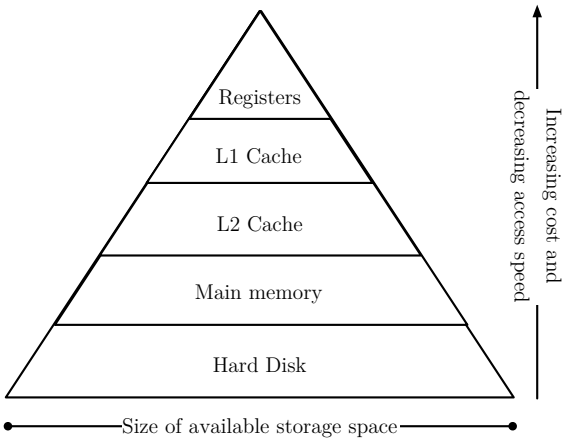


Figure 2.1.: Memory hierarchy, taken from Pereira (2008)

are hidden in the memory management and the register allocator has no control about it. Therefore, it is hard to estimate the load and store time to main memory. According to Pereira (2008) an optimal allocator is over 250% faster than the code produced by a naive algorithm.

Hack (2004) highlight that in modern processor architecture the time for accessing a register is at least one order of magnitude faster than accessing a memory location. This shows the enormous potential for optimization in register allocation.

2.2. Register Set for the IA-32 architecture

The Java HotSpot™ Virtual Machine, and therefore the server compiler, can be used for the IA-32 architecture and for the Sparc architecture with several operating systems. Therefore, most algorithms in the compiler are platform independent. The difference between the two architectures is that the IA-32 is a so-called complex instruction set computer (CISC) architecture with an irregular instruction set and a low number of registers, while the Sparc is a reduced instruction set computer (RISC) architecture with a regular instruction set and a higher number of registers. In a CISC architecture, complex instructions are supported

directly in hardware. These are for example, complex addressing modes and procedure calls. Therefore, a lower number of instructions are necessary and the code will be more dense. Since there are fewer instructions, fewer operands are needed and therefore fewer registers are required. However, the complex instructions will take more time and probably lead to an irregular architecture. The RISC architecture on the other hand only supports simple instructions which can be executed in a very fast way. More complex instructions have to be replaced by a sequence of simple instructions. Therefore, there are more registers available in this architecture. In RISC architectures, most instructions have a similar length and structure and operands have to be located in a register. Transfers to memory can only be done with load and store instructions. These properties enable better and easier register allocation on a RISC architecture. Moreover, efficient pipelining is facilitated. These properties make RISC architecture, that has a regular and simpler architecture, competitive to the CISC architecture.

In this thesis the IA-32 architecture is used, because it is the most common architecture at the moment. Since we are dealing with an irregular architecture, not all registers can be used freely.

The first eight registers are called general purpose registers. However, they cannot be used for every purpose. They can generally be used for arithmetic integer and logical operations, for address calculations and for memory pointers. The names of the registers are *eax*, *ebx*, *ecx*, *edx*, *esi*, *ebp* and *esp*. However, the *esp* register is always used for holding the stack pointer. It is not allowed to use *esp* for any other purpose, because all instructions that support the stack management use this register for looking up the current stack pointer. Therefore, only seven registers can be used freely, which are shown in Figure 2.2. In the client compiler the register *ebp* is used for the base pointer, but this is not the case for the server compiler.

Figure 2.2 shows the effect of aliasing. The 32-bit architecture supports programs written for the older 16-bit and 8-bit architectures. They do not have special physical 16-bit and 8-bit registers, but use the 32-bit registers. For example, the 16-bit register *ax* uses the first 16 bits of the *eax*. This has to be taken into account in the register allocation process. For instance, when *ax* is used, *eax* cannot be allocated at the same time. The same holds for the 8-bit architecture

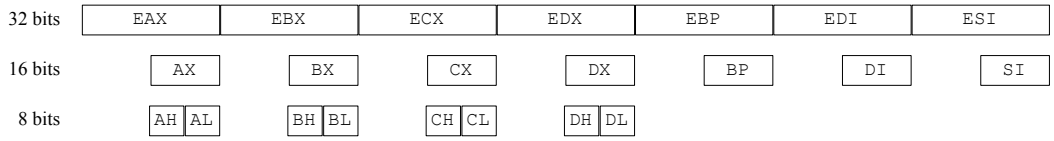


Figure 2.2.: General purpose registers from the x86 architecture showing aliasing, taken from Pereira and Palsberg (2008)

registers.

The seven general purpose registers can be used freely for most of the operations, but some operations still require special registers. For example, there are some instructions that need fixed, predefined registers. This property is often referred to as precoloring. The term was coined in the context of the graph coloring problem. For example the return value of a procedure needs to be in *eax*.

Additionally to the eight general purpose registers there are eight floating point, or FPU, registers. These registers are organized as a stack because historically they were located in the floating point co-processor (x87), which was stack based. Modern CPUs use the SSE extensions for floating point computations. The SSE instructions operate on XMM registers. Finally, there are eight multi media extension (MMX) registers and one flags register. The MMX registers alias the floating point registers.

2.3. The Java Virtual Machine

One advantage of the programming language Java is that "Java executables" can be used for different systems, as opposed to specialized languages. More precisely, Java source code is compiled to the so-called Java bytecode, which is the binary representation of the source code and still a high level language. The code is the native code for the Java Virtual Machine (JVM). As the name suggests, the JVM is not a real machine, but a piece of software that runs on different physical machines. The JVM makes Java a machine-independent language. Moreover, the bytecode of Java only needs a virtual machine (VM) to run and therefore the same bytecode can run on different platforms. For other languages, such as C++

for example, different executables are needed for different operating systems and architectures. In the early beginnings, the bytecode was interpreted by the JVM. This made Java slower compared to languages that were compiled to machine code. JIT compilation overcomes this drawback and has therefore been included in many JVMs.

One implementation of a JVM is the Java HotSpot™ Virtual Machine, which has been developed by Sun Microsystems. In the following, the Java HotSpot™ Virtual Machine is described based on the information in Sun Microsystems (2001).

There is a general rule that most programs spend the majority of their time in certain methods. The Java HotSpot™ Virtual Machine uses this property. Methods that are executed very often, the so-called "hot spots", are compiled, while the remaining code is interpreted. Since the remaining code (the infrequently performed methods) is not compiled, there is more time available to optimize the runtime of the hot spots in compilation. Therefore, a focus on the hot spots is guaranteed, while the compilation time can be kept lower. By using both, interpreted and compiled code, the overall performance can be optimized.

The hot spots are detected by using runtime information. During the interpretation this information is collected. There are two counters: the method-entry and the loop back-branch counter. The first counter counts each start of the method, while the latter one is incremented when a backward branch is executed. If any of these counters reaches a given threshold, the method is scheduled for compilation.

The Java HotSpot VM is included in the Java Platform SE 6, and is available on the Solaris Operating Environment (SPARC Platform Edition and Intel Architecture Edition), and the Linux and Microsoft Windows operating systems for the Intel Architecture platform. It supports 32-bit and 64-bit architectures.

Figure 2.3 shows the Java HotSpot™ Client and the Java HotSpot™ Server VM. They use a different compiler, but they both have an interface to the same VM and therefore use the same garbage collector (GC), interpreter and so on.

The difference between the client and the server compiler is that the server VM is tuned to maximize peak operating speed. Therefore, the fastest possible operating speed is more desirable than a fast startup time or smaller runtime

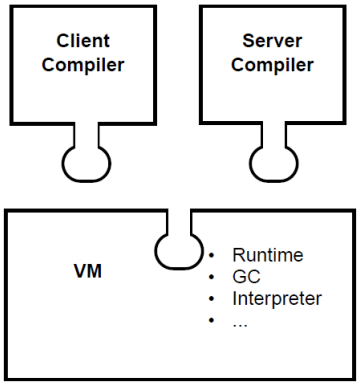


Figure 2.3.: The Java HotSpot™ Client and the Java HotSpot™ Server VM, taken from Sun Microsystems (2001)

memory footprint.

The server compiler is a high-end fully-optimizing compiler. The SSA-based representation is used for optimizations.

In the compilation process, the bytecode is first transformed to the graph-based intermediate representation (IR). Then several optimization steps including register allocation are applied. The machine code together with additional meta data is created from the IR after register allocation.

2.4. The Server Compiler

The compilation is performed in several phases: First, the parser is executed. Then there is the machine-independent optimization, followed by the instruction selection. The next step is global code motion and scheduling. After that, the register allocation phase takes place, which is subject to study in this thesis. Finally, peephole optimizations and code generation are performed. For more details see Paleczny et al. (2001).

For the intermediate representation (IR), the static single assignment (SSA) form is used.

The SSA-form based IR of the server compiler consists of the so-called sea of nodes. Initially, all instructions are represented by nodes, which are linked

together by edges that state dependencies to each other. On the sea of nodes optimizations are performed. During the compilation, the nodes are scheduled to basic blocks and build a control flow graph. This graph is given as an input to the register allocator. Each node that returns a value has a register mask, which defines the set of feasible locations. These locations can either be registers or stack slots. For input values of a node, input register masks are given, that are used in the same way as output register masks.

2.5. Fundamentals of Register Allocation

This chapter describes the basics in register allocation. First, it is necessary to define a program. Then the SSA-form will be described in detail and it will be shown how the translation out of SSA-form works.

2.5.1. Programs

The focus in this thesis is on global register allocation, which is based on method or procedure-level. The input for such a global register allocator is basically a single procedure of a program. In the following the notion of a program will be described in more detail, based on the description in Hack (2004). The program is represented by its control flow graph, which consists of labels and instructions. Register allocation is the task of assigning storage locations, i.e., physical registers and stack positions. Additionally, move instructions can be inserted to the program to change the allocation of the variables. The type of instructions in a program and what the program is computing is not relevant to register allocation. For register allocation only the input and output of an instruction are important, because this defines how many registers are needed, when they are needed and for how long, and also which kind of register is required.

A program including all the necessary information for register allocation can be described by the tuple $(V, O, L, pred, arg, res, op, start)$, where V represents the variables and O the set of operations. The only operation that will be inserted additionally in the program during the register allocation phase is the copy operation. Hence, this operation is required to be in O. Let L be the

set of labels. Each label represents an instruction of the program and in each instruction an operation is carried out. The function *op* maps a label to the operation executed at that label. For example, ADD is an operation and $a := a+4$ is an instruction that corresponds to that operation. Let *pred* be a function that assigns each label a set of predecessor labels. The labels have to be ordered linearly. Let $pred(l, i)$ be the l-th predecessor of i. Note that the predecessor function also defines a complementary map wich defines all successors. Let arg be the set of input operands and res the set of output operands. Finally, *start* represents the start of the control flow. Therefore, *start* is the entry point of the program.

An example is given by the following program, which is shown in Figure 2.4. The procedure $fac(i)$ computes the factorial of i. It uses i as an input and r as an output. The control flow graph of the program is given in Figure 2.5. If we look for example at label three (l3), $res(l_3) = r$, the output operand is r and $arg(l_3) = (r, i)$ where r and i are the input operands. Label l_3 has only one predecessor ($|pred(l_3)| = 1$) and $pred(l_3, 1) = l_2$.

```
1 procedure fac(i){
2   r:=1;
3   while (i>0){
4     r:=r*i;
5     i:=i-1;
6   }
7   return r;
8 }
```

Figure 2.4.: Example program E_1: procedure fac(i)

A path in a program is defined as a linearly ordered set of labels $(l_i, ..., l_n)$, where l_i is a predecessor of l_{i+1} which is a predecessor of l_{i+2} and so on, $l_i \rightarrow l_{i+1}....l_{n-1} \rightarrow l_n$.

Basic Blocks. A path that consists of labels that have only one predecessor and one successor is a basic block. More precisely, a basic block B is a sequence of labels where each label has only one predecessor and one successor. Hence, B does not include branches and the instructions in B have to be executed sequentially. The sequence of instruction in a basic block can be referred to

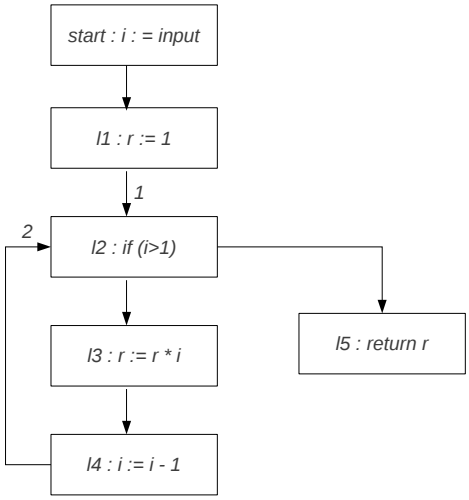


Figure 2.5.: Control flow graph of example E_1

straight line code.

Moreover, a basic block needs to be maximal. This means that it cannot be extended and all labels that can be combined to a basic block B are included in B.

The control flow graph of program fac(i) in Figure 2.5 can be combined in basic blocks. This is shown in Figure 2.6. Here it can be seen that *start* and l_1 can be combined to a basic block and also l_3 and l_4 can be combined to a basic block. The control flow within a basic block is simple and the control flow edges only exist between basic blocks.

Dominance Relation. The dominance relation is a relation between two labels, that is helpful to define other important properties of a program. A label l_1 dominates l_2 ($l_1 \preceq l_2$) if and only if each path from *start* to l_2 contains l_1. Dominance holds the properties of an order relation and is therefore reflexive, transitive and anti-symmetric.

A program is strict, if each label, where a variable v is used, is dominated by a label where v was defined. This means that in strict programs each variable has to be defined before it is used. This thesis only deals with strict programs. Not strict programs can be converted to strict programs by inserting instructions

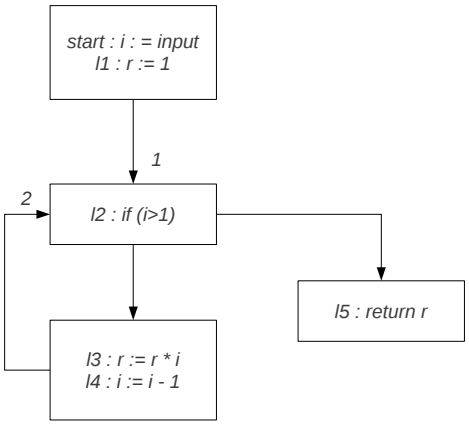


Figure 2.6.: Control flow graph of example E_1 in basic blocks

where a variable gets defined a dummy value($undef$) at locations where the above mentioned property was violated.

2.5.2. Static Single Assignment Form

A program that fulfills the so-called static single assignment (SSA) property is in SSA-form. The SSA-property says that each variable has to be statically defined once. A program in SSA-form is not necessarily strict. As mentioned above, this thesis will only deal with strict programs and therefore all properties described are only guaranteed for strict programs. For each usage of a variable there is exactly one label where the variable is defined. Let D_x be the label where x is defined. Let us look at the piece of code in Figure 2.7.

```
1  i = 2;
2  a = 3;
3  i = func(i,a);
4  a = a + 1;
```

Figure 2.7.: Straight line code not in SSA-form

Each line corresponds to a label. The program is not in SSA-form because a is defined in line (label) 2 and line (label) 4. Moreover, i is also defined twice,

in line 1 and line 3. It can be simply converted to a program in SSA form by giving each definition a unique name. Therefore, $i2$ and $a2$ are introduced. The code in SSA form then looks like the code in Figure 2.8.

```
1  i1 = 2;
2  a1 = 3;
3  i2 = func(i1,a1);
4  a2 = a1 + 1;
```

Figure 2.8.: Straight line code in SSA-form

In this example only straight-line code was used. In the following example in Figure 2.9 there are two variables and one *while*-loop. Pieces of code like this are very common, where the value of the variable depends on the path that is taken.

