JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
**DI Josef Eisl, BSc.**

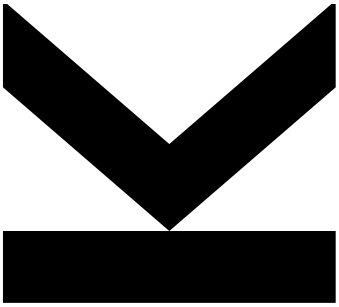
Submitted at
**Institute for System Software**

Supervisor and
First Examiner
**o.Univ.-Prof. DI Dr.Dr.h.c. Hanspeter Mössenböck**

Second Examiner
**Ao.Univ.-Prof. DI Dr. Andreas Krall**

October 2018

# Trace Register Allocation



Doctoral Thesis
to obtain the academic degree of
Doktor der technischen Wissenschaften
in the Doctoral Program
Technische Wissenschaften

**JOHANNES KEPLER**
**UNIVERSITY LINZ**
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

In memory of my brother.

(Wolfgang Eisl, 1973–2016)

# Abstract

Register allocation, i.e., mapping the variables of a programming language to the physical registers of a processor, is a mandatory task for almost every compiler and consumes a significant portion of the compile time. In a just-in-time compiler, compile time is a particular issue because compilation happens during program execution and contributes to the overall application run time. Compilers often use global register allocation approaches, such as graph coloring or linear scan, which only have limited potential for improving compile time since they process a whole method at once. With growing methods sizes, these approaches are reaching their scalability boundary due to their limited flexibility.

We developed a novel *trace register allocation* framework, which competes with global approaches in both, compile time and code quality. Instead of processing a whole method at once, our allocator processes linear code segments (traces) independently and is therefore able to (1) select different allocation strategies based on the characteristics of a trace in order to control the trade-off between compile time and peak performance, and (2) to allocate traces in parallel in order to reduce compilation latency, i.e., the time until the result of a compilation is available.

We implemented our approach in GraalVM, a production-quality Java Virtual Machine developed by Oracle. Our experiments verify that, although our allocator works on a non-global scope, it performs equally well (or even better) than a state-of-the-art global allocator. This result is already remarkable, since it refutes the common believe that global register allocation is a necessity for good allocation quality. Furthermore, to demonstrate the flexibility of trace register allocation, we seamlessly reduce register allocation time in a range from 0 to 40%, depending on how much peak performance penalty we are willing to sacrifice (from 0–12% on average). As an extra benefit, we present parallel trace register allocation, a mode where traces are allocated concurrently by multiple threads. In our experiments, this reduces the register allocation latency by up to 30% when using 4 threads compared to a single allocation thread.

Our work adds a new class of register allocators to the list of state-of-the-art approaches. Its flexibility opens manifold opportunities for future research and optimization, and enriches the landscape of compiler construction.

# Kurzfassung

Eine der wesentlichen Aufgaben eines Compilers ist es die potentiell unbegrenzte Anzahl von Variablen eines Programms auf die physisch vorhandenen Maschinenregister eines Prozessors abzubilden. Dieser Vorgang – die sogenannte Registerallokation – trägt erheblich zur Übersetzungszeit eines Programms bei. Die Optimierung solcher Registerallokatoren spielt besonders für dynamische Übersetzer (*Just-In-Time Compiler*), in denen die Übersetzung erst zur Laufzeit stattfindet, eine wichtige Rolle, da die Allokationszeit hier maßgeblich zur Gesamtlaufzeit des Programms beiträgt. Viele Übersetzer verwenden globale Registerallokatoren, zum Beispiel nach dem *Graph Coloring* oder *Linear Scan* Verfahren, in denen das Optimierungspotential aufgrund des methoden-basierten Ansatzes eingeschränkt ist. Mit zunehmender Methodengröße erreichen diese bestehenden Ansätze die Grenzen ihrer Skalierbarkeit.

Um dieses Problem zu lösen, haben wir im Zuge dieser Arbeit einen neuartigen *Trace-basierten Registerallokator* entwickelt, der eine Methode nicht als Ganzes verarbeitet, sondern sie in lineare Codesegmente (*Traces*) zerlegt, die dann unabhängig voneinander bearbeitet werden. Zum einen können so abhängig von den Eigenschaften der Codesegmente unterschiedliche Strategien angewendet werden, die uns beispielsweise erlauben, für kritische Teile mehr Zeit aufzuwenden als für weniger kritische Teile. Zum anderen können die einzelnen Segmente parallel verarbeitet werden, was die Latenzzeit einer Übersetzung verringert.

Die Implementierung unseres Registerallokators erfolgte in GraalVM, einer von Oracle entwickelten hochoptimierenden virtuellen Maschine für Java. Unsere Experimente haben gezeigt, dass unser lokaler Ansatz mindestens gleich gut – und in einigen Fällen sogar besser – als globale *State-of-the-Art*-Allokatoren arbeitet. Diese Erkenntnis widerlegt somit die weit verbreitete Annahme, dass für gute Ergebnisse globale Allokationsverfahren benötigt werden. Aufgrund der Flexibilität unseres Ansatzes konnte, abhängig von den tolerierten Performance-Einbußen (durchschnittlich zwischen 0–12%), die Registerallokationszeit um bis zu 40% reduziert werden. Zusätzlich erlaubt unser Allokator eine parallele Abarbeitung, die beispielsweise bei der Verwendung von vier Threads anstelle von einem, die Allokationslatenz um bis zu 30% verringert.

Unser hier vorgestellter Registerallokator fügt sich nahtlos in die Liste der State-of-the-Art-Ansätze ein. Die zusätzliche Flexibilität eröffnet eine Vielzahl an zukünftigen Forschungsmöglichkeiten und bereichert die bestehende Übersetzerbau-Landschaft.

# Acknowledgement

First, I thank my advisor Hanspeter Mössenböck for his continuous support and for encouraging me to pursue my ideas. Thanks for numerous discussions and thorough feedback on all of my work, even if the schedule was sometimes tight due to my commitment to the *just-in-time* principle. Although we might never agree on the usage of the terms *variable*, *value* and *temporary*, I learned a lot, especially about the importance of presenting complex topics in a simple and approachable way.

I am very grateful to Oracle Labs for funding my position and providing me with a productive environment for exploring my vision. Special thanks to Thomas Würthinger without whom this work would not exist. I thank Doug Simon for being an amazing manager and for making working within the Graal compiler team a delightful experience. Also, for finding every single of my JavaDoc typos. Thanks to the other members of the team, especially to Roland Schatz, Gilles Duboscq, Tom Rodriguez, Lukas Stadler, and Stefan Anzinger, who received me with open arms when I joined the ride. Your reviews and suggestions made my code do something useful. Thanks also to Christian Wimmer for enlightening discussions about register allocation and for the *C1Visualizer*.

I want to thank two former colleagues from the Institute for System Software, Matthias Grimmer and Stefan Marr, who taught me a lot about paper writing and academia in general. Ask Stefan about the $3 \times 3$ rule for a successful conference poster if you happen to run into him. Definitely worked for me.

Special shout-out to my office buddies (and fellow ACM Student Research Award Winners!) David Leopoldseder and Manuel Rigger. Although the atmosphere in the office was occasionally a bit heated—partially due to an undersized air condition, partially because of me trying to finish my thesis on time—the overall experience was very joyful and fun. All the best for your upcoming endeavors.

I am grateful to my friends for their continuous friendship, even when the intervals between our get-togethers are often longer than I want them to be. Special thanks to Bernhard Urban, who convinced me to come to Linz to join the Graal team. See you all soon.

I thank my family for their everlasting support and for always believing in me. Especially, I thank my parents for supporting all of my wild ideas. It seems like some of them really worked out.

Most importantly, I want to thank my wonderful wife Marianne, who not only moved with me to Linz for my studies, but also supported me in every possible way over the last years. Thank you for encouraging me to continue when I was in doubt and for giving me critical feedback when I needed it. And thank you for every once in a while reminding me that there is more to life than compilers and virtual machines. It was utterly important for keeping my mental health. I love you!

Pursuing a Ph.D. is not a one-person show. In fact, it involves a lot of people, institutions and lucky coincidences. In honor of those who were making this work possible, I include them in every “we” inhere.

# Contents

# Chapter 1

# Introduction

## Teaser

Most optimizing compilers use *global* register allocation approaches, such as *graph coloring* or *linear scan*, which process a whole method at once. Compiler optimizations such as *inlining* or *code duplication* cause methods to become large. This poses two problems. First, register allocation time increases with method complexity, often in a non-linear fashion [Poletto and Sarkar, 1999]. Second, different regions contribute differently to the overall performance of the compiled code [Bala et al., 2000]. Global allocators do not adjust the allocation strategy based on the expected execution frequency of the code region. *Important* and *unimportant* parts of a method are allocated with the same algorithm, i.e., they contribute equally to the overall allocation time.

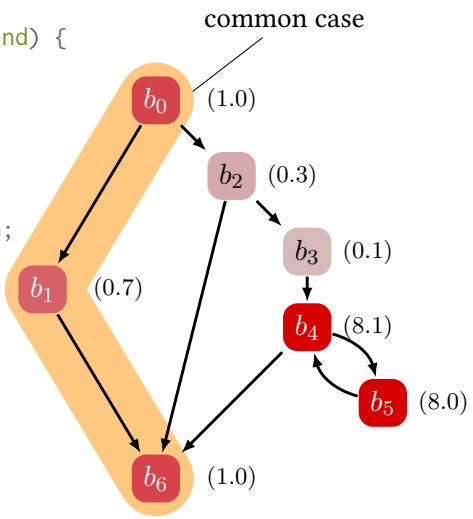
Let us have a look at the `allocateArray()` example in Figure 1.1. A global allocator spends about the same amount of time for the *uncommon cases* ($b_2$–$b_5$) as for the *common case* ($b_0$, $b_1$ and $b_6$). However, we want to spend our time budget more wisely, e.g., by spending $80\%$ on the most likely case, and only $20\%$ for the rest.

In addition to *compile time*, i.e., the time required to compile a method, *compilation latency*, i.e., the duration until the result of a compilation is ready, is an important metric, especially for just-in-time compilers. In contrast to the former, *latency* can be tackled with *parallelization*, in case multiple threads are available to the compiler. Because global register allocators process a whole method at once, they offer only few opportunities to do work concurrently.