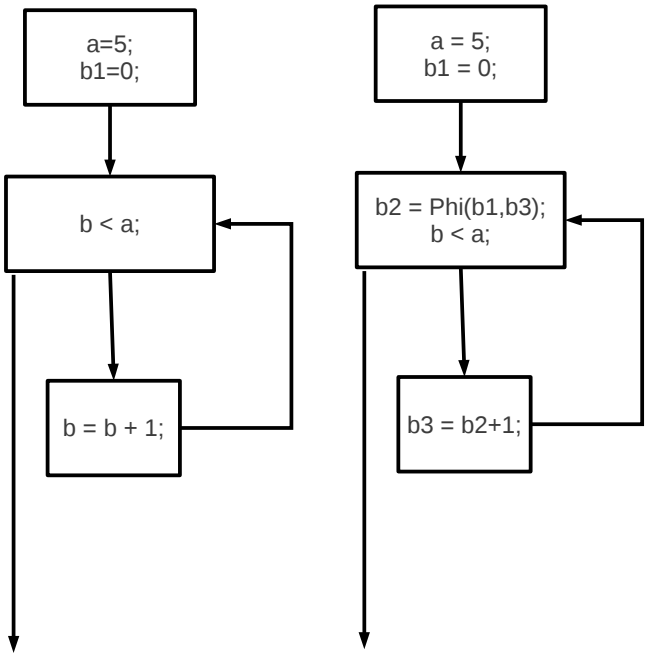
```
1  a=5;
2  b=0;
3  while(b<a){
4     b=b+1;
5  }
```

Figure 2.9.: While-loop not in SSA-form

Figure 2.10 shows a try to transfer the example to SSA-form. However, it is not possible to simply replace each variable by a new one at every definition, because it is not clear which variable should be used for b_x in line 3 and 4. It will depend on the control flow graph. The control flow graph can be constructed for the example, which is shown in Figure 2.11(a).

```
1  a1=5;
2  b1=0;
3  while(bx<a1){
4     b2=bx+1;
5  }
```

Figure 2.10.: Transferring a while-loop to SSA-form



(a) Without SSA-form (b) In SSA-form

Figure 2.11.: Comparison of code not in SSA-form and code in SSA-form

The unknown variables, b_x, occur where the control flow merges and the value of a variable depends on the control flow. In this case artificial instructions are needed to merge the values. For this purpose so-called Φ-functions are introduced. In our example the definition of b_x depends if the control flow comes from the loop or from the initial input. b_x is replaced by b_3. b_3 can be defined by the following instruction: $b_2 := \Phi(b_1, b_3)$. The instruction needs to be inserted where the control flow is merging. If the flow originates from predecessor 1, b_1 is assigned, else b_3 is assigned to b_2.

The Φ operation has as many input operands as block predecessors. Φ operations are used at the beginning of a basic block and can be seen as parallel copies as they are not ordered. More precisely, all Φ functions can be placed together, because they are ordered arbitrarily and dependencies between each other can be resolved. They can be pooled together and can be written in matrix notation in order to emphasize that they belong to no natural sequence. Hence, all Φ-functions can be represented using one label.

Figure 2.12 shows the control flow graph of example E_1 in SSA-form. It was necessary to introduce Φ-functions. They were pooled together in the matrix form. If the flow originates from edge 1, $i3 := i1$ and $r3 := r1$. Else, if the flow originates from edge 2, $i3 := 12$ and $r3 := r2$. It is also possible to use line-wise notation and split the matrix per row.

2.5.3. Translating out of SSA-form

The SSA-form facilitates powerful program optimizations in a compiler (Sreedhar et al. (1999), Cytron et al. (1991)). However, Φ-instructions are non-native instructions and therefore they must be eliminated before final code generation. There is a simple algorithm by Cytron et al. (1991) for this translation. For each input operand of the Φ-function a copy is inserted. Figure 2.13 shows the translation out of SSA for example E_1, where the two Φ-functions are replaced by four copies ($l6$- $l9$), one for each operand. The Φ -function $i_3 := \Phi(i_1, i_2)$ is replaced by $i_3 := i_1$ if edge 1 is taken and $i_3 := i_2$ if edge 2 is taken. The removal of the Φ-function for r_3 works analogously.

The described algorithm does not work in all cases. It works well for opti-

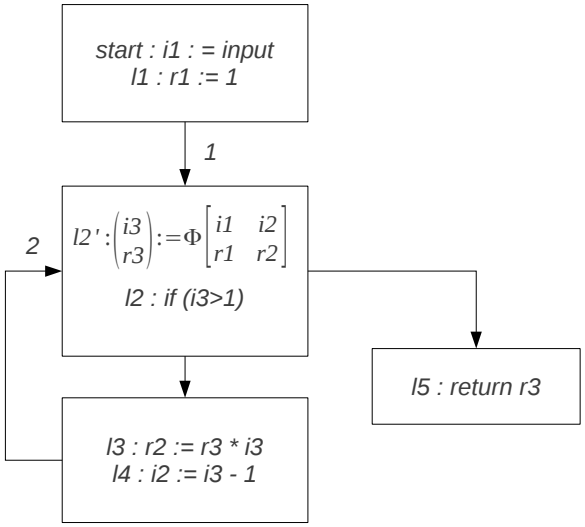


Figure 2.12.: Control flow graph of Example E_1 in SSA-form

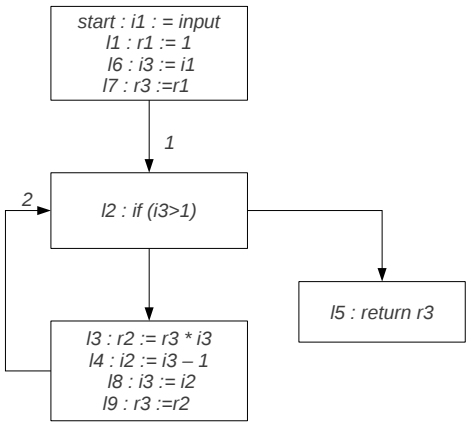


Figure 2.13.: Example E_1 after translating out of SSA

mizations that do not radically change the namespace like constant propagation and dead code elimination, but for optimizations that do radically change the namespace, such as copy folding and aggressive value numbering, it does not work (Briggs et al. (1998)). Two examples where the algorithm fails are the lost copy problem and the swap problem. This problems will be described in the following.

Lost Copy Problem. The lost copy problem occurs when copy folding is done in a program in SSA-form. If the copies are folded in a basic block with critical backward edges, the simple algorithm for translating out of SSA-form will lead to incorrect code. Critical edges go from blocks with multiple predecessors to blocks with multiple successors. Therefore, a new basic block has to be inserted. An example is given in Figure 2.14.

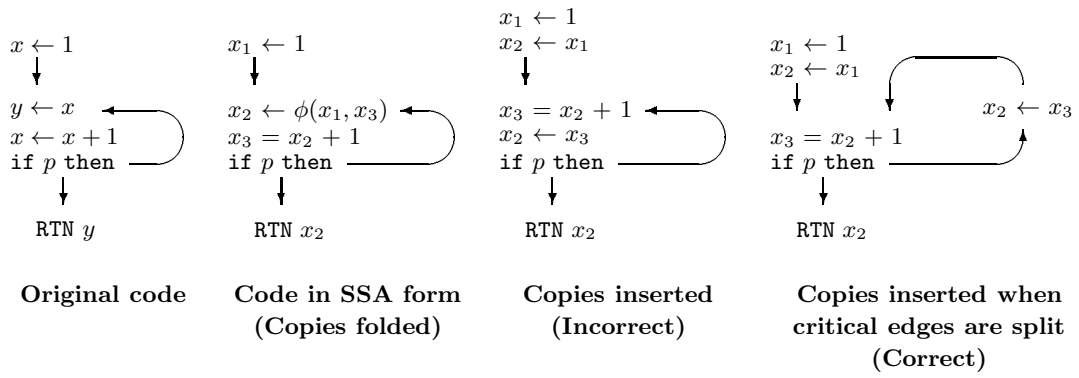


Figure 2.14.: Example of the lost copy problem taken, from Briggs et al. (1998)

The first subfigure shows the original program that has a critical edge, which is the backward edge of the loop. The program is translated to SSA-form in the second subfigure. Copies are folded in SSA-form and therefore the variable y is not needed any longer and is discarded. In the third subfigure the code after applying the naive algorithm to remove Φ-functions is shown. This leads to incorrect code, because the move from x_3 to x_2 has one incorrect execution before the return statement. The last subfigure shows the solution to the lost copy problem, where an additional basic block is inserted.

Swap Problem. Another problem which occurs because of copy folding is the swap problem. If the input operand of a Φ-function is the output operand to another Φ-function in the same block, the naive algorithm to remove Φ-nodes may lead to incorrect code. This situation occurs after code that swaps the value of two variables. After copy folding in SSA-form, the helper variable for the swap will disappear and translating out of SSA with the naive algorithm will fail.

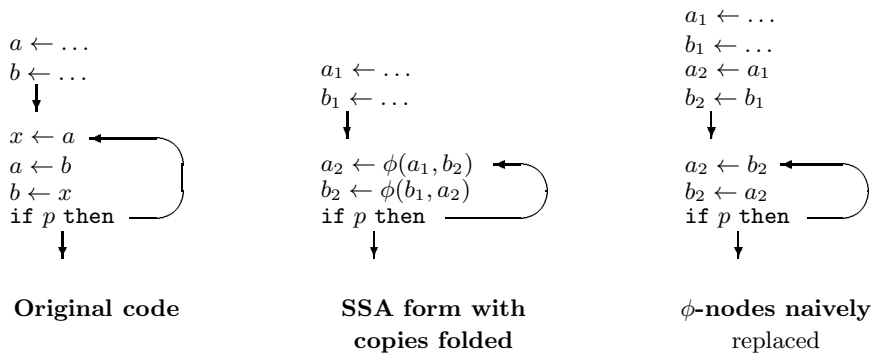


Figure 2.15.: Example of the swap problem, taken from Briggs et al. (1998)

Figure 2.15 shows an example for the swap problem. The first subfigure shows the original code where a swap is performed between a and b and x is the temporary help variable. In the second subfigure the program is translated to SSA-form and copy folding was performed. Therefore, x was removed. The last subfigure shows the incorrect code due to the removal of x. To overcome this drawback, the copies need to be ordered carefully.

There are three possibilities concerning the time when the translation out of SSA-form can be done.

Perform the elimination before the register allocation phase. The first approach is the traditional approach and it was done for example in Wimmer (2004). Standard algorithms to eliminate Φ-functions are used. The register allocation is done in non-SSA-form. The approach is very similar to an allocator for a compiler, whose intermediate representation (IR) has never been in SSA-form.

Perform the elimination in the last step of register allocation. For the second approach, an example is Wimmer and Franz (2010), where the authors improved a previous linear scan algorithm (Wimmer (2004)), that did the elimination before the register allocation phase. The algorithm operates on SSA-form and they show that the simpler algorithm is faster and generates equally good or slightly better machine code than the previous version. The conversion out of the SSA-form is included in the resolution phase of the linear scan algorithm. The authors state the following advantages of the SSA-form for their algorithm:

Lifetime analysis: In IRs that are not in SSA-form an iterative data flow analysis has to be performed. The basic data flow analysis has to be repeated until a stable state is reached. In SSA-form each variable is only defined once at a single point of definition. This property can be used to determine live ranges in a single pass over the CFG.

Lifetime holes: Holes of live ranges always end at the end of a basic block. This property can be used to guarantee that variables defined during a lifetime hole of another variable cannot have overlapping life ranges. This eliminates expensive overlapping checks.

No artificial order: Moves that were inserted in the deconstruction phase of SSA-form do not have an artificial order. In the register allocator there is no order that is given by the Φ-functions.

Figure 2.16 shows the steps of the linear scan algorithm with (right subfigure) and without SSA-form (left subfigure). The main differences are that in the SSA-form no data flow analysis has to be done. Furthermore, SSA deconstruction is necessary and is included in the resolution phase.

The drawback of their algorithm is that costly memory transfers occur in the resolution phase where they translate out of SSA. Hack and Goos (2006) proposed an graph coloring algorithm based on the second approach. They show that the interference graph of programs in SSA-form are cordal and can therefore be colored in polynomial time. They improve the Chaitin-Briggs allocator.

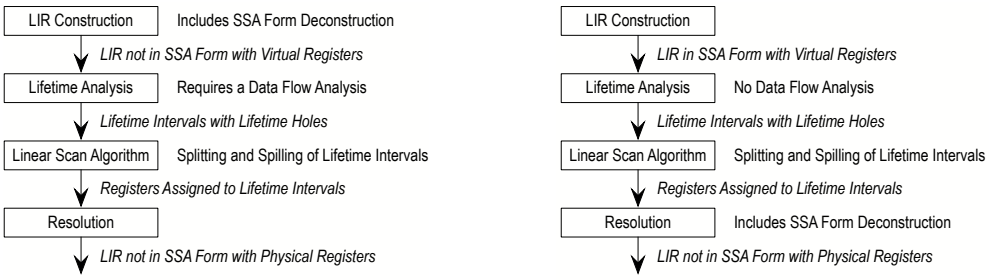


Figure 2.16.: Steps of the linear scan algorithm with and without SSA-form, taken from Wimmer and Franz (2010)

Perform the elimination after the register allocation. The third approach was used by Pereira and Palsberg (2009). They use the conventional SSA (CSSA)-form, which is a more restricted version of the SSA-form. The CSSA form was introduced by Sreedhar et al. (1999). The CSSA form ensures that variables in the same Φ-function do not interfere. More precisely, the live ranges of the operands and the assigned value in a Φ-function do not overlap. The costly memory transfers that occur in the second approach can be avoided in this approach. Moreover if the program is in CSSA-form, the lost copy problem and the swap problem do not occur while translating out of the SSA-form.

The CSSA form is a special case of the SSA form. The output of the standard algorithm for translating a program to SSA-form is in CSSA-form (Sreedhar et al. (1999)). However, in the optimization phases this property can get lost, for example during copy folding. The following example illustrates how the CSSA-property can get lost. First, there is a small example that uses three variables, a, b and c (see Figure 2.17).

This piece of code is transformed to CSSA form as shown in Figure 2.18. The definitions are replaced by unique names and a Φ-function is inserted where the control flow is merging.

As shown in Figure 2.19 the CSSA form can get lost due to optimization steps. In this example $c1$ is eliminated. However, because of the *print* statement in the last line, $a1$ and $a3$ have overlapping live ranges.

```
1 a=func();
2 b=2;
3 c=a
4 if(a<b){
5    a=a+b;
6 }
7 print(a);
8 print(c);
```

Figure 2.17.: Example not in SSA-form

```
1 a1=func();
2 b1=2;
3 c1=a
4 if(a1<b1){
5    a2=a1+b1;
6 }
7 a3=phi(a1,a2);
8 print(a3);
9 print(c1);
```

Figure 2.18.: Example for the CSSA form

In the CSSA-form all inputs of a Φ-function can share the same register, because they do not interfere. This property is especially important for memory coalescing. It is possible to translate out of CSSA-form without memory transfers, which is not always possible if the properties of CSSA are not fulfilled.

2.5.4. Intermediate Representation

As the name suggests, the intermediate representation (IR) is the representation that is used in the transition from the written code to the machine language, that is performed by the compiler. The IR in SSA form is used by many compilers. The IR is the representation used between the bytecode and the machine code. On the IR the optimizations are performed.

```
1 a1=func();
2 b1=2;
3 if(a1<b1){
4    a2=a1+b1;
5 }
6 a3=phi(a1,a2);
7 print(a3);
8 print(a1);
```

Figure 2.19.: Example not in the CSSA form due to optimization steps

In the following, the IR of the Java HotSpot™ server compiler is described, based on the description in Click and Paleczny (1995). In their paper they provide a simple, fast and easy to use, graph-based IR. More precisely, the representation is a directed graph with labeled vertices and ordered inputs. The label on a node shows the kind of operation that the node represents. Edges do not have a label. The inputs to the node's operation are on the input edges. On the output edges is the value that is defined by the node based on its inputs and operation. For example, the operation $a := b + c$ is represented by three nodes, a, b and c. Nodes b and c are input nodes to a. The operation on a is ADD.

In the traditional approach there was a distinction between control flow and data flow. In the new representation there is a top level, the CFG which consists of basic blocks. Then there is the bottom level where each basic block contains instructions. The authors replace this traditional distinction with a new representation. Instead of basic blocks, they use so-called region nodes. A region node has input control values from the predecessor nodes and delivers a merged control as an output. The information that says in which basic block a node is in, is taken by a control input. The control edges are optional.

3. Modeling Approaches of Register Allocation

Register Allocation is a hard problem and an important task in code optimization. Different abstractions were used to simplify the problem and find good feasible register allocations. Before describing the different modeling approaches a basic distinction has to be made. We can distinguish between the following types:

Local Allocation: Local allocation assigns registers for each basic block, i.e., a block which consists of only straight-line code without any branches. Local allocators focus only on a part of the currently compiled method. Very crucial parts of the program like the innermost loops can be picked out for performing local allocation.

Global Allocation: Global allocation, on the other hand, deals with allocating a complete method or function. Therefore, it is necessary to deal with branches and control structures. Global allocation is usually slower than local allocation, but can achieve better results.

Farach and Liberatore (1998) and Sethi (1975) show that both global and local register allocation are NP-complete.

In the following, six different approaches for global allocation are described. The first one is graph coloring, then we describe the modeling as a multi-commodity network flow problem. Then the linear scan algorithm, partitioned boolean quadratic programming and register allocation by puzzle solving are described. Finally ILP based models are presented. Most of these approaches simplify the optimization problem in order to make it easier to find a good feasible

solution. Therefore, they do not explore the entire search space. The simplified problems are mostly well-known optimization problems, which are still NP-hard. Therefore, most solution methods are heuristics.

3.1. Graph Coloring

A well known and widely used approach to model register allocation is by modeling the problem as a graph coloring problem. Graph coloring is the task of assigning colors to vertices such that no two vertices that are connected by an edge have the same color. A k-coloring is a coloring that can use at most k colors. The graph coloring problem is NP-hard except for special cases such as chordal graphs.

So-called virtual registers are used to model the interference graph for which the graph coloring is performed. More precisely, each variable is assigned to a virtual register. In this representation the fact that only a limited number of registers is available is ignored. So the task of register allocation is to replace the virtual registers by the available physical registers on the target machine. For a more detailed description see Briggs (1992).

The following example shows how the interference graph is constructed. Figure 3.1 shows the definition of live ranges. The live range of a virtual register begins at the first definition and ends at the last use.

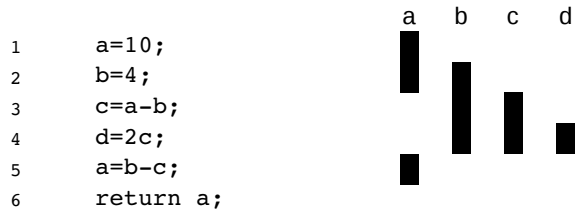


Figure 3.1.: Code example with live ranges

An interference graph is constructed with the information of the control flow graph. Each variable is represented as a node. If two variables are live at the same time at some part of the program, the two variables cannot share a single register, but must reside on two different registers or one of them has to be spilled

to memory. This interference is modeled as an edge between the interfering nodes. A certain color corresponds to a physical register. Nodes with the same color will be assigned to the same register. The drawback of this modeling is that some information of the problem gets lost.

The interference graph is defined on an undirected graph $G = (V, E)$, where V is the set of nodes, which are the virtual registers, and E is the set of edges, which correspond to the interferences. Two virtual registers interfere if their life ranges overlap. In that case, they cannot be assigned to the same physical register. Once the interference graph is built, the graph coloring problem can be solved, to get a solution to the register allocation. We are looking for a k-coloring of G, where k corresponds to the number of physical registers available.

It can happen that there is no feasible solution to this problem, then some virtual registers have to be spilled to memory.

The first graph coloring based allocator was implemented by Chaitin (1982) and was called the Yorktown allocator. Figure 3.2 illustrates the phases of this allocator. The algorithm is described as presented in Briggs (1992).

Renumber: In the first step the right number of names and the live ranges are determined.

Build: Based on the information of the previous step, the interference graph is constructed.

Coalesce: Copies that are not necessary are removed. A copy can be removed if the live ranges do not interfere. The removal can change the interference graph, therefore the interference graph has to be updated, i.e., the build step is performed again. The steps are repeated until no more copies can be removed.

Spill Costs: In this phase the spilling cost is approximated. The cost is computed by taking into account the number of instructions, where the variable is used, and the loop depths of these instructions.

Simplify: The phase *simplify* and the phase *select* are the phases where the coloring is performed. In this phase the nodes with a degree smaller than

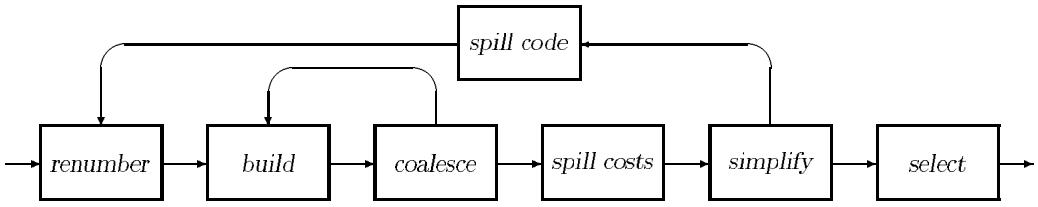


Figure 3.2.: Steps of the yorktown allocator as shown in Briggs (1992)

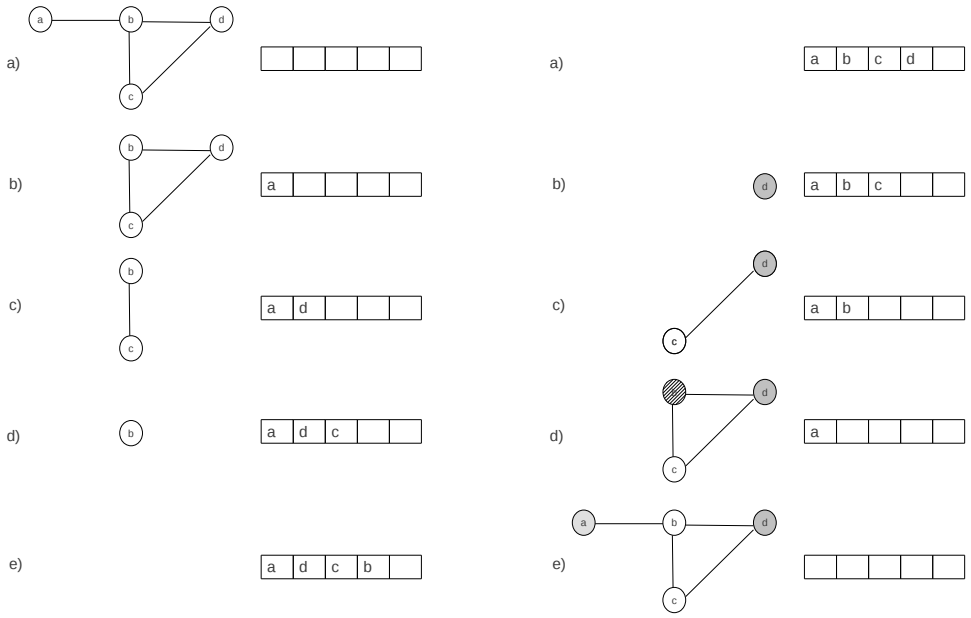
k are subsequently removed and pushed on the stack. This is illustrated in Figure 3.3(a) for the example with 4 nodes that was defined in Figure 3.1. We assume that three registers are available. Node a is removed first because it has the smallest degree. Then all nodes have the same degree. In that case a node can be chosen arbitrarily. Therefore, node d is removed, then node c and finally node b. It can happen that all nodes in G have a degree $\geq k$. In that case a node is chosen for spilling. Now it would be possible to go back to the *renumbering* phase and perform all the subsequent steps again. However, it is better to only mark the node for spilling and continue the *simplify* phase to find other nodes that have to be spilled. When G is empty and no node has been marked for spilling, the *select* phase can be executed, which will guarantee to find a feasible coloring. Otherwise it is necessary to go to the *spill code* phase and back again to the beginning.

Select: In this phase the nodes from the stack are removed one after another, reinserted in G and a feasible color is chosen for them. The select phase is illustrated in Figure 3.3(b). The last node, node d, is popped from the stack and inserted into the graph. A new color is assigned to the node. This procedure is repeated until all nodes are removed from the stack, inserted in the graph and have assigned a color.

Spill Code: For each node that was chosen to spill, the spill code is inserted. This means that for every instruction that uses a spilled variable, a load and store instruction has to be inserted.

A solution to the register allocation problem, constructed by the graph coloring

algorithm, is shown in Figure 3.4. Node a is put on the first register, node b on the third register and node c on the second register. Then the live range of node a is finished and the first register can be used again for node d, whose live range is finished too. Finally, node a is put on the first register again.



(a) Simplify phase

(b) Selection phase

Figure 3.3.: Two phases in the graph coloring algorithm

The graph coloring algorithm was adapted for programs in SSA-form by Hack et al. (2006). Because of the simpler structure of the interference graph due to the SSA-property, an algorithm without iterations could be developed. The phases are shown in Figure 3.5. The interference graph of a program in SSA-form is chordal. Chordal graphs have the property that their chromatic number is equal to the size of the largest clique in the graph. The chromatic number is the maximal k for which a k-coloring is possible. The largest clique in the interference graph corresponds to the liveset of label with the most variables live. Therefore, after the liveness analysis the spilling decisions can be made, so that it can be guaranteed that graph coloring will succeed in one iteration. Moreover,

```
1  a=10;              //r1=10
2  b=4;               //r3=4
3  c=a-b;             //r2=r1-r3
4  d=2c;              //r1=2r2
5  a=b-c;             //r1=r3-r2
6  return a;          //r1
```

Figure 3.4.: Solution to the register allocation problem, constructed by the graph coloring algorithm

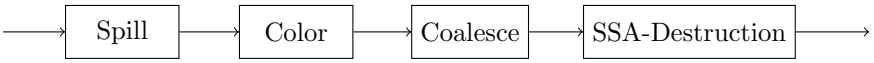


Figure 3.5.: Phases of SSA-form based graph coloring register allocation (taken from Hack et al. (2006))

it can be assured that such a k-coloring can be found in quadratic time. After all virtual registers are colored, coalescing can be performed. Note, that coalescing before this phase would destroy the SSA-form-structure that is exploited in this approach. Finally SSA-form deconstruction is done, which consists of inserting parallel copies. In the rare cases where circular shifts are necessary and no spare register is available, moves to the memory and back are necessary.

In general, graph coloring approaches are successful for regular architectures with a high number of registers. For these architectures, good results are obtained in reasonable runtime. Extensions to the basic algorithm exist for irregular architecture. However, they increase the runtime and do not always yield good results.

3.2. Multi Commodity Network Flow

The register allocation problem can also be modeled as a multi-commodity network flow (MCNF) problem. This method was first applied by Koes and Goldstein (2005) and Koes and Goldstein (2006) for the gcc compiler. They report code size improvements of on average 6.84% compared to a traditional graph allocator.

In the following, the MCNF problem is described, and then the modeling of the register allocation problem as a MCNF.

The MCNF problem occurs in communication systems, urban traffic systems, railway systems, multi-product production-distribution systems and military logistics systems. For an overview of MCNF problems see Tomlin (1966). The problem is defined on an directed graph $G = (V, A)$, where V is the set of nodes, and A is the set of arcs. The graph is not necessarily complete and usually quite sparse. Let x_{ij}^k be the flow of commodity k along arc (i, j). Let c_{ij}^k be the cost of arc (i, j) for commodity k, u_{ij} be the capacity that can flow through arc (i, j), which is the maximum flow of all commodities. Furthermore, let v_{ij}^k be the capacity for a single commodity. Let b_i^k be the source and sink information. If b_i^k is positive, node i is a source for commodity k and its value is its supply. If b_i^k is negative, node i is a sink for commodity k and its absolute value is its demand.

Let I_i be the set of predecessor nodes of node i and O_i the set of successor nodes of node i.

The MCNF can be formulated as follows

$$\min \sum_{k \in K} \sum_{ij \in A} c_{ij}^k x_{ij}^k$$

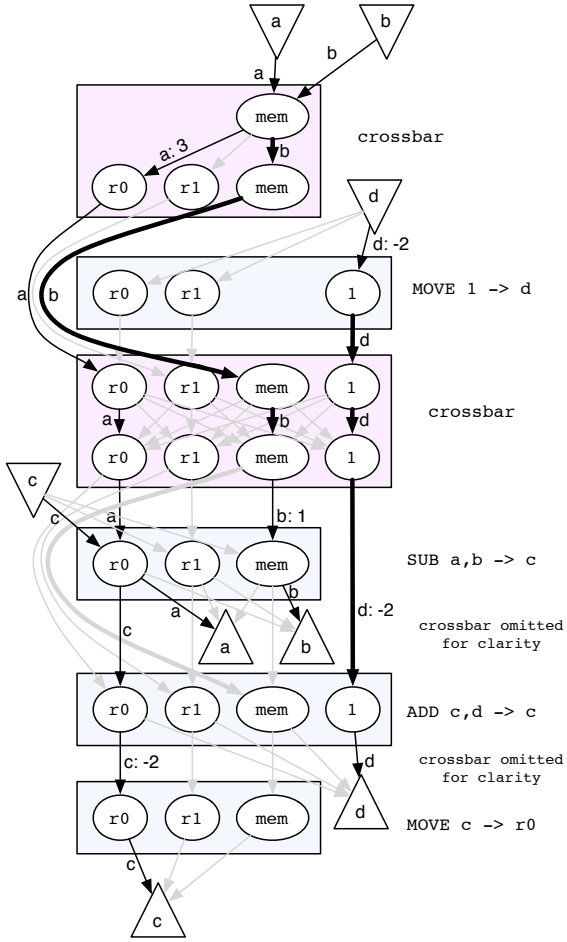
$$x_{ij}^k \leq u_{ij} \tag{3.1}$$

$$0 \leq x_{ij}^k \leq v_{ij}^k \quad \forall k \in K, \forall (ij) \in A \tag{3.2}$$

$$\sum_{j \in O_i} x_{ji}^k = b_i^k - \sum_{j \in I_i} x_{ij}^k \quad \forall (i) \in V, \forall k \in K \tag{3.3}$$

The objective function minimizes the total flow cost. Constraints (3.1) limit the total flow through arc (i, j) by the capacity u_{ij} and constraints (3.2) limit the total flow of commodity k through arc (i, j) Constraints (3.3) represent the inflow and outflow constraints.

The MCNF modeling can be used for local register allocation, which allocates only straight-line code. But there exist several techniques to expand the modeling so that it can also be used for global register allocation. The following example shows how to model the register allocation problem as a MCNF problem (see Figure 3.6).




```
int example(int a, int b)
{
   int d = 1;
   int c = a - b;
   return c+d;
}
```

Source code of example

```
MOVE 1 -> d
SUB a,b -> c
ADD c,d -> c
MOVE c -> r0
```

Assembly before register allocation

```
MOVE STACK(a) -> r0
SUB r0,STACK(b) -> r0
INC r0
```

Resulting register allocation

Figure 3.6.: Modeling of the register allocation problem as a MCNF problem as shown in Koes and Goldstein (2006)

The nodes represent the registers and the memory. In our example there are two registers and the memory, which is the stack, where an infinite amount of data can be stored. For every time unit a crossbar is introduced. A time unit corresponds to an instruction. A crossbar can contain multiples of the nodes to model copying between registers and/or between the memory. This technique assigns costs to the arcs and can therefore reinforce certain actions, like allocating values to a constant and so on.

The basic model does not take into account that a variable has to be copied only once to the memory. Since the storage at the memory is unlimited, it can be guaranteed that a value is kept in the memory even if it is copied to the register. By introducing a dummy variable, this can be modeled in the MNCF representation. Further details can be found in Koes and Goldstein (2005).