We propose a novel *non-global* register allocation approach, called *trace register allocation*, that tackles both issues of traditionally global allocators, namely to control the *compile time* on a fine-granular level, and to reduce *compilation latency* using parallelization.

```
long *allocateArray(int length, boolean fillContents) {
        long *result;
/*B0*/  if (TLAB.top + length + 1 <= TLAB.end) {
/*B1*/      result = TLAB.top;
            TLAB.top += length + 1;
        } else {
/*B2*/      result = heapAlloc(length + 1);
            if (fillContents) {
/*B3*/          long *ptr = result + length;
/*B4*/          while (result < ptr) {
/*B5*/              *ptr = 0;
                    ptr--;
                }
            }
        }
/*B6*/  result[0] = length;
        return result;
}
```

(a) C-like Pseudo Source Code




(b) Control-flow Graph

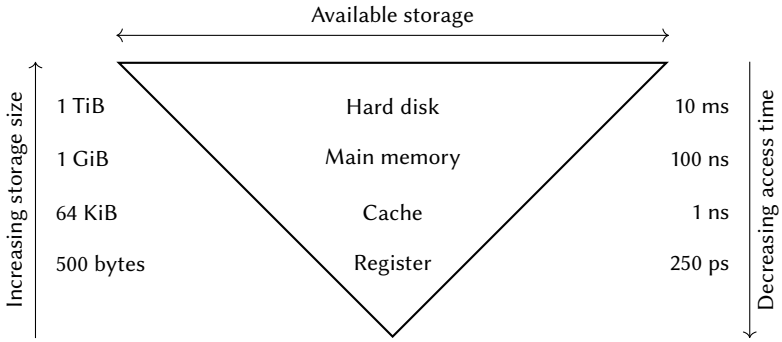
The method `allocateArray()` models an *array allocation* snippet. It returns a pointer to memory of size `lenght + 1` where the first cell is the array `length`. The `fillContents` parameter specifies whether a zero-initialized array is required. There are two allocation scenarios. The array is either allocated in a *thread-local allocation buffer* (TLAB, block $b_1$) or on the heap ($b_2$–$b_5$) if the TLAB is too small. In addition, we assume that TLAB space is *zero-initialized* whereas the heap contains garbage. While TLAB allocation is just a pointer increment, `heapAlloc` might require some time to free up space. The profiled execution frequencies (parentheses) suggest that TLAB allocation usually succeeds. If it does not and we require clean memory, we spend a lot of time in the initialize loop ($b_4$ and $b_5$). The snippet is inspired by the one used in Graal (Chapter 3), although details were changed for presentation purposes.

**Figure 1.1:** The `allocateArray()` example code snippet

## 1.1 Background on Register Allocation

In most programming languages, we can use an unlimited number of *variables*. They allow us to refer to a *value* via a *symbolic name* without knowing its exact storage *location* on the underlying execution platform. Variables allow us to *name* information and therefore to express our intent. Using variables and naming them in a meaningful way, increases the readability of programs and makes them easier to maintain. When a program is executed on a physical machine—either on an interpreter, a compiler, or a combination of both—we need to find a mapping from variables to *physical* storage locations.

Nowadays, there are a variety of machines with different storage models. We focus on architectures that are most common today. The memory hierarchy of a typical computer is depicted in Figure 1.2. These architectures basically provide two classes of storage which are directly accessible to a programmer: *registers* and *memory*.




The pyramid representation is inspired by Pereira [2008, Figure 2.1]. The numbers are taken from Hennessy and Patterson [2003, Figure 5.1].

**Figure 1.2:** Typical memory hierarchy of a computer

*Registers* are located in the processor and are the fastest type of storage. Depending on the processor, their size is usually in the range of 16 to 512-bits. Their number is limited to support efficient encoding. For example, the AMD64 architecture features only 16 general purpose registers[1] in 64-bit mode. A SPARC processor provides a register set of 32 registers[2] for general usage.

With *memory* we refer to non-persistent, random-access storage. The number of storage locations available in memory is significantly higher than the number of registers. For the course of this work, it is safe to assume that it is *unlimited*. Regarding access time, memory is orders of magnitudes slower than registers [Hennessy and Patterson, 2003]. To speed up memory access, most computers use one or multiple levels of caches. A cache basically mirrors parts of recently used data from the main memory for faster consecutive access. Caches are transparent to the programmer.[3] A downside of memory is that processors only have limited support for accessing it directly. Many machine instructions require their operands to reside in registers.