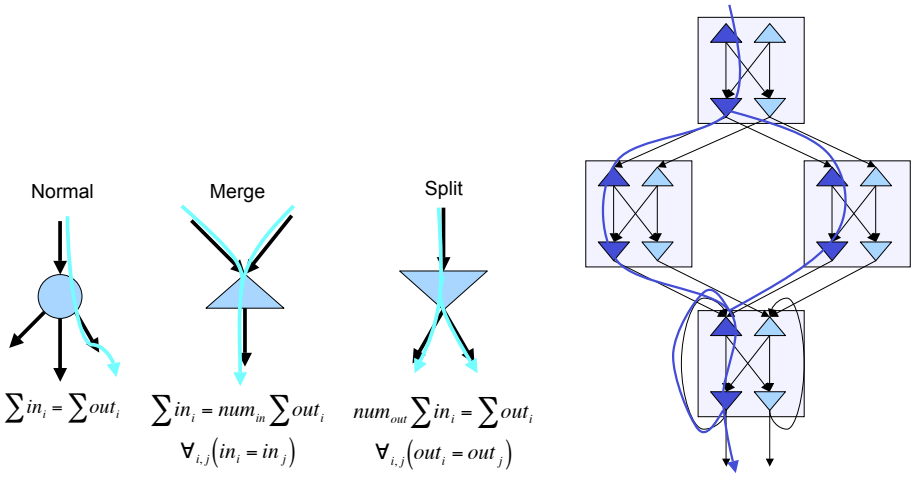
The necessary extension for the global allocation will be explained. The difference to the local allocation is that there is not only straight-line code any more, but there are branches, so it is necessary to introduce control flow. This can be represented by so-called merge and split nodes. These nodes are at the boundaries of a basic block. Merge nodes are at the beginning of a basic block, while split nodes are at the end of a basic block. Merge nodes are used for basic blocks that have more than one predecessors and split nodes for nodes that have more than one successor. It can be distinguished between three types of nodes: normal nodes, merge nodes and split nodes. Figure 3.7 (a) (taken from Koes and Goldstein (2005)) shows these three types of nodes. In Figure 3.7 (b) the merge and split nodes are included in the MCNF representation.

In Koes and Goldstein (2005), the authors used Lagrangian Relaxation as solution method to solve the MCNF problem, but also other exact or heuristic solution methods could be used.

This model can easily be extended to SSA-form. The extended version would have to include merge constraints for Φ-nodes that are similar to the merge nodes in Figure 3.7. Moreover, SSA-form deconstruction is included. However, the complexity of the model would not be reduced by the SSA-form.

The MCNF approach leads to good quality solutions but needs a long runtime. Memory coalescing and possibility of circular copies are not modeled. The objective function could be extended to take costs of copies into account and not

Figure 3.7.: Types of nodes in a global MCNF representation, taken from Koes and Goldstein (2005)



(a) Normal, merge and split nodes

(b) Merge and split nodes included in the MCNF representation

only the code size.

3.3. Linear Scan Algorithm - Second Chance Bin Packing

The linear scan algorithm was introduced in Poletto et al. (1997). An improved version, the so-called second chance bin packing was developed by Traub et al. (1998). The algorithm was implemented for the Java HotSpot™ Client Compiler in Wimmer (2004). The basic linear scan algorithm is described in detail in Poletto and Sarkar (1999). The algorithm is described as presented in Wimmer (2004).

The basic idea of the linear scan algorithm is to assign registers in a single pass over the linearized CFG. This means, that register allocation can be done in linear time with respect to size of the input nodes. The first step is to flatten the CFG. More precisely, the program is treated like straight line code even though branches occur. Therefore, the basic blocks need to be ordered. The sequence of the basic blocks plays an important role in the algorithm and effects the code quality.

Lifetime intervals are computed differently than in the other approaches. The interval starts at the definition and ends at the last use. Therefore, a variable is considered live from the first definition to the last use.

The basic linear scan algorithm is a greedy algorithm, because variables that occur first in the sequence are assigned to a register first and if no more registers are available the register with the farthest away lifetime end is spilled to memory. This is only useful if the variable having a long lasting lifetime is not used often during the lifetime. An easy way for improving the algorithm would be using estimated spill costs, which could take into account uses in inner loops with higher weights.

A lot of improvements to the basic algorithm have been proposed, which significantly increase the produced code quality but also need more runtime.

The basic algorithm works as follows. The compiler assigns a physical register to the first interval and continues to assign physical registers for the intervals in

the list. When no feasible assignment is possible, i.e. when there is no physical register available for the whole lifetime, some intervals must be spilled to memory. Two intervals that do not interfere, which means that their ranges do not intersect, can get the same physical register assigned. More precisely, the algorithm iterates over the sorted list of intervals. A so-called active list is kept, where all the intervals that overlap with the current position are stored. If the lifetime ends, the interval is removed from the list. Each interval currently in the list has a physical register assigned. If there are more intervals than registers, spilling has to be performed. The basic strategy is to spill the interval with the highest end position.

The advantage of the algorithm is the low runtime complexity. The drawback on the other hand is the very conservative view of the lifetime, which might lead to a worse allocation compared to more time consuming approaches.

An extension of the linear scan algorithm is second chance binpacking. Second chance binpacking deals with the drawbacks of the linear scan algorithm while providing almost the same runtime complexity.

As shown in the example in Figure 3.8, the basic linear scan algorithm does not take holes in the lifetime into ccount, which occur due to conditional branches and loops.

Figure 3.8.: Code example with live ranges for linear scan algorithm

In the linear scan algorithm, an interval that is spilled, is spilled for the whole lifetime. Splitting is not considered. Second chance binpacking also allows to split intervals. More precisely, an interval can be on a register for part of the lifetime and spilled later. It can also be spilled and moved back to a register, i.e. it gets a so-called second chance. The live ranges of the second chance binpacking are shown in Figure 3.9.

The linear scan algorithm does not consider the real control flow due to the linear ordering of the blocks. Therefore, inserted moves might not be feasible for all control flows. It is necessary to do a second pass that is called resolution. In this resolution phase, move instructions are inserted and a data flow analysis is used to minimize the number of moves. Therefore, the runtime complexity is higher than the one of the basic linear scan algorithm, because the data flow analysis cannot be performed in linear time. The linear scan algorithm with second chance bin packing produces results that are nearly as good as the ones of graph coloring. The algorithm is suited for cases where not only the runtime is important, but the compilation time is an issue.

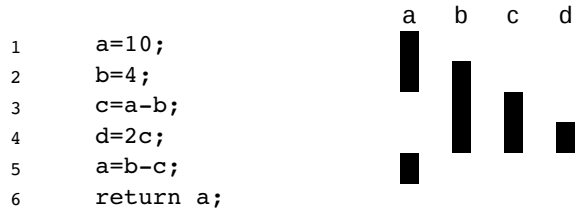


Figure 3.9.: Code example with live ranges for second chance binpacking

The solution to the register allocation problem of the basic linear scan algorithm is shown in Figure 3.10. First, a is assigned to register 1, then b is assigned to register 2 and c to register 3. Since there is no empty register for d, one variable has to be spilled. According to the heuristic rule that the interval with the highest end position should be spilled, a is spilled to the memory and d is put on register 1.

```
1  a=10;          //mem1=10
2  b=4;           //r2=4
3  c=a-b;         //r3=mem1-r2
4  d=2c;          //r1=2r3
5  a=b-c;         //mem1=r2-r3
6  return a;      //mem1
```

Figure 3.10.: Solution to the register allocation problem, constructed by the linear scan algorithm

The solution to the register allocation problem of the second chance bin packing algorithm is shown in Figure 3.11. In the second chance bin packing algorithm no interval has to be spilled to memory because the live range of a has a hole and therefore the three registers are sufficient.

```
1 a=10;          //r1=10
2 b=4;           //r3=4
3 c=a-b;         //r2=r1-r3
4 d=2c;          //r1=2r2
5 a=b-c;         //r1=r3-r2
6 return a;      //r1
```

Figure 3.11.: Solution to the register allocation problem, constructed by the second chance bin packing algorithm

The linear scan algorithm was adapted for programs in SSA-form by Wimmer and Franz (2010). They exploit the property, that lifetime holes of intervals in SSA-form always end at block boundaries and that intervals beginning at a lifetime hole of another interval cannot interfere. Generally, more and shorter intervals exist. Parallel Copies have to be inserted at SSA-form deconstruction which may lead to memory moves.

3.4. Partitioned Boolean Quadratic Optimization Problem

Another modeling approach for register allocation is a formulation as a partitioned boolean quadratic optimization problem (PBQP).

This modeling approach was developed by Scholz and Eckstein (2002) for highly irregular architectures such as digital signal processors. The approach is inspired by the the graph coloring approach.

A PBQP is defined as follows, given is a number n of symbolic registers and a number m of real physical registers. A $n \times (m+1)$ cost matrix $\mathbf{D}$ is defined, where d_{ik} represents the cost of locating the symbolic register i on the physical register k or on the spill slot. The cost can be set to ∞ if a given location is infeasible.

For each pair of symbolic registers, i and j, exists a $(m+1) \times (m+1)$ cost matrix $\mathbf{C_{ij}}$. The cost value c_{kl} states the cost if symbolic register i is assigned to physical register r and symbolic register j is assigned to physical register l. This cost matrix can model the interference between two symbolic registers. More precisely, if their lifetimes interfere, the according value in the cost matrix will be set to ∞. Moreover, register aliasing and the usage of more than one physical register for symbolic registers, as in case of double and long values, can be modeled in this cost matrix. Let x_i^{sp} be a binary variable that is 1 if the symbolic register i is spilled to memory, and 0 otherwise. Furthermore, let x_i^{Rk} be 1 if register k is used by symbolic register i.

$$\min \sum_{1 \leq i < j \leq n} \vec{x}_i \mathbf{C_{ij}} \vec{x}_j^T + \sum_{1 \leq i \leq n} \vec{d}_i \vec{x}_i^T$$

s.t.

$$\vec{x}_i \vec{1}^{\mathrm{T}} = 1 \quad \forall i = 1, .., n \tag{3.4}$$

The objective is to minimize total cost, i.e., the sum of the cost of the symbolic registers that are allocated to physical registers and those that have to be spilled to memory. Constraints (3.4) ensure that each symbolic register is allocated. The PBQP is NP-complete and solving it to optimality would take too much time for reasonable sized instances. The problem was solved by a heuristic, three-phased, dynamic programming approach in Scholz and Eckstein (2002).

3.5. Register Allocation by Puzzle Solving

Pereira and Palsberg (2008) proposed an abstraction to model register allocation by solving a set of puzzles. A puzzle has to be solved for every elementary program, i.e., every instruction. Figure 3.12 shows an example of an architecture with two registers and no register aliasing for the puzzle solving algorithm. For each register, there is a puzzle board consisting of one column and two rows. For more complicated architectures where register aliasing is allowed, more columns are needed.

A puzzle board consists of two rows. The upper row represents the register

before the instruction is executed, while the lower row represents the register after the instruction is execuded. For this puzzle board, three kinds of puzzle pieces exist. Type y uses both rows, while types x only uses the upper row and type z only covers the lower row. A puzzle is solved, if all pieces are placed on the puzzle board without overlaps. For different architectures, different types of puzzle boards and different kinds of pieces exist. Therefore, different algorithms have to be used to solve the puzzle for different architectures.

For the architecture shown, a simple linear time algorithm exists to solve the puzzle. More precisely, in the first step, all y pieces have to be placed. After that, all x pieces and all z pieces can be placed in an arbitrary order.

If the puzzle can be solved, a register allocation without spilling is found. Otherwise some variables have to be chosen for spilling. If spill free register allocation is not possible, a simple spilling heuristic is used.

For executing the puzzle solving algorithm, the program is converted to a sequence of elementary programs. This is done in the following way. First, the program is converted to SSA form. Then, the program is converted to a variation of the SSA form, where each variable live at the beginning of a basic block is renamed by a Φ-function. This program is further converted in the so-called static single information (SSI) form.

The SSI form was developed by Ananian (1999). The SSI form extends the SSA form. Additionally, at the end of a basic block, all variables that are live are renamed by a Π-function. Using the SSI form guarantees that a variable is only defined and used in one basic block. A Π-function renames all values that are live at the end of a block. It has the opposite effect of a Φ-function. Therefore, a good global register allocation can be obtained by local register allocation for every basic block and linking the results afterwards. For the puzzle solving algorithm this is not enough. Each basic block is splitted to a sequence of elementary programs, which only consist of single instructions. Therefore, parallel copies are inserted between consecutive instructions.

Figure 3.13 shows how a simple program can be mapped to puzzle pieces. The program has four instructions. For each instruction the puzzle is solved. Given are the puzzle pieces and the free positions in the board. If a value is defined in the instructions, it is mapped to a z piece. If its last use is in the instruction, it

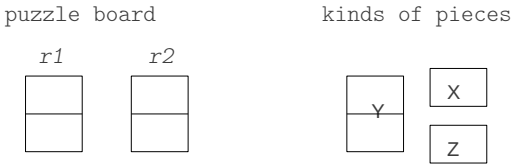


Figure 3.12.: Board and puzzle pieces for register allocation by puzzle solving

is mapped to a x piece. If the value is live, it is mapped to a y piece. In other words, if a value is *live-in* but not *live-out* at an instruction, it is mapped to an x piece, if it is only *live-out*, it is mapped to a z piece and if it is *live-in* and *live-out*, it is mapped to a y piece.

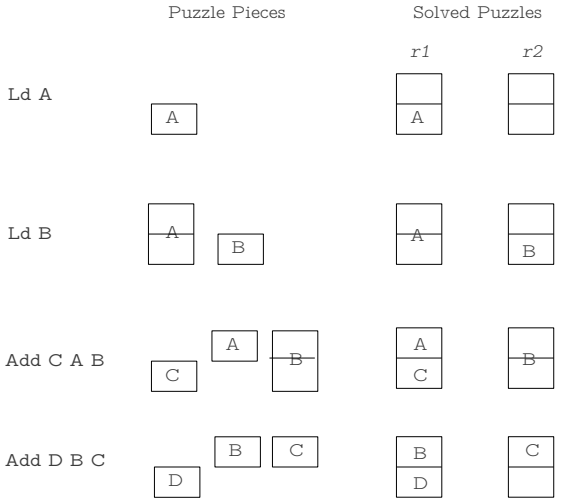


Figure 3.13.: Example for register allocation by puzzle solving

3.6. Integer Linear Program Based Approaches

This section deals with integer linear program (ILP) based approaches. ILPs are mathematical programs, where some decision variables are restricted to integral values, but all other constraints and the objective function are linear. (see Nemhauser and Wolsey (1999) for an introduction to integer programming). The

models can be solved with ILP solvers. First, an ILP model for register allocation is presented. Then, an ILP model for the spilling decisions is described.

3.6.1. ILP Model for Register Allocation

Goodwin and Wilken (1996) present an ILP model for register allocation. They formulate the problem as 0-1 integer programming problem and solve it with a MIP solver. They refer to their approach as optimal register allocator (ORA). The model can be used for a regular architecture and takes copy elimination, live range splitting, rematerialization, precolered registers and paired registers into account. The allocator is included in the Gnu C Compiler (GCC). The objective is to minize the total costs of inserting spill instructions. The model was first proposed by Goodwin and Wilken (1996), extended for irregular architectures by Kong and Wilken (1998) and improved by Fu et al. (2005).

First, an instruction graph is constructed from the CFG. From this construction graph, a symbolic register graph is constructed for each symbolic register. A symbolic register is the same as a virtual register, where each variable is assigned to a symbolic register and the fact that only a limited number of registers is available is ignored. A symbolic register graph is shown in Figure 3.14 and Figure 3.15 for a sample program. Three instructions are used in the sample program.

Figure 3.14 shows the symbolic register graph for variable A. The graphs starts at the definition and ends at the last use. Since the last use is not defined, the live time is continued. Figure 3.15 shows the same for variable B. The graph starts at the definition and ends at the last use, which is $c = a + b$ in the sample program. The allocation and deallocation decisions are represented by the edges in the graph. The ILP is based on the insight that in the optimal solution only non-dominated allocation and deallocation positions are present. Therefore, it is sufficient to only consider non-dominated positions. These non-dominated positions can be identified in the symbolic register graph.