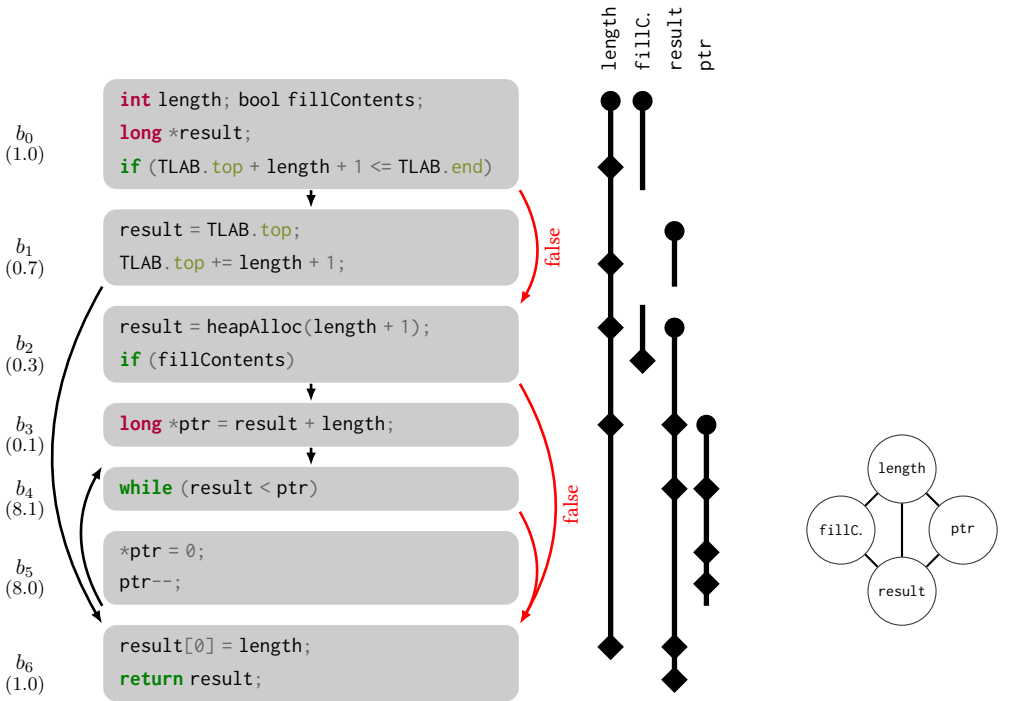
In this work, we will focus mainly on *method-based compilers*. Method-based compilers translate a method (function) from a source language to a target language, in our case to machine code. We use a control-flow graph (CFG) to represent methods. The nodes of this graph are *basic blocks*, i.e., a sequence on *branch-free* instructions. Figure 1.1 shows a method in C-like pseudo code and its corresponding control-flow graph.

Given the options above, the simplest solution is to map every variable to exactly one memory location. Since memory is virtually unlimited, finding this mapping is trivial. Whenever a value needs to reside in a register we can load it from memory into a register and store the result back

[1] *Intel® 64 and IA-32 Architectures Software Developer's Manual — Volume 1 Basic Architecture* by Intel [2013a] Section 3.4.1.1

[2] *Oracle SPARC Architecture 2011* by Oracle Corporation [2012] Section 5.2

[3] However, many processors provide instructions to manipulate the cache, for example for loading specific memory locations into the cache or flushing it.




(a) Lifetime intervals (b) Interference graph

The numbers in parentheses on the left are execution frequencies. The *lifetime intervals* for the `allocateArray()` method are shown on the left. The bullet symbol (●) marks the definition (*write*) of the value, a square (◆) depicts a usage (*read*). The *interference graph* is shown on the right. Vertices represent variables, edges *interference* between them. See also Figure 1.1 for more context.

**Figure 1.3:** Lifetime intervals and interference graph for `allocateArray()`

to memory. However, repeatedly moving values back and forth between registers and memory is not efficient. Therefore, optimizing compilers try to keep values in registers whenever possible. This optimization is known as *register allocation*. For programs where there are more variables than registers the decision whether to keep a value in a register is non-trivial. In this case, multiple variables need to share a register. To do so, we need to find variables that are not *live* at same time, i.e., they do not *interfere*.

**Example** Let us have a look at our recurring example in Figure 1.1. In total, there are 4 variables, `length`, `fillContents`, `result`, and `ptr`. Figure 1.3a depicts the lifetime intervals of these variables. The interference relation can be visualized as a graph, the so-called *interference graph (see Section 2.3)*. The vertices of the graph represent the variables. An edge between two vertices means that the variables are *live* at the same time. Figure 1.3b shows the interference graph for our example. While `length` and `result` *interfere* with all other variables, the *lifetime intervals* for `fillContents` and `ptr` do not overlap. Therefore, they can use the same register.

This leads to the basic definition of the register allocation problem:

**The Register Allocation Problem**

Allocate a *finite* number of machine *registers* to an *unbounded* number of *variables* such that variables with interfering live ranges are assigned to different registers.

It is not always possible to find a solution for this problem. For example, in cases where the number of registers is smaller than the number of live values at any instruction. In that case we need to store one or more variables in memory, usually on the *stack* of the method. There are two concepts that are closely related to register allocation. *Live Range Splitting:* Given the live ranges of a variable, split them in such a way that the variable can reside in different locations (register or stack) at different points of the method. *Spilling:* Move a variable to the stack, so that a register becomes free and can be used for some other variable. This brings us to the problem that is by far more interesting—and also more complicated—than the basic register allocation problem:

**The Splitting and Spilling Problem**

In case the register pressure (the number of live variables) is higher than the number of available registers, decide which variables to *split* and/or *spill* to reduce the register pressure.

*Splitting and spilling* is often formulated as an optimization problem, i.e., to make decisions to minimize a *cost function*. The cost function might be based on a *static* property such as the number of *added spill moves*, a *dynamic* property such as the number of *executed spill moves*, or something more abstract such as *system performance*. It is often the structure of the cost function that causes the problem to get difficult.

There are different approaches to register allocation. Most register allocators in a method-based compiler can be categorized into *local* and *global* approaches, based on the scope they work on. *Local* register allocators work on the scope of a single basic block. Due to their limited scope they are fast and simple to implement but have limited optimization potential. On the other hand, *global* allocators process the whole method at once. While this offers opportunities for optimization it also makes the problem more difficult to solve. Most optimizing compiler use an allocator that follows the global principle. Two kinds of *global* register allocation are commonly used today. The first one is register allocation based on *graph coloring*, which was proposed by Chaitin et al. [1981]. The other approach is *linear scan* register allocation, first described by Poletto and Sarkar [1999].

## 1.2 Graph Coloring

Graph coloring is used in many compilers today, for example in WebKit,[4] the HotSpot server compiler,[5] or in GCC.[6] The idea is to solve the register allocation problem via *coloring* [Aho et al., 1974, Chapter 10.4] of the *interference graph*. Each color represents a machine register. Since *graph coloring* with more than two colors is NP-complete [Karp, 1972], it is intractable to find the optimal solution in polynomial time. Therefore, Chaitin et al. [1981] and others [Briggs et al., 1994; Chow and Hennessy, 1990] proposed heuristics to approximate solutions in polynomial time.

### 1.2.1 Chaitin's Allocator

Chaitin et al. [1981] first showed a practical application of graph coloring for register allocation. Figure 1.4 shows an overview of the original Chaitin allocator. It operates in seven phases. *Renumber* calculates lifetime intervals for each variable definition. *Build* constructs the interference graph. *Coalesce* combines live ranges which are connected via a move instruction and do not interfere. *Spill costs* computes the estimated cost of spilling a variable based on its *definition* and *use* positions and the expected execution frequency of its live ranges. *Simplify* removes all nodes from the graph where the number of neighbors (*degree*) is smaller than the number of available registers ($k$) until the graph is empty; the removed nodes are put onto a stack. Then, the removed nodes are added back to the graph in reverse order. Every re-added node is guaranteed to have less than $k$ neighbors, so it can be colored. If during node removal we get to a point where there are only nodes with $k$ or more neighbors, one is selected for spilling based on its spill costs. In the worst case all neighbors have a different color. However, there is still at least one color left for the current node. From the nodes with higher degree, one is selected for spilling based on its spill costs. *Spill code* inserts spill and reload code for spilled nodes. *Select* assigns colors to the nodes in the reverse order they were removed from the graph by *simplify*. Each node gets a color different to all its neighbors. Note the feedback loop from *simplify* to *renumber* via *spill code*. Whenever *simplify* decides to spill a node the algorithm is restarted. This is one of the main sources of time complexity of this approach.

[4] *Graph Coloring Register Allocator in WebKit* by WebKit [2017a]
[5] *Chaitin Allocator in C2* by OpenJDK [2017a]
[6] *Integrated Register Allocator in GCC* by GCC [2017]

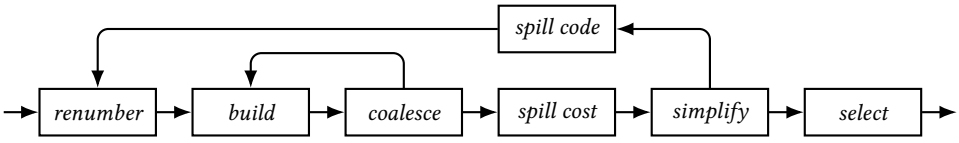
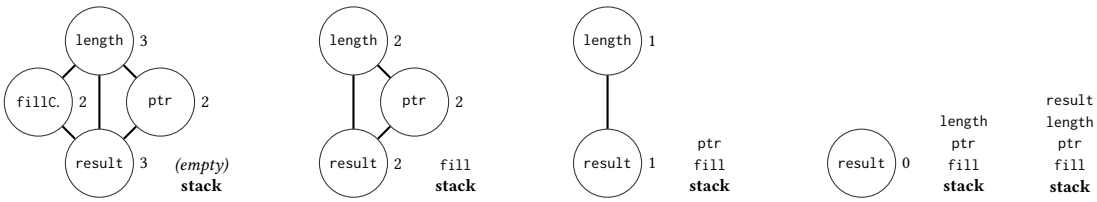


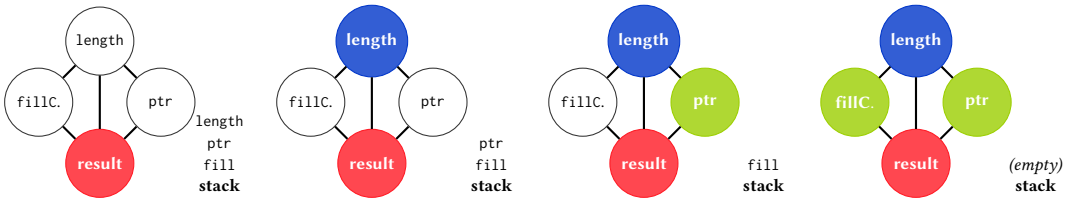

Figure adapted from [Briggs et al., 1994, Figure 1].

**Figure 1.4:** Chaitin's allocator




The *simplify* phase removes nodes with a *degree* (right of the nodes) less than 3 and pushes them onto the stack. This decreases the degree of the neighbors. The procedure is continued until there are no more nodes, or all nodes have a degree greater than 2.

(a) Simplify phase



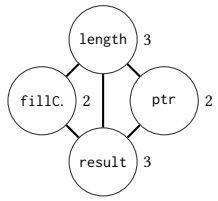

The *selection* phase assigns colors (registers) to the nodes. The nodes are processed in the reverse order they where removed from the graph in the *simplify* phase. In the example there are *three* registers available, $reg_1$ 🔴, $reg_2$ 🔵, and $reg_3$ 🟢.