To identify these positions, the authors distinguish between load, store and deallocation decisions. Load decisions are placed before every symbolic register use and at the merge of basic blocks. Store decisions can be placed after a

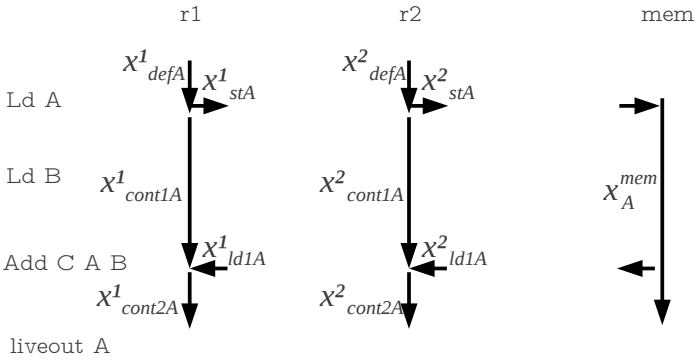


Figure 3.14.: Symbolic register graph for symbolic register A

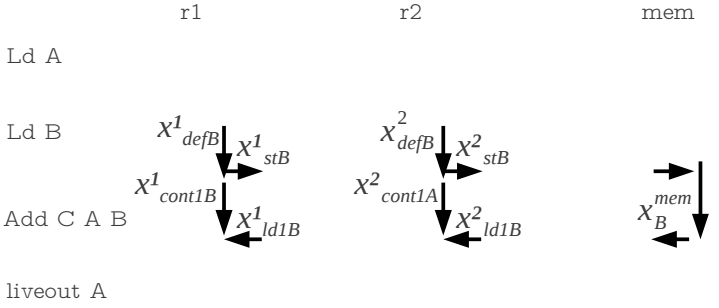


Figure 3.15.: Symbolic register graph for symbolic register B

symbolic register definition and at the end of a basic block which has several sucessors, i.e., it diverges. Finally, deallocation decisions can be placed after every definition, after every use and at every ORA symbolic register graph diverge. In the sample program, we can identify load, store and deallocation decisions.

Let x^r_{sp} be a set of binary decision variables. x^r_{sp} is 1 if symbolic register s at location p is assigned to real register r, and 0 otherwise.

In the following the ILP is explained as presented in Fu et al. (2005) and the different types of constraints are described.

$$\min \sum_s \sum_r c^r_{sp} x^r_{sp}$$

s.t.

$$\sum_r x^r_{sp} \geq 1 \tag{3.5}$$

$$\sum_s x^r_{sp} \leq 1 \tag{3.6}$$

$$\sum_r x^r_{sp} + x^{mem}_{sp} \geq 1 \tag{3.7}$$

$$x^r_{sp-1} \geq x^r_{sp} \tag{3.8}$$

$$x^r_{sp} = x^r_{sp'} \tag{3.9}$$

$$x^r_{sp} \in \{0, 1\} \tag{3.10}$$

Constraints (3.5) are so-called *must-allocate* constraints. They ensure that for every definition and use of a symbolic register, a real register must be allocated. For example, $x^1_{defA} + x^2_{defA} \geq 1$ in the sample program ensures that register one or register two is chosen for the definition of variable A. Constraints (3.6) are the *single-symbolic* constraints, that state that a real register can be allocated to not more than one symbolic register for any point in the program. For example, $x^1_{cont1A}+x^1_{defB} \leq 1$ because the definition of B is at the same time as the allocation of x^1_{cont1A}. So these two allocation decisions cannot be made for the same real register.

Constraints (3.7) are the *liveness* constraints guarantee that a symbolic register is live in either a real register, the memory or in both for every point in the program where the variable is live. For example the inequality $x^1_{cont1A} + x^2_{cont1A} + x^{mem}_A \geq 1$ has to hold for the sample program. The *deallocation* constraints (3.8) ensure that variables can deallocate at certain positions. An example constraint is $x^1_{defA} \geq x^1_{cont1A}$. Finally, the *merge* constraints (3.9) make sure that at a merge vertex all incoming edges have the same allocation states, i.e., the symbolic registers must be allocated to the same physical registers for each predecessor of a merge. Constraints (3.10) ensure that each allocation decision is either zero or

one.

3.6.2. ILP Model for Optimal Spilling

Appel and George (2001) decompose the problem into a spilling decision and a register assignment decision. In the spilling phase, it is decided when and where variables are spilled to the memory. This phase is solved to optimality with an ILP. Usually spilling is expensive and should therefore be planned first. Once the first stage is done and an optimal solution to the spilling problem is found, it is guaranteed that a spill free register allocation can be found. However, this is not a trivial task, because usually this is not possible without live range splitting. So a traditional graph-coloring based register allocator could still find variables to spill.

The decomposition makes the algorithm faster. However, as mentioned above, an optimal allocation cannot be guaranteed.

4. Register Allocation in the Java HotSpot™ Server Compiler

This chapter gives an overview of register allocation in the Java HotSpot™ Server Compiler for the irregular IA-32 architecture. Moreover, the existing implementation of the register allocator is described. Then, a simple stack based register allocator that was implemented is presented.

4.1. Detailed Problem Description

In this chapter a detailed overview of the task of register allocation in the Java HotSpot™ server compiler for the IA-32 architecture is given. The IR of the optimized program in SSA-form is given. Due to heavy optimization the IR is not in CSSA, which makes register allocation, spill slot assignments and translating out of SSA a lot harder. It is possible to manipulate the IR by inserting new move instructions and updating the input edges. The sequence of the existing nodes should not be altered in the register allocator phase. It is required to map each node defining a value to a physical register or stack position. The IR is based on SSA-form and therefore it has to keep the Φ-nodes also after register allocation. However, they are redundant because all input nodes and the Φ-node itself have to be mapped to the same position. Therefore, the live ranges involved in a Φ-node are not allowed to overlap, which implies that the program is in CSSA-form.

In the following, the term position will be used for a physical register or a stack position. Table 4.1 shows a simple input program to the register allocator. In the first two columns the labels and the *asm* code is given. Columns three to five state the information obtained from the IR, which is relevant to register

allocation. As we are only dealing with straight line code in this example, all nodes will be in a linear order within a basic block. Column three shows the operation that is performed in the node and the index of the node. For register allocation, the type of node is not a relevant information, but the index of the node is important. Since the program is in SSA-form, every value that is defined, has to be assigned to an unique name. As labels are unique names, every unique name corresponds to a label. The index of the node can be seen as the label of the node. The unique name can be seen as a symbolic register that has to be assigned to a physical register or stack position. Therefore, the input to the register allocator would be a feasible assignment if there would be an infinite number of registers and no register constraints would exist. Column four shows the possible registers for the value that is defined by the node and column five and six show the possible input registers for the respective input operands. The input operands are given as a pointer to the label where the respective value was defined. The index of the input nodes is shown in parentheses. Finally, in the last column, a feasible solution is shown, which is the output of the register allocator. Each node that defines a value to a position has to be mapped.

A node can either define a value, or use other values or none of both if it is only used for the control flow. A node defining a value holds a set of feasible registers or stack positions. In the IR of the server compiler the information is given by a bitmask called *out_RegMask*. The last bit stands for any stack location greater than the last position in the register mask. Fat projection nodes are used to kill values in certain positions. For example after a method call the values in all registers become invalid, because they may have been overwritten by the called method. This is modeled by fat projection nodes. All positions in their *out_RegMask* are killed.

An example, problem 1, is given in Table 4.1. In this example there are only at most two input operands, but in general an infinite number of input operands is possible. For example, function calls and save points have a lager number of input operands. In the following, the format of the input data is described and the irregularities of the IA-32 architecture and how they are included to the input data are presented.

Precoloring. In the example we can see an irregularity of the architecture, because in label four the DIV instruction needs both the output value and the first input value to be in register two. This is called precoloring. The term was coined in the context of graph coloring based register allocation and means that some operands of instructions are constrained to a fixed register. Such instructions are division and shift operators. This means for example that for a shift operation the shift count must always be in the register *ecx*. Parameters are also passed in fixed registers and fixed stack slots, and the return value is always on *eax*.

A feasible solution is computed to problem 1. In terms of feasibility it is important that the live ranges of values that are assigned to the same register, do not interfere and all input and output register constraints are fulfilled. In our example, label one and label four are assigned to register two. The assignment is feasible, because they do not interfere. Concerning the input and output constraints, label one has register two in the output register mask and also for every usage, register two is contained in the input register mask. Node one is used as input operand one in label three and four in the example.

label	ASM	nodes,idx	out regmask	INP1	INP2	Reg
l1	LD a	LD,1	{1,2,3,4}			2
l2	LD b	LD,2	{1,2,3,4}			3
l3	ADD c,a,b	ADD,3	{1,2,3,4}	(1) {1,2,3,4}	(2) {1,2,3,4}	4
l4	DIV d,a,b	DIV,4	{2}	(1) {2}	(2) {1,2,3,4}	2
l5	CMP c,d	CMP, 5		(3) {1,2,3,4}	(4) {1,2,3,4}	

Table 4.1.: Problem 1: Sample input for the register allocator - straight line code

In the next example in Table 4.2 a return node is inserted. The input operand of the return node can only be located in register one. However, when the value was defined, in label four, the output register was restricted to register two. Therefore, it is necessary to introduce a move operation, in order to get a feasible register allocation. In this example it would be possible to assign a real register to each symbolic register without any violation of the live range interference constraints, but because of register constraints due to the irregular architecture a move instruction needs to be inserted.

label	ASM	nodes,idx	out regmask	INP1	INP2	Reg
l1	LD a	LD,1	{1,2,3,4}			2
l2	LD b	LD,2	{1,2,3,4}			3
l3	ADD c,a,b	ADD,3	{1,2,3,4}	(1) {1,2,3,4}	(2) {1,2,3,4}	4
l4	DIV d,a,b	DIV,4	{2}	(1) {2}	(2) {1,2,3,4}	2
l5	RET d	RET,5		(4) **{1}**		?

Table 4.2.: Problem 2: No feasible allocation possible

This is shown in Table 4.3, where a new node was inserted (label six) that moves d from one position to another position. Now a feasible allocation can be obtained.

label	ASM	nodes,idx	out regmask	INP1	INP2	Reg
l1	LD a	LD,1	{1,2,3,4}			2
l2	LD b	LD,2	{1,2,3,4}			3
l3	ADD c,a,b	ADD,3	{1,2,3,4}	(1) {1,2,3,4}	(2) {1,2,3,4}	4
l4	DIV d,a,b	DIV,4	{2}	(1) {2}	(2) {1,2,3,4}	2
l6	MOV e,d	MOV,5	{1,2,3,4,st}	(4) {1,2,3,4,st}		1
l5	RET e	RET,5		(6) {1}		1

Table 4.3.: Problem 2: Insertion of a move, to obtain a feasible allocation

Two Operand Form. In the IA-32 there are more irregularities. One of them is that the instructions need to be in the two operand form. This means that the position of the left operand is always the position of the result operand. For example the following operation is supported

```
x += a;
```

while the operation

```
x = a+b;
```

must be rewritten as

```
x = b;
x += a;
```

The intermediate representation of the server compiler is in the three operand form, but additional information is given to model the two operand form. Table 4.4 shows an example where the additional information is represented by a

new column SR. If there is an entry i, the position of the i-th input operand is forced to be the same as the position of the output operand. The two operand form is necessary for label three and label four in the example problem 3, in Table 4.4. For label three, a, which is the first input operand, has to be on the same position as c. For label four, the second input operand, a, has to be on the same position as d. Since the life ranges in our example interfere, a move has to be inserted to enable a feasible allocation. This is shown in Table 4.5.

label	ASM	nodes,idx	SR	out regmask	INP1	INP2	Reg
l1	LD a	LD,1		{1,2,3,4}			1
l2	LD b	LD,2		{1,2,3,4}			2
l3	ADD c,a,b	ADD,3	1	{1,2,3,4}	(1) {1,2,3,4}	(2) {1,2,3,4}	1
l4	ADD d,c,a	ADD,4	2	{1,2,3,4}	(3) {1,2,3,4}	(1) {1,2,3,4}	?

Table 4.4.: Problem 3: Two operand form

label	ASM	nodes,idx	SR	out regmask	INP1	INP2	Reg
l1	LD a	LD,1		{1,2,3,4}			1
l2	LD b	LD,2		{1,2,3,4}			2
l5	MOV e,a	MOV,5		{1,2,3,4,st}	(1) {1,2,3,4,st}		3
l3	ADD c,e,b	ADD,3	1	{1,2,3,4}	(1) {1,2,3,4}	(2) {1,2,3,4}	3
l4	ADD d,c, a	ADD,4	2	{1,2,3,4}	(3) {1,2,3,4}	(1) {1,2,3,4}	1

Table 4.5.: Problem 3: Insertion of a move, to obtain a feasible allocation

CISC Spilling. For instructions that use two input operands, the second one may reside in the memory in certain architectures. This is called CISC-spilling in the IA-32 architecture. For example in $x = b + a$, a can be on memory without being reloaded to a register. Table 4.6 shows an example for CISC-spilling. Information about which operand can reside on the stack is given in the IR. This is represented by the column CS in our example, problem 4. If the stack position is chosen, the instruction will take more time than if a register is chosen. Therefore, it is preferred to use a register position. However, if the operand is already on the stack, it is faster to use CISC spilling and no register needs to be occupied for the operand.

Long Values. In the IA-32 architecture, long values with 64-bits need two registers to store the value. In the Sparc architecture these registers have to

label	ASM	nodes,idx	SR	CS	out regmask	INP1	INP2	Reg
l1	LD a	LD,1			{1,2}			1
l2	LD b	LD,2			{1,2}			2
l3	LD c	LD,3			{1,2}			?
l4	ADD d,a,b	ADD,4	1	2	{1,2}	(1) {1,2}	(2) {1,2}	?

Table 4.6.: Problem 4: CISC Spilling

label	ASM	nodes,id	SR	CS	out regmask	INP1	INP2	Reg
l1	LD a	LD,1			{1,2}			1
l2	LD b	LD,2			{1,2}			2
l5	MOV e,b	MOV, 5			{1,2,st}	(2) {1,2,st}		st
l3	LD c	LD,3			{1,2}			2
l4	ADD d,a,b	ADD,4	1	2	{1,2}	(1) {1,2}	(2) {1,2}	1

Table 4.7.: Problem 4: Insertion of a move and usage of cisc spilling

be two adjacent physical registers. Therefore, in the Java HotSpot™ server compiler the two registers have to be adjacent too. This information is again given as additional information to a node. In problem 5, in Table 4.8, the values defined in label one and two are long values. Therefore, they need two adjacent registers. Since label one needs register one and two, label two cannot be stored. In Table 4.9 an additional move is inserted that moves label one to the stack so that label two can be stored on register one and two. In the add instruction in label three, CISC spilling is used.

label	ASM	nodes,id	SR	CS	Long	out regmask	INP1	INP2	Reg
l1	LD a	LD,1			L	{1,2}			1,2
l2	LD b	LD,2			L	{1,2}			?
l3	ADD c,b,a	ADD,3	1	2		{1,2}	(2) {1,2}	(1) {1,2}	?

Table 4.8.: Problem 5: Use of long values

Control Flow. Table 4.10 shows an example with control flow (Problem 6). Φ-nodes are in the places where the control flow merges. A special property of Φ-nodes is that all input and output operands have to be on the same position. Otherwise move instructions have to be inserted to the predecessor blocks. Due to this property, the Φ-instructions can be omitted in the final translation to machine code after register allocation. More precisely, in the example there is a

label	ASM	nodes,id	SR	CS	Long	out regmask	INP1	INP2	Reg
l1	LD a	LD,1			L	{1,2}			1,2
l4	MOV d,b	MOV,4				{1,2,st}	(2) {1,2,st}		st
l2	LD b	LD,2			L	{1,2}			1,2
l3	ADD c,b,d	ADD,3	1	2		{1,2}	(2) {1,2}	(4) {1,2}	1,2

Table 4.9.: Feasible allocation of problem 5

Φ-function $d := \Phi(a, c)$. This means that d will be a copy of either a or c and that they all have to be on the same position, i.e., either register one, register two or the same stack slot.

Block	pred	label	ASM	nodes,idx	out regmask	INP1	INP2	Reg
B1								
		l1	LD a	LD,1	{1,2}			1
		l2	LD b	LD,2	{1,2}			2
		l3	CMP a,b	CMP,3		(1) {1,2}	(2) {1,2}	
B2	B1							
		l4	ADD c,a,b	ADD,4	{1,2}	(1) {1,2}	(2) {1,2}	1
B3	B1,B2							
		l5	PHI d,a,c	PHI,5	{1,2,st}	(1) {1,2,st}	(4) {1,2,st}	1
		l6	RET d	RET,6	{1}			

Table 4.10.: Problem 6: Control flow

Fat Projection. The fat projection is a special instruction that is not an irregularity of the IA-32 architecture, but it is part of the IR of the Java Hotspot™ server compiler. It is a dummy instruction that is used to model deletion of values in a register. All values located in registers and stack positions that are defined in the *out_RegMask* of the fat projection are deleted. Fat projections are used after method calls to model that the registers may be overwritten by instructions in the called method. Therefore, values that are in register, which are killed by the fat projection need to be copied by the register allocator.

4.2. Existing Implementation

The basic steps of the existing register allocator are the following:

DeSSA: In this phase additionally to the nodes in the SSA-form based IR, live ranges are built, which are not necessarily in SSA-form. The structure of

live ranges will be updated parallel to the IR, which will always stay in SSA-form. The IR and the live ranges are used and manipulated in the following phases of the register allocator, but both are always kept up to date in all phases of the register allocator.

Liveness Analysis: The liveout sets of live ranges for all basic blocks are computed. Because the live ranges might not be in SSA-form, the SSA-structure of the IR cannot be exploited.

Construct IFG: In this phase the interference graph (IFG) is constructed based on the information of the liveness analysis.

Coalesce: Optimistic and pessimistic coalescing is done in this phase.

Simplify: The IFG is simplified by removing live ranges with a low number of interferences because for these live ranges feasible colors can be guaranteed.

Select: Registers are assigned to the live ranges by reinserting them to the IFG. If no feasible coloring is possible, live ranges are marked for spilling.

Split: Live ranges marked for spilling are split in this phase. Due to shorter live ranges coloring will be easier in the next iteration. After this phase the liveness analysis has to be done again.

This steps are repeated until no live ranges need to be spilled any more.

4.3. Basic algorithm for feasible allocation

A simple stack based register allocator, that takes into account all irregularities of the IA-32 architecture was implemented and is described in this section. The existing Chaitin-Briggs graph coloring allocator can be fully or partly replaced by the new allocator.

In the basic algorithm all the values defined are copied to a unique stack position. For each instruction that needs the value on a register it has to be copied to a register again.

The algorithm consists of two steps. First the IR is parsed and all necessary information is gathered. In the second step move instructions from register to stack or from the stack to a register are added to the IR and the virtual registers are replaced by stack slots or real registers. The first step is described in Algorithm 1, while the second step is shown in Algorithm 2.

Algorithm 1 Determination of all virtual registers and register constraints

```
for all basic blocks b_i ∈ B do
  for all nodes n_ij ∈ b_i do
    if n_ij defines value then
      Add new virtual register
      Gather live range information (two operand form, long)
    end if
    for all input nodes p ∈ In(n_ij) do
      if p is a virtual register then
        add new use to virtual register p
        gather use information (CISC spillable, two operand form)
      end if
    end for
  end for
end for
```

For each virtual register defined, a stack position with the necessary size is assigned. Furthermore, information about register constraints is collected. Then it is determined if the instruction requires the output in the same register as a certain input. Fat projections, which overwrite registers do not need to be considered in the algorithm, because a valid copy of the register is always available on the stack.

Algorithm 2 describes how moves are inserted and feasible registers or stack slots are chosen. Directly after the definition of a virtual register the value is copied to the stack. Because of the SSA-form property, it is guaranteed that a valid copy of the virtual register is always available on the stack. Moves from the stack slot to a register need to be inserted directly before each use, that requires a register. The registers are chosen as follows. First uses with register constraints (precolered registers) are handled. Then all other uses are replaced by a free register and a move from the stack to the assigned register is added. Uses that

do not require a register are for example parameters of method calls, safepoints or operands that can be used for CISC spilling. In this cases the virtual register is simply replaced by the stack slot.

Algorithm 2 Assign registers and insert moves

```
for all virtual registers do
  insert move to unique stack position right after definition
end for
for all basic blocks b_i ∈ B do
  for all nodes n_ij ∈ b_i do
    for all precolored input nodes p ∈ In(n_ij) do
      Insert Copy from Stack position
      Assign precolored register to the copy
    end for
    for all other input nodes p ∈ In(n_ij) do
      if Stack postion feasible (CISC spilling, parameter, safepoint) then
        Use stack copy
      else
        Insert Copy from Stack position
        Assign feasible (and not yet used) register
      end if
    end for
    if n_ij defines value then
      Assign feasible register
    end if
  end for
end for
```

The algorithm was tested on the SPECjvm2008 benchmark instances (Standard Performance Evaluation Corporation (2008)) and results show that the stack based allocator delivers code that is 2.2 times slower than the code of the existing register allocator. This shows the high potential of a good register allocator.

5. Mathematical Program for Optimal Register Allocation for the Java HotSpot™ Server Compiler

A mathematical program formulation for the task of global register allocation is presented. The goal is to define the task of register allocation in an exact mathematical way. We can formulate the discrete optimization problem as an Integer Linear Program (ILP).

Although it would be possible to find the optimal allocation for rather small instances using a (commercial) ILP solver, it would not be of practical relevance because of the high computation time. However, it defines the task of register allocation exactly and improvements of the formulation can yield a tractable model that can be solved exactly or heuristically in reasonable time.

ILP models for register allocation have been proposed by Goodwin and Wilken (1996) and Kong and Wilken (1998), but they do not take spill slot assignments, memory coalescing and SSA-form into account. To the best of our knowledge this is the first mathematical model for SSA-form based register allocation for an irregular architecture.

In our model we are using the same information as the register allocator of the server compiler in the actual implementation. Most information is obtained from the IR in SSA-form. Moreover, the loop depths and expected execution times of the basic blocks are used for estimating the costs for moving and rematerializing data. The liveness analysis is done in the ILP implicitly and does not need to be computed beforehand.

The formulation takes into account the various special constraints and properties of register allocation for the IA32-architecture. These properties are precolored values, values that color two adjacent registers, CISC spilling and the two operand form, where the register of the first input operand has to equal the register of the output operand.

The problem can be separated into two main parts. The first part consists of the moves and rematerializations within a basic block and the second of linking the basic blocks, which consists of placing input values of Φ-functions and other values that are live between block boundaries to appropriate positions. This separation between the two parts will be used in the structure of our formulation.

The basic components of an ILP are the decision variables, the input data, the constraints and finally the objective function. Therefore, the first step is to identify the decision variables and constraints to obtain a feasible model.

5.1. Identification of Decision Variables and Constraints

This section is illustrated by using examples. The first example is shown in Table 5.1. One basic block is considered, $B = \{1\}$. Furthermore, we are given seven labels or nodes, l1-l7, and four values, $V = \{a, b, c, d\}$. An architecture similar to the IA-32 is considered, but with fewer registers. The architecture in the example consists of two registers (r1 and r2). The number of stack positions is assumed to be infinite. However, this number is limited by a trivial upper bound, which is the number of values in the program ($|V| = 4$ in the sample program).

Additionally, we need a special position, which is used for cisc spilling. A move to this register represents the cost of using the cisc spill option instead of forcing the value to be on a register. The set of possible positions, P, consists hence of the two registers, the four stack positions and the position for cisc spilling, $P = \{r1, r2, s1, s2, s3, s4, c\}$. The set $P' \subset P$ is the set of all registers and stack positions without the cisc spill position, $P \backslash \{c\}$.

For each node (label) j, we define a number of move iterations, $m = 0, .., m_{\max}$.

In each move iteration m a single move is performed. Hence, $m = 0$ represents the first move after the label is executed. The register allocator is basically allowed to insert an infinite number of moves, but an upper bound for $m_{\max}$ in the optimal solution is $|V||P|$.

In the example in Table 5.1 we have *fat projection*, *add*, *call* and *div* operations in the labels. For every label, the output register mask and the input register masks are given. All instructions that need two input operands are in two-operand form, which means that the position of the input operand equals the position of the output operand.

label	ps. ASM	idx	SR	CS	out regmask	INP1	INP2
l1	LD a	1			$\{r_1,r_2\}$		
l2	LD b	2			$\{r_1,r_2\}$		
l3	ADD c,a,b	3	1	2	$\{r_1,r_2\}$	(1) $\{r_1,r_2\}$	(2) $\{r_1,r_2\}$
l4	CALL a,b	4				(1) $\{r_2\}$	(2) $\{s_1\}$
l5	FATP	5			$\{r_1,r_2,\ s_1\}$		
l6	DIV d,b,a	6	1	2	$\{r_2\}$	(2)$\{r_2\}$	(1)$\{r_1,\ r_2\}$
l7	RET d	7				(6) $\{r_1\}$	

Table 5.1.: Sample instance for the IP model

The following decision variables are used in the mathematical model. Let x_{ijm}^{vk} be 1 if value v is in position k in block i at label j in move iteration m, and 0 otherwise. Let z_{ijm}^{vkl} be 1 if value v is copied from position k to position l in block i at label j in move iteration m, 0 otherwise. Figure 5.1 illustrates the role of the decision variables. The figure shows possible states of the registers and stack positions for a given basic block, $i = 1$, and a given label, $j = 3$, and how these states are represented by the decision variables. There are four move iterations, $m_{\max} = 3, m = 0, ..., m_{\max}$.

Right after the execution of the instruction of label three, the move iteration index m is 0. We assume that the values in the registers and stack positions are as seen in the figure. The value a is located in register one. Therefore, the decision variable x_{130}^{a1} equals 1. All other decision variables for register one at block one, label three and move iteration zero equal 0, i.e., $x_{130}^{b1} = x_{130}^{c1} = x_{130}^{d1} = 0$. Hence, the following constraint has to hold.

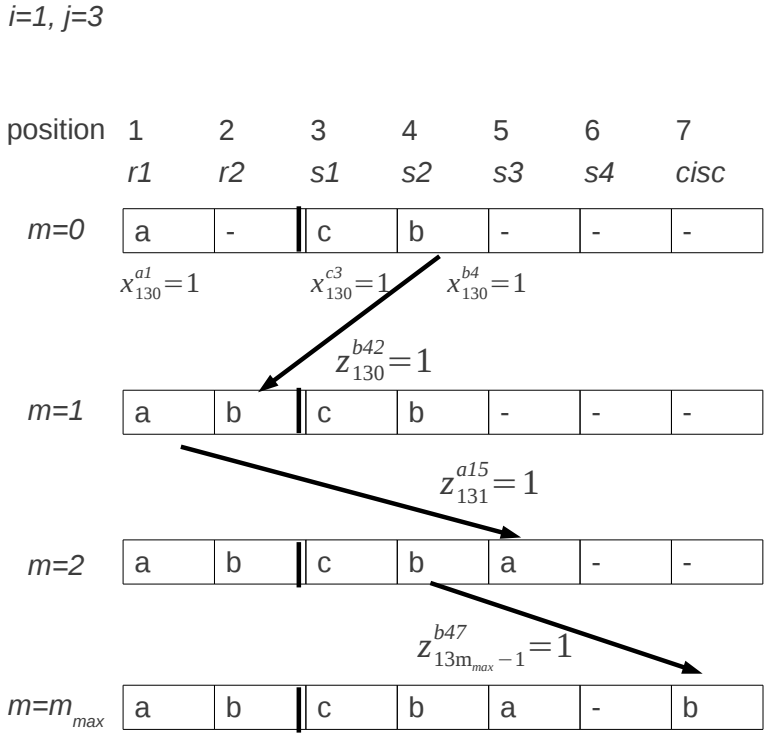


Figure 5.1.: States of registers and stack positions during move iterations

$$x_{130}^{a1} + x_{130}^{b1} + x_{130}^{c1} + x_{130}^{d1} \leq 1 \tag{5.1}$$

Therefore, we can identify the first set of constraints for the model in a general form. Constraints (5.2) state that a position at a given time slot can hold at most one value. B represents the set of basic blocks, N_i is the set of nodes of block i and M is the set of move iterations.

$$\sum_{v \in V} x_{ijm}^{vk} \leq 1 \quad \forall i \in B, j \in N_i, m \in M, k \in P \tag{5.2}$$

In register two, there is currently no value and all decision variables equal 0. At the stack position s_1 is the value c and in s_2 there is b.

Between the two move iterations we are allowed to insert at most one move. The value b is moved from s_2 to r_1 in Figure 5.1. Therefore, the decision variable z_{130}^{b42} equals 1. Moves of values can only be performed from a register if the value is actually in the register. Therefore, the following constraint has to hold in our example

$$z_{130}^{b42} \leq x_{130}^{b4} \tag{5.3}$$

We can identify the next set of constraints in a general form for the model. Constraints (5.4) ensure that values can only be copied from a position if they are stored at that position. M'' represents the set of move iterations excluding the last iteration $m_{\max}$.

$$z_{ijm}^{vkl} \leq x_{ijm}^{vk} \quad \forall i \in B, j \in N_i, m \in M'', k, l \in P \tag{5.4}$$

It is only allowed to perform a single move in one iteration because otherwise circular shifts could occur. A simple circular shift is for example a swap of two registers. This is the smallest circular shift that can occur. If a swap is performed by executing two moves, the value of one register is overwritten. Therefore, it is not possible to map a circular shift to a feasible register allocation without using a spare register which would cause additional costs that are not modeled in the IP model. This is ensured by the following constraint that has to be satisfied in

the example

$$\sum_{v \in V} \sum_{k \in P} \sum_{l \in P} z_{130}^{vkl} \leq 1 \tag{5.5}$$

In a general form, constraints (5.6) ensure that there are no circular shifts.

$$\sum_{v \in V} \sum_{k \in P} \sum_{l \in P} z_{ijm}^{vkl} \leq 1 \quad \forall i \in B, j \in N_i, m \in M'' \tag{5.6}$$

It is furthermore necessary to ensure that states of the positions can only be changed by copy instructions between two consecutive slots m and $m+1$. For example, the value a can only be in register 1 in time $m = 1$, if it has been in register 1 in time $m = 1$ or if it has been copied from another position in this iteration. Therefore, we can state

$$x_{131}^{a1} \leq x_{130}^{a1} + \sum_{k \in P} z_{130}^{ak1} \tag{5.7}$$

In a general form, constraints (5.8) link the move iterations to each other. M' represents the set of move iterations excluding the first with the index 0.

$$x_{ijm}^{vl} \leq x_{ijm-1}^{vl} + \sum_{k \in P} z_{ijm-1}^{vkl} \quad \forall i \in B, j \in N_i, m \in M', v \in V, l \in P \tag{5.8}$$

The state of the positions before and after a label may only be altered by an instruction defining a new value. For example, the state in $m = 0$ of label three is defined by the state of label two in $m = m_{\max}$ plus the changes that were made by the instruction in the label. The following constraint can be stated for our example.

$$x_{130}^{vk} \leq x_{12m_{\max}}^{vk} \quad \forall v \in V \backslash \{c\}, k \in P \tag{5.9}$$

The value c is defined at label three, therefore the following constraint has to be respected.

$$\sum_{k \in \{1,2\}} x_{130}^{ck} = 1 \tag{5.10}$$

For the fat projection, a special constraint has to be respected, so that all the values in the output register mask are killed. For example, for the fat projection in label five, the following constraint has to be ensured.

$$\sum_{v \in V} \sum_{k \in \{1,2,3\}} x_{150}^{vk} = 0 \tag{5.11}$$

These constraints can be stated in a general form.

$$x_{ij0}^{vk} \leq x_{ij-1M}^{vk} \quad \forall i \in B, j \in N_i \backslash \{0\}, v \in V \backslash \{Def(i,j)\}, k \in P \tag{5.12}$$

Constraints (5.12) ensure that only instructions can change the states before and after a label, where $Def(i,j)$ is the value that is defined by node ij, which was c in the example above.

$$\sum_{k \in O(Def(i,j))} x_{ij0}^{ck} = 1 \quad \forall i \in B, j \in N_i' \tag{5.13}$$

Constraints (5.13) state that one position in the output register mask of a label has to be chosen to store the defined value of a label. O represents the positions that are given in the output register mask and N_i' is the set of nodes of block i that define a value.

$$\sum_{v \in V} \sum_{k \in O(ij)} x_{ij0}^{vk} = 0 \quad \forall i \in B, j \in F_i \tag{5.14}$$

Constraints (5.14) ensure that the values in the output register mask are killed in the fat projection. F_i represents the set of fat projection nodes of block i.

For the labels, it is also necessary to ensure that the values of the required input operands are in the appropriate position. For example, for input operand a in label three, a is required to be either in register one or in register two before label three. The following constraint ensures this property.

$$\sum_{k \in \{1,2\}} x_{12m_{\max}}^{ak} \geq 1 \tag{5.15}$$

This can be stated in a general form as follows.

$$\sum_{k \in I(v,i,j)} x^{ak}_{ijm_{\max}} \geq 1 \quad \forall i \in B, j \in N_i, v \in Inp(i,j) \tag{5.16}$$

Constraints (5.16) ensure that the input operands are in a feasible position as defined in the corresponding input regmask, where $I(v,i,j)$ represents the input regmask of value v of node ij and $Inp(i,j)$ is the set of all the input operands of node ij. Note, that this set may be empty.