(b) Select phase

**Figure 1.5:** Graph coloring of `allocArray` using three registers

### Example without Spilling

To demonstrate the approach, let us refer to our example in Figure 1.1. Let us assume that we have *three* registers available, $reg_1$ 🔴, $reg_2$ 🔵, and $reg_3$ 🟢. First, the algorithm builds an interference graph (Figure 1.5a). Next, we *simplify* the graph by removing nodes with a degree of less than $k = 3$. The removed nodes are pushed onto the stack. Once all nodes have been removed from the graph, we can continue with the *select* phase. We pop a node from the stack and assign it a color that is different from the colors of its neighbors. Due to the construction of the stack in the *simplify* phase this is always possible (see Figure 1.5b).




(a) Interference graph

| Name | def/use frequencies | $cost\ (\sum)$ | $degree$ | $cost/degree$ |
|---|---|---|---|---|
| `length` | $1.0+1.0+0.7+0.3+0.1+1.0$ | 4.1 | 3 | 1.4 |
| `fillC.` | $1.0+0.3$ | 1.3 | 2 | 0.6 |
| `result` | $0.7+0.3+0.1+8.1+1.0+1.0$ | 11.2 | 3 | 3.7 |
| `ptr` | $0.1+8.1+8.0+8.0$ | 24.2 | 2 | 12.1 |

(b) Spill costs for `allocateArray()`

The spill costs are the sum of definitions and usages weighted by their estimated execution count [Chaitin, 1982]. The node with the lowest cost divided by its degree is selected for spilling, in this case `fillContents`. Execution frequencies for the blocks of `allocArray` are given in Figure 1.7a.

**Figure 1.6:** Graph coloring of `allocArray` using two registers (interference before spilling)

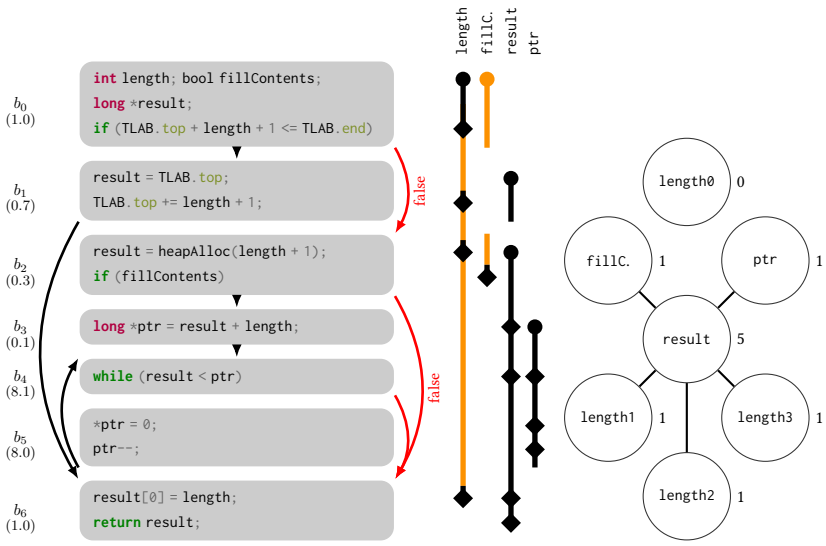
#### Example with Spilling

Let us redo the example with only *two* registers, $reg_1$ 🔴 and $reg_2$ 🔵. As can be seen in Figure 1.6a, there is no node with a degree lower than 2. Therefore, we need to select a node for spilling. To do so, we calculate the spill costs. The result is shown in Figure 1.6b. We first select `fillContents` for spilling because it has the lowest *spill cost*. We therefore split the interval and reload it at every usage of the variable. However, this does not change the degrees in the interference graph. We continue with the next best candidate, `length`. This time the register pressure reduces as shown in Figure 1.7. We can simplify the graph and push the variables onto the stack (Figure 1.8). Finally, the *select* phase colors the nodes by popping them from the stack and assigning colors, as shown in Figure 1.9.

### 1.2.2 Other Graph Coloring Approaches

Following the work of Chaitin et al., many refinements have been proposed to improve the result of the original approach. While the underlying idea was retained, the details of the spilling heuristics or the interaction of the various phases changed.

Chaitin's allocator is rather pessimistic regarding spilling. It spills whenever it cannot prove that there is a valid coloring, i.e., that every node has a degree lower than the number of registers. However, in many cases it is possible to color the graph also in such situations. See Figure 1.10 for an example. Although all nodes have a degree of 2, the graph is 2-*colorable*. To overcome this issue, Briggs et al. [1994] proposed the *optimistic* graph coloring allocator. As shown in Figure 1.11, the main difference to the original algorithm is that instead of eagerly spilling nodes, Briggs et al. try to select a color also for *potentially spilled* nodes. Only when this fails, a node




(a) Lifetime Intervals (b) Interference Graph

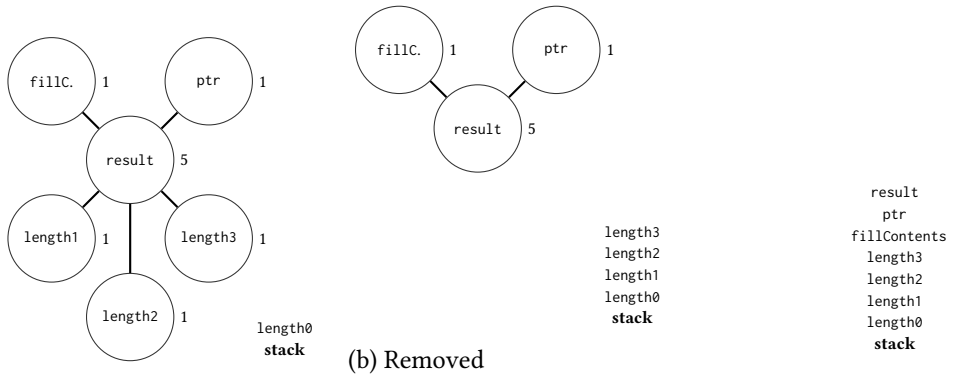
Spilled interval in orange (●).

**Figure 1.7:** Graph coloring of `allocArray` using two registers (interference after spilling)

is *actually spilled*. Another difference to Chaitin et al. is their *conservative coalescing* approach. Instead of combining all nodes which are connected with a move instruction, only those are considered that will provably not introduce spilling.

Building on the results of Chaitin et al. and Briggs et al., George and Appel [1996] proposed *iterative register coalescing*. As the name suggests, they *iteratively* perform coalescing in a loop with the simplification phase. This way, they are able to coalesce more nodes but still guarantee that they do not introduce spilling. Investigations by Pereira and Palsberg [2005] suggest that George and Appel's approach is among the best performing polynomial-time register allocators with respect to register allocation quality.

**Time Complexity** Giving a definite asymptotic time complexity measure for graph coloring allocators is difficult as it strongly depends on the concrete implementation, e.g., optimistic vs. pessimistic coloring or conservative vs. iterative coalescing. Building the interference graph repeatedly whenever spill code is inserted is considered the most expensive part of the approach [Cooper and Torczon, 2011, Chapter 13]. Also, the size of the graph can gets quadratic in the number of live ranges [Poletto and Sarkar, 1999]. Experiments conducted by Briggs [1992] suggest that in practice the allocator behaves quadratic in terms of its "input size."




(a) Removed `length0` (b) Removed `length1`–`length3` (c) Empty Graph

**Figure 1.8:** Graph coloring of `allocArray` using two colors (simplify)

Compile time is an important metric for all compilers. For static compilation, i.e., where the program is compiled once *ahead-of-time* (AOT), long compilation times are often *set off* as an unfavorable but necessary condition.[7,8,9] The picture changes when talking about dynamic or *just-in-time* (JIT) compilation systems. Since compilation happens at application run time, it is an integral part of the performance of a system. No matter how good the quality of the compiled code is, if the code is not ready quickly enough, the user will be unsatisfied. Therefore, JIT compilation always has to consider a compile-time/code-quality trade-off.

## 1.3 Linear Scan

Poletto and Sarkar [1999] proposed the *linear scan* approach as a fast register allocator that produces reasonable results for the code generation system `tcc` [Poletto et al., 1997]. Instead of coloring an interference graph, linear scan performs a linear pass over the lifetime intervals of the method. First, the basic blocks of the control-flow graph are organized in a linear manner and instructions are numbered in ascending order.[10] Lifetime intervals are defined by a *start* and *end* position which represent the first and last occurrence of the respective variable in the linear stream of instructions. To allocate registers, the intervals are visited in the order of increasing start positions. The visited interval is moved into an *active* list, which is sorted by increasing end positions. Whenever a new interval is *activated*, intervals with an *end* position less than the new interval's *start* position are removed from the *active* list. In case the *active* list is longer than the number of registers, intervals need to be spilled. In Poletto and Sarkar's implementa-

[7] Exhibit A: `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=27140`
[8] Exhibit B: `https://www.rust-lang.org/en-US/faq.html#why-is-rustc-slow`
[9] Exhibit C: `https://lists.llvm.org/pipermail/llvm-dev/2016-March/096488.html`
[10] For the algorithm, the block order does not matter. However, it has a big impact on code quality [Wimmer and Mössenböck, 2005].