Furthermore, it is important that the irregularities of the IA-32 architecture are respected. These include the two operand form, cisc spilling and long values, that were explained in Chapter 4.

The two operand form occurs in the example in label three, where the position of input operand one, a, has to equal the position of the output operand, c. This can be ensured by the following constraints.

$$x^{ak}_{12m_{\max}} \geq x^{ck}_{130} \quad \forall k \in P \tag{5.17}$$

In a more general form the constraints are

$$x^{uk}_{ij-1m_{\max}} \geq x^{vk}_{ij0} \quad \forall i \in B, j \in N''_i, v = Def(i,j), u = Inp'(i,j), k \in P \tag{5.18}$$

$Inp'(i,j)$ represents the input node which position has to be the same as the position of the defined value.

For input operands where cisc spilling is allowed, the input register mask is extended by the artificial cisc position. A move to this position models the cost of cisc spilling. The cisc spill register is deleted after each label again, because it cannot be used for storing values, but only for simulating cisc spilling.

$$x^{vc}_{ijm} = 0 \quad \forall v \in V, i \in B, j \in N, m \in M' \tag{5.19}$$

For values that need two adjacent registers like long or double values, constraints (5.2) have to be extended.

$$\sum_{v \in V} x^{vk}_{ijm} + \sum_{v \in V'} x^{vk-1}_{ijm} \leq 1 \quad \forall i \in B, j \in N_i, m \in M, k \in P \tag{5.20}$$

At the entry point of the program, the x-variables have to be initialized with 0, which is guaranteed by constraints (5.21).

$$x_{100}^{vk} = 0 \quad \forall v \in V, k \in P \tag{5.21}$$

Some values can be rematerialized. More precisely, for certain values that can be easily computed, it is faster to compute them again from scratch, than spilling them to memory and moving them back to the register. To include this in the model, it is necessary to introduce a new set of binary decision variables y_{ij}^{vk}. Let y_{ij}^{vk} be one if value v is rematerialized to position k in block i at label j.

Rematerializations take place directly before a label and therefore constraints (5.8) can be extended.

$$x_{ijm_{\max}}^{vl} \leq x_{ijm_{\max}-1}^{vl} + \sum_{k \in P} z_{ijm_{\max}-1}^{vkl} + y_{ij}^{vl} \quad \forall i \in B, j \in N_i, v \in V, l \in P \tag{5.22}$$

These constraints are sufficient to model the solution space of a program consisting of only one basic block. In the following, an extended version of the model that can handle more than one basic block is considered. An example is shown in Figure 5.2 that has one basic block and two predecessor blocks. Furthermore, two Φ functions are given. For basic block six, the values in the positions are the same in both predecessor blocks in position one and five (see Figure 5.2). The value in position one and five in block six is obviously the same as in the predecessor blocks. If the values are different in the predecessor blocks, they are either defined by Φ-functions, such as in position two, four and six, or the values cannot be propagated to the successor block, such as in position three.

The following constraints ensure the feasible propagation of the value a of position one for all predecessor blocks (block three and four).

$$x_{600}^{a1} \leq x_{3j_{\max}m_{\max}}^{a1} \tag{5.23}$$

$$x_{600}^{a1} \leq x_{4j_{\max}m_{\max}}^{a1} \tag{5.24}$$

The feasible propagation is stated in a general form in constraints (5.25).

$$x_{i00}^{vk} \leq x_{pj_{\max}m_{\max}}^{vk} \quad \forall v \in V, k \in P, i \in B, p \in Pred(i) \tag{5.25}$$

$Pred(i)$ represents the set of all predecessor blocks of block i. Furthermore, it is necessary to link Φ nodes to each other. Φ nodes are not considered as normal nodes. In the example for position two and value e, the following Φ-node linking constraints have to hold for block three and four.

$$x_{600}^{e2} \leq x_{3j_{\max}m_{\max}}^{a2} \tag{5.26}$$

$$x_{600}^{e2} \leq x_{4j_{\max}m_{\max}}^{c2} \tag{5.27}$$

In a general form, constraints (5.28) link Φ nodes to each other

$$x_{i00}^{vk} \leq x_{pj_{\max}m_{\max}}^{uk} \quad \forall k \in Pi \in Bv \in Phi(i), p \in Pred(i), u = Inp(v,p) \tag{5.28}$$

$Phi(i)$ represents the set of all Φ-nodes of block i. For critical backward edges, an empty block needs to be inserted, which consists of only one dummy node. The move iterations of this node are used to change the state of the registers and stack positions. If no moves are performed in this block, it can be removed again. Otherwise it has to be inserted into the IR by the register allocator. If this is considered, the lost copy and swap problem cannot occur. Furthermore, efficient coalescing of Φ-nodes can be done, which includes memory coalescing.

All the relevant notation can be found in Table 5.2.

5.2. Complete Model Formulation

The model can be described as a biobjective mixed integer program. The first objective is the minimization of the total execution time, which is modeled as the total costs of moves and rematerializations inserted by the register allocator, while the second objective is the minimization of the number of stack slots used for spilling. These objectives can be conflicting.

Table 5.2.: Notation for the register allocation MIP

Decision variables	
x_{ijm}^{vk}	binary decision variable indicating if value v is in position k at slot ijm
z_{ijm}^{vkl}	binary decision variable indicating if value v is copied from k to l at slot ijm
y_{ij}^{vk}	binary decision variable indicating if value v is rematerialized to position k at slot ijm
s^k	nonnegative decision variable indicating if stack slot k is used
t	nonnegative decision variable measuring the used stack slot with the highest index
Sets and Parameters	
B	set of basic blocks indexed by i
N_i	set of time slots before a node i indexed by j
P'	set of registers and stack positions
c	artificial cisc spill register
P	set of all positions $P = P' \cup \{c\}$
P^s	set of all stack positions $P^s \subseteq P$
P^0	set of all positions including the dummy position 0 $P^0 = P \cup 0$
M	set of all states between two labels $M = \{0, 1, ..., m_{\max}\}$
M'	set of states between two labels excluding the state directly after a label $M' = M \backslash 0$
M''	set of states M excluding the state directly before the following label $M'' = M \backslash m_{\max}$
$Def(i,j)$	the value that is defined by node ij
$O(i,j)$	the positions that are given in the out regmask of node ij
N_i'	the set of nodes of block i that define a value
F_i	the set of fat projections nodes of block i
N_i''	is the set of nodes requiring the two operand form in block i
$I(v,i,j)$	represents the input regmask of value v of node ij
$Inp(i,j)$	is the set of all the input operands of node ij.
$Inp'(i,j)$	is the input operand that requires the same position as the output operand
$Inp^c(i,j)$	is the input operand that can be used for cisc spilling
$Inp^\Phi(v,p)$	is the input operand of Φ-node v corresponding to block p
V	set of values
V'	is the set of values requiring two adjacent positions $V' \subseteq V$
V^m	set of values that can be rematerialized
$Pred(i)$	is the set of predecessor blocks of block i
$Phi(i)$	is the set of all Φ-nodes of block i
F_i	set of fat projections in block i
R_{ijt}	number of available registers for unused but live values at time slot ijt
L_v	set of time slots in which value v is live
L_{ijt}^1	set of values using one physical register that are live at time slot ijt
L_{ijt}^2	set of values using two physical register that are live at time slot ijt
α_i^{vkl}	costs for moving a value from register k to register l in block i
β_{ij}^{vk}	costs for rematerializing value v to register k in block i at node j
γ_{ij}^{v}	costs for moving value v from a register to the stack at slot ij
δ_{ij}^{v}	costs for moving value v from the stack to a register at slot ij
η_{ij}	costs for using the cisc spill option at node ij
Indices	
i	block index, $i \in B$
j	time slot index, $j \in N_i$
k, l	index for registers and stack slots, $k, l \in R$
v	index for values, $v \in V$
m	move iteration index, $m \in M$

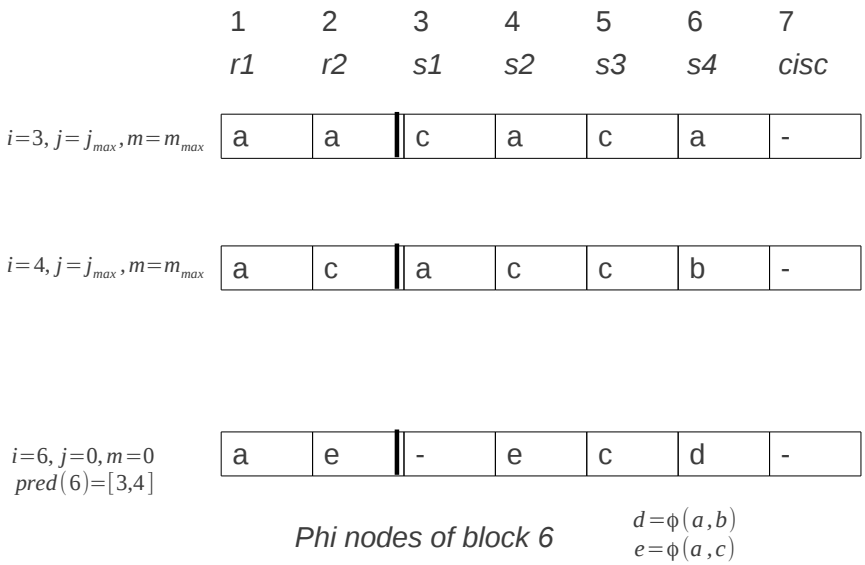


Figure 5.2.: Linking of basic blocks and ϕ-nodes

An optimal solution to the problem that considers only the first objective function, delivers the register allocation with the fastest possible runtime. However, the cost in the objective function is only an estimation of the real cost. The cost is composed of the cost of a single move from position k to position l in block i multiplied by the execution frequency of block i.

The parameter α_i^{vkl} is the estimation of the cost of moving value v from position k to position l in block i, while the parameter β_{ij}^{vk} is the estimation of the cost of rematerialization for value v to position k for node j in block i.

The cost of a single move depends on the positions. A movement between registers is cheaper than a movement between a register and a stack position. The most expensive move is a move from memory to memory. The estimation of the execution frequency is based on the loop depth of the block.

The second objective is the minimization of the number of stack slots used for spilling. This objective optimizes memory usage.

Solution methods for biobjective optimization problems include the determination of the set of pareto optimal solutions, combining the two objectives as weighted sum or the lexicographic ordering of the two objectives. For the theory on multiobjective optimization, see Ehrgott (2005) and Deb (2005). In the model

for optimal register allocation for the Java HotSpot™ Server Compiler, the first objective is usually more important. Therefore, a solution approach that solves the biobjective problem by lexicographic ordering of the two objective functions, is promising. Lexicographic ordering means that first the first objective function is optimized, while the second is neglected. Then, the first objective is fixed by introducing a new constraint. In a minimization problem, the first objective can be set less than or equal to the objective function value of the previous optimization problem. Obviously, the objective function value can not be less than the objective function value of the previous problem, but inequalities are easier to handle in optimization.

In the following, the biobjective mathematical model is described. Let s^k be a set of variables that indicate if stack slot k is used and t measures the stack slot with the highest index.

$$\min \mathbf{F} = (F_1, F_2)$$

subject to:

$$F_1 = \sum_{v \in V} \sum_{k \in P} \sum_{l \in P} \sum_{i \in B} \sum_{j \in N_i} \sum_{m \in M} \alpha_i^{vkl} z_{ijm}^{vkl} + \sum_{v \in V^m} \sum_{k \in P} \sum_{i \in B} \sum_{j \in N_i} \sum_{m \in M} \beta_{ij}^{vk} y_{ij}^{vk} \tag{5.29}$$

$$F_2 = t \tag{5.30}$$

$$\sum_{v \in V} x_{ijm}^{vk} + \sum_{v \in V'} x_{ijm}^{vk-1} \leq 1 \quad \forall i \in B, j \in N_i, m \in M, k \in P^0 \tag{5.31}$$

$$z_{ijm}^{vkl} \leq x_{ijm}^{vk} \quad \forall i \in B, j \in N_i, m \in M'', k, l \in P \tag{5.32}$$

$$\sum_{v \in V} \sum_{k \in P} \sum_{l \in P} z_{ijm}^{vkl} \leq 1 \quad \forall i \in B, j \in N_i, m \in M'' \tag{5.33}$$

$$x_{ijm}^{vl} \leq x_{ijm-1}^{vl} + \sum_{k \in P} z_{ijm-1}^{vkl} \quad \forall i \in B, j \in N_i, m \in M \backslash \{0, m_{\max}\}, v \in V, l \in P \tag{5.34}$$

$$x_{ijm_{\max}}^{vl} \leq x_{ijm_{\max}-1}^{vl} + \sum_{k \in P} z_{ijm_{\max}-1}^{vkl} + y_{ij}^{vl} \quad \forall i \in B, j \in N_i, v \in V^m, l \in P \tag{5.35}$$

$$x_{ijm_{\max}}^{vl} \leq x_{ijm_{\max}-1}^{vl} + \sum_{k \in P} z_{ijm_{\max}-1}^{vkl} \quad \forall i \in B, j \in N_i, v \in V \backslash V^m, l \in P \tag{5.36}$$

$$x_{ij0}^{vk} \leq x_{ij-1m_{\max}}^{vk} \quad \forall i \in B, j \in N_i \backslash \{0\}, v \in V \backslash \{Def(i,j)\}, k \in P \tag{5.37}$$

$$\sum_{k \in O(i,j)} x_{ij0}^{ck} = 1 \quad \forall i \in B, j \in N_i', c = Def(i,j) \tag{5.38}$$

$$\sum_{v \in V} \sum_{k \in O(i,j)} x_{ij0}^{vk} = 0 \quad \forall i \in B, j \in F_i \tag{5.39}$$

$$\sum_{k \in I(v,i,j)} x_{ij_{\max}}^{vk} \geq 1 \quad \forall i \in B, j \in N_i, v \in Inp(i,j) \tag{5.40}$$

$$x_{ij-1m_{\max}}^{uk} \geq x_{ij0}^{vk} \quad \forall i \in B, j \in N_i'', v = Def(i,j), u = Inp'(i,j), k \in P \tag{5.41}$$

$$x_{ijm}^{vc} = 0 \quad \forall v \in V, i \in B, j \in N, m \in M' \tag{5.42}$$

$$x_{100}^{vk} = 0 \quad \forall v \in V, k \in P \tag{5.43}$$

$$x_{i00}^{vk} \leq x_{pj_{\max}m_{\max}}^{vk} \quad \forall v \in V, k \in P, i \in B, p \in Pred(i) \tag{5.44}$$

$$x_{i00}^{vk} \leq x_{pj_{\max}m_{\max}}^{uk} \quad \forall k \in P, i \in B, v \in Phi(i), p \in Pred(i), u = Inp^{\Phi}(v,p) \tag{5.45}$$

$$x_{ijm}^{v0} = 0 \quad \forall i \in B, j \in N_i, m \in M, v \in V \tag{5.46}$$

$$s^k \geq x_{ijm}^{vk} \quad \forall i \in B, j \in N_i, m \in M, v \in V, k \in P^s \tag{5.47}$$

$$t \geq ks^k \quad \forall k \in P^s \tag{5.48}$$

$$x_{ijm}^{vk} \in \{0,1\} \quad \forall v \in V, k \in P, i \in B, j \in N_i, m \in M \tag{5.49}$$

$$z_{ijm}^{vkl} \in \{0,1\} \quad \forall v \in V, k,l \in P, i \in B, j \in N_i, m \in M \tag{5.50}$$

$$y_{ij}^{vk} \in \{0,1\} \quad \forall v \in V, k \in P, i \in B, j \in N_i \tag{5.51}$$

$$s^k \geq 0 \quad \forall k \in P^s \tag{5.52}$$

$$t \geq 0 \tag{5.53}$$

As mentioned above, the objective function minimizes the total cost composed of F_1 and F_2.