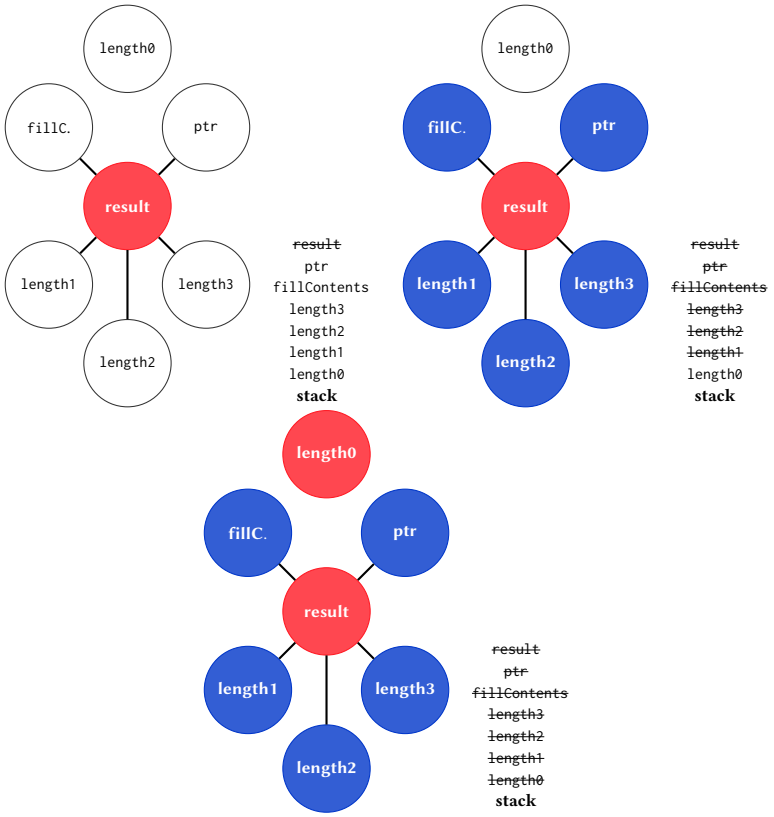



**Figure 1.9:** Graph coloring of `allocArray` using two colors (select)

tion, they choose the interval with the highest *end* position. When an interval is spilled, it is completely kept on the stack throughout its lifetime, also if it was previously selected to reside in a register.

Figure 1.12 illustrates how the linear scan algorithm would allocate our running example. We restricted the set of available registers to *two*. However, since some instructions require a register operand and cannot deal with stack slots directly, we need an additional scratch register.

### 1.3.1 Lifetime Holes and Interval Splitting

Two shortcomings of Poletto and Sarkar's linear scan approach are eye-catching. First, it always spills the whole interval, whereas it would be sufficient to split the interval at the position where the register pressure is too high. Also, there might be a register available later, so spilling it indefinitely is overly conservative. The second issue is about liveness. When linearizing the blocks of a control-flow graph, the liveness of a variable is not a continuous range, but a list of ranges. This means that intervals have *lifetime holes* (see Section 2.3) where the variable is not needed.

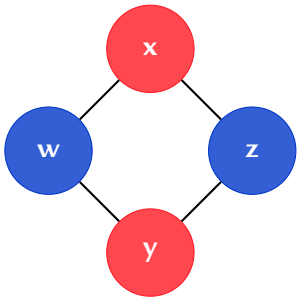



Figure adapted from [Briggs et al., 1994, Figure 2].

**Figure 1.10:** Diamond-shaped interference graph diamond which is 2-colorable

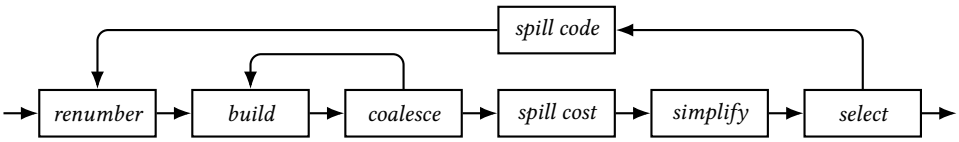
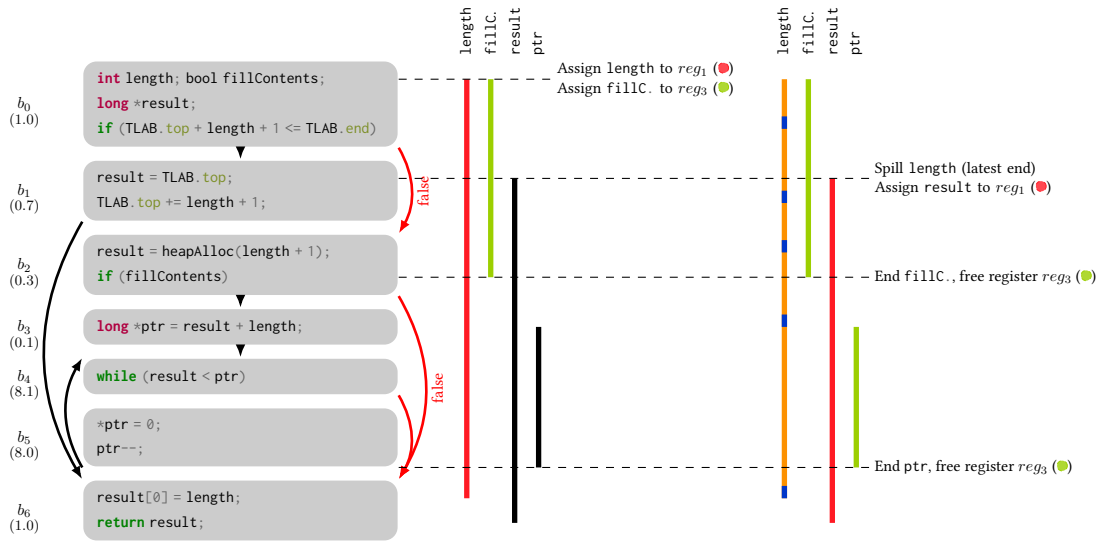



Figure adapted from [Briggs et al., 1994, Figure 4].

**Figure 1.11:** Briggs' optimistic allocator

Traub et al. [1998] introduced both improvements shortly after the first disclosure of linear scan by Poletto et al. [1997]. They proposed modeling liveness of a variable in linear scan as a list of live ranges. The emerging lifetime holes reduce the register pressure