Constraints (5.29) represent the first objective function. As mentioned above, the cost of moves and of rematerializations is minimzed. Constraints (5.30) represent the second objective function, which is the minimization of the maximal spill slot register. Constraints (5.31) ensure that only one value can be stored at each position k at a time slot ijm. Moreover, they make sure that values of set V', which are values that need two adjacent registers, are stored in two adjacent registers. Constraints (5.32) guarantee that only feasible moves can be made, i.e.,

it is only possible to move a value from a certain position if it was stored there. Constraints (5.33) make sure that only one move can be performed in each move iteration. Constraints (5.34) show the state transition between two consecutive move iterations. More precisely, a value can only be stored in a position, if it was stored there before or if copied from another position. Constraints (5.35) are the rematerialization constraints. They state that rematerializations can only be done in the last move iteration. Constraints (5.36) state that rematerializations are not allowed for values that cannot be rematerialized. Constraints (5.37) ensure that the state between labels can only be changed by the instruction of a node. Constraints (5.38) guarantee that the value that is defined by a node has to be stored in a position defined by the out regmask. Constraints (5.39) are the fat projection constraints, that say that the values in positions of the corresponding out regmask are killed. Constraints (5.40) ensure that the values of the input operands are in the corresponding in regmask. Constraints (5.41) are the two operand form constraints, that ensure that the two operand form is respected. Constraints (5.42) ensure that the artificial cisc spill register c is only used in the last move operation and can therefore not be used for another purpose than for simulating cisc spilling. Constraints (5.43) make sure that no values exist at the entry point of a program. Constraints (5.44) link basic block i with its predecessor blocks, while constraints (5.45) define that the ϕ-nodes are respected. constraints (5.46) says that the artificial zero position, which is used for modeling long values, is always set to zero. Constraints (5.47) is a linking constraint that measures if stack position k is used. Constraints (5.48) ensure that the index of the stack position with the highest index is assigned to t. Finally, constraints (5.49) to constraints (5.53) define the types of decision variables. More precisely, the variables x_{ijm}^{vk}, z_{ijm}^{vkl} and y_{ij}^{vk} are binary, while the variables s^k and t are nonnegative.

A solution to the model can be easily mapped to real copies and rematerializations which need to be inserted to the intermediate representation. This information can be taken from the z_{ijm}^{vkl} and y_{ij}^{vk} decision variables. The translation out of SSA-form is done in the model, because all the positions of the operands of a Φ-node have to equal and therefore the Φ-node can be simply omitted in the final code generation phase. The model formulation is based

on SSA-form. If the program would not be in SSA-form the model formulation would need additional constraints at every redefinition of a variable. At the point of redefinition it must be ensured that only the new defined value exists and the old values are deallocated.

The model can be improved by reducing indices. For instance the move iterations can be combined to a single step and therefore the index m can be omitted. But as mentioned above, cyclic shifts are forbidden. This can be ensured by constraints (5.54).

$$\sum i \in V \sum_{k \in S} \sum_{l \in S} z_{ij}^{vkl} \leq |S| - 1 \quad \forall v \in V, k \in P, i \in B, j \in N_i \tag{5.54}$$

Liveness analysis is done implicitly in the model. At each node the value of an input node must be live, which is ensured by constraints (5.40) and (5.45). From the point of definition (Constraints (5.38)) to the nodes where the values are used they can only be propagated by the constraints (5.34) and (5.37). Therefore, it is ensured that a value exists on a position for all time slots where it is live and liveness analysis is not necessary. However, exact liveness analysis can be used to strengthen the formulation by forbidding values to be live when they are not needed any more or are not yet defined and strengthen constraints while they are live. Moreover, the number of decision variables could be reduced because the variables x_{ijm}^{vk}, z_{ijm}^{vkl} and y_{ij}^{vk} are only needed while the value v is live.

5.3. Model for Near-Optimal Spilling Decisions for the Java HotSpot™ Server Compiler in SSA-form

In this section a model for deciding when variables should be spilled to the memory is proposed. The model does not take into account rematerializations and memory coalescing for Φ-functions. Furthermore, for a given solution of the program, the registers still have to be assigned. This can be done by algorithms for spill free allocation for programs in SSA-form as described in Pereira and Palsberg (2008) and Hack et al. (2006). The model is similar to the one developed

by Appel and George (2001), but it is extended to the SSA-form. Liveness analyses needs to be performed and the computed data is used in the sets L_v which contain all time slots in which value v is live.

The set of binary variables r^v_{ijt} states if value v is on a register in block i at node j at time t. The time $t \in \{1, 2\}$ says whether the time unit before or after the execution of the instruction is used. More precisely, if $t = 1$, the time unit before the execution of the instruction is used, while if $t = 2$, the unit afterwards is used. Let m^v_{ij} be one if value v is in the memory at node ij, 0 otherwise. The objective is to minimize the cost of load and stores to and from memory and the usage of memory operands. Furthermore, let s^v_{ij} and l^v_{ij} be two sets of binary variables that indicate if store and load operations are performed for value v at node ij. These variables only exist for the lifetime of value v. Therefore, a lifetime analysis has to be performed to obtain the required data for the IP. Finally, c_{ij} are the decision variables indicating if the cisc spill option is used at node ij or not.

$$\min \sum_{v \in V} \sum_{ij \in L_v} (\gamma^v_{ij} l^v_{ij} + \delta^v_{ij} s^v_{ij}) + \sum_{i \in B} \sum_{j \in N_i} \eta_{ij} c_{ij}$$

$$\sum_{v \in L^1_{ij1}} r^v_{ij1} + \sum_{u \in L^2_{ij1}} 2r^u_{ij1} \leq R_{ij1} + w_{ij} c_{ij} \qquad \forall i \in B, j \in N_i \tag{5.55}$$

$$\sum_{v \in L^1_{ij2}} r^v_{ij2} + \sum_{u \in L^2_{ij2}} 2r^u_{ij2} \leq R_{ij2} \qquad \forall i \in B, j \in N_i \tag{5.56}$$

$$r^v_{ij2} = 1 \qquad \forall i \in B, j \in N_i, v = Def(ij) \tag{5.57}$$

$$r^v_{ij1} = 1 \qquad \forall i \in B, j \in N_i, v \in Inp(ij) \backslash Inp^c(ij) \tag{5.58}$$

$$r^v_{ij1} + c_{ij} = 1 \qquad \forall i \in B, j \in N_i, v \in Inp^c(ij) \tag{5.59}$$

$$r^v_{ij2} + m^v_{ij} = 1 \qquad \forall v \in V, ij \in L_v \tag{5.60}$$

$$r^v_{ij2} \leq r^v_{ij1} \qquad \forall v \in V, ij \in L_v \backslash Def(v) \tag{5.61}$$

$$r^v_i j1 \geq r^v_{i(j-1)2} + l^v_{ij-1} \qquad \forall v \in V, ij \in L'_v \tag{5.62}$$

$$l^v_{ij} \geq m^v_{ij} \qquad \forall v \in V, ij \in L_v \tag{5.63}$$

$$m^v_{ij} \geq m^v_{i(j-1)} + s^v_{ij-1} \qquad \forall v \in V, ij \in L'_v \tag{5.64}$$

$$s^v_{ij} \geq r^v_{ij2} \qquad \forall v \in V, ij \in L_v \tag{5.65}$$

$$r^v_{i01} \leq r^v_{lj_{max}2} \qquad \forall i \in B, v \in L_{i0}, l \in Pred(i) \tag{5.66}$$

$$m^v_{i0} \leq m^v_{lj_{max}} \qquad \forall i \in B, v \in L_{i0}, l \in Pred(i) \tag{5.67}$$

$$r^v_{i01} \leq r^u_{lj_{max}2} \qquad \forall i \in B, v \in Phi(i), l \in Pred(i), u = Inp^{\Phi}(v,p) \tag{5.68}$$

$$m^v_{i0} \leq m^u_{lj_{max}} \qquad \forall i \in B, v \in Phi(i), l \in Pred(i), u = Inp^{\Phi}(v,p) \tag{5.69}$$

$$r^v_{ijt}, m^v_{ij} s^v_{ij}, l^v_{ij}, c_{ij} \in 0,1 \qquad \forall v \in V, ij \in L_v, t \in \{1,2\} \tag{5.70}$$

The objective function is to minimize the total costs for loads and stores to and from memory and the costs for using the cisc spill option. γ^v_{ij} represents the costs for the loads and δ^v_{ij} the costs for stores. The cost for using the cisc spill option is given by η_{ij}. The set L_v defines the liveness of variable v. Constraints (5.55) ensure that the maximum number of available registers is not exceeded before the node is executed, while constraints (5.56) ensure the same after the node is

executed. If the cisc spill option is used w_{ij} additional registers are available. w_{ij} is either 2 or 1 depending on either a long or a normal value is used as the memory operand. R_{ijt} states the number of available registers. It is computed as the number of registers minus the number of registers used as input or output operands or for a fat projection. Constraints (5.57) make sure that the value that is defined at node ij is in a register afterwards. Constraints (5.58) guarantee that all input values of an instruction are on a register, except for the one with cisc spill option, which is handeled in constraints (5.59). This operand can either be on a register or the cisc spill option is chosen. Constraints (5.60) state that a value that is live must be either allocated to a register or to memory. Constraints (5.61) state that a value can only be on a register after an instruction when it was on a register before the instruction, except for the definiton of the value. Constraints (5.62) ensure that a value can be on a register before an instruction, if it has been on a register after the previous instruction or it is loaded from memory. Constraints (5.63) say that a value can only be loaded from memory if it was there before. Constraints (5.64) state that a value can only be in the memory if it was in the memory one instruction before or it was stored to the memory. Constraints (5.65) guarantee that a value can only be stored from a register to the memory if it was actually on the register. Constraints (5.66) make sure that a value can be at the begining of a basic block if it has been in a register at the end of the blocks of all predecessors and constraints (5.67) ensure the same for the memory. Constraints (5.68) state that a value that gets defined by a Φ-node can only be in the register if all input values are in a register at the end of their corresponding block and constraints (5.69) state the same for the memory. Finally, constraints (5.70) ensure that the decision variables are binary.

6. Conclusions

Registers are used to store local variables and temporary values. Compared to the memory, registers can be accessed much faster. However, the number of registers is limited. Therefore, the problem of register allocation is to decide which values to store on the fast accessible registers and which to spill to the memory.

In this thesis a mathematical programming formulation for the register allocation problem was proposed. To the best of our knowledge, this is the first time that a mathematical programming formulation was presented for the problem of SSA-form based register allocation. The model is capable of dealing with all the irregularities of the IA-32 architecture. Another model was proposed that solves the subproblem of spilling variables to memory in a register allocator for programs in SSA-form.

Furthermore, different modeling approaches for register allocation were presented and advantages and disadvantages of the different approaches were discussed. Register allocation can be modeled as a graph coloring problem, as a multi commodity network flow problem, as a partitioned boolean quadratic optimization problem and can be solved by the linear scan algorithm with second chance bin packing.

An analysis of the intermediate representation of the server compiler was provided. The focus was on intermediate representation in SSA-form. Finally, a simple feasible register allocator was implemented.

Future work can focus on improving the implementation so that it is competitive to the existing allocator of the Java Hot Spot™ Server Compiler.

A. Abbreviations

Abbreviation	Description
CISC	complex instruction set computer
CPU	central processing unit
CSSA	conventional static single assignment form
FPU	floating point unit
GC	garbage collector
GCC	gnu C compiler
HIR	high level intermediate representation
IP	integer program
ILP	integer linear program
IR	intermediate representation
JIT	just in time
JVM	Java virtual machine
LIR	low level intermediate representation
MCNF	multi-commodity network flow
MIP	mixed integer program
MMX	multi media extension
ORA	optimal register allocator
PBQP	partitioned boolean quadratic optimization problem
RISC	reduced instruction set computer
SIMD	single instruction multiple data
SSA	static single assignment
SSE	streaming SIMD extension
SSI	static single information
VM	virtual machine

List of Figures

List of Tables

Bibliography

Ananian, C. (1999). The Static Single Information Form. Master's thesis, Massachusetts Institute of Technology.

Appel, A. W. and George, L. (2001). Optimal spilling for CISC machines with few registers. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 243–253, New York, NY, USA. ACM.

Briggs, P. (1992). *Register allocation via graph coloring*. PhD thesis, Rice University.

Briggs, P., Cooper, K., Harvey, T., Simpson, L., et al. (1998). Practical improvements to the construction and destruction of static single assignment form. *Software-Practice and experience*, 28(8):859–882.

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

Chaitin, G. (1982). Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–105. ACM.

Click, C. and Paleczny, M. (1995). A simple graph-based intermediate representation. In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, IR '95, pages 35–49, New York, NY, USA. ACM.

Cytron, R., Ferrante, J., Rosen, B., Wegman, M., and Zadeck, F. (1991). Efficiently computing static single assignment form and the control depen-

dence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490.

Deb, K. (2005). Multi-objective optimization. In Burke, E. K. and Kendall, G., editors, *Search Methodologies*, pages 273–316. Springer US.

Ehrgott, M. (2005). *Multicriteria optimization*, volume 491. Springer Verlag.

Farach, M. and Liberatore, V. (1998). On local register allocation. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, SODA '98, pages 564–573, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Fu, C., Wilken, K., and Goodwin, D. (2005). A faster optimal register allocator. *The Journal of Instruction-Level Parallelism*, 7:1–31.

Goodwin, D. and Wilken, K. (1996). Optimal and near-optimal global register allocation using 0–1 integer programming. *Software: Practice and Experience*, 26(8):929–965.

Hack, S. (2004). *Register allocation for programms in SSA-form.* PhD thesis, Universität Karlsruhe.

Hack, S. and Goos, G. (2006). Optimal register allocation for SSA-form programs in polynomial time. *Information Processing Letters*, 98(4):150 – 155.

Hack, S., Grund, D., and Goos, G. (2006). Register allocation for programs in SSA-form. In *Compiler Construction*, pages 247–262. Springer.

Koes, D. and Goldstein, S. C. (2005). A progressive register allocator for irregular architectures. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 269–280, Washington, DC, USA. IEEE Computer Society.

Koes, D. R. and Goldstein, S. C. (2006). A global progressive register allocator. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 204–215, New York, NY, USA. ACM.

Kong, T. and Wilken, K. D. (1998). Precise register allocation for irregular architectures. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, MICRO 31, pages 297–307, Los Alamitos, CA, USA. IEEE Computer Society Press.

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

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

Nemhauser, G. and Wolsey, L. (1999). *Integer and combinatorial optimization*. Wiley New York.

Paleczny, M., Vick, C., and Click, C. (2001). The Java HotSpot Server Compiler. Technical report, Sun Microsystems.

Pereira, F. M. Q. (2008). A survey on register allocation. Technical report, University of California Los Angeles.

Pereira, F. M. Q. and Palsberg, J. (2008). Register allocation by puzzle solving. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 216–226, New York, NY, USA. ACM.

Pereira, F. M. Q. and Palsberg, J. (2009). SSA Elimination after Register Allocation. In *Proceedings of the 18th International Conference on Compiler Construction*, CC '09, pages 158–173, Berlin, Heidelberg. Springer-Verlag.

Poletto, M., Engler, D. R., and Kaashoek, M. F. (1997). 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*, PLDI '97, pages 109–121, New York, NY, USA. ACM.

Poletto, M. and Sarkar, V. (1999). Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21:895–913.

Scholz, B. and Eckstein, E. (2002). Register allocation for irregular architectures. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*, LCTES/SCOPES '02, pages 139–148, New York, NY, USA. ACM.

Sethi, R. (1975). Complete register allocation problems. *SIAM Journal on Computing*, 4(3):226–248.

Sreedhar, V., Ju, R., Gillies, D., and Santhanam, V. (1999). Translating out of static single assignment form. In *Static analysis: 6th international symposium, SAS'99, Venice, Italy, September 22-24, 1999: proceedings*, volume 4, page 194. Springer Verlag.

Standard Performance Evaluation Corporation (2008). SPECjvm2008.

Sun Microsystems, I. (2001). The Java HotSpot Virtual Machine. Technical White Paper.

Tomlin, J. (1966). Minimum-cost multicommodity network flows. *Operations Research*, 14(1):45–51.

Traub, O., Holloway, G., and Smith, M. D. (1998). Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 142–151, New York, NY, USA. ACM.

Wimmer, C. (2004). Linear scan register allocation for the Java HotSpot client compiler. Master's thesis, Johannes Kepler University Linz.

Wimmer, C. and Franz, M. (2010). Linear scan register allocation on SSA form. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 170–179, New York, NY, USA. ACM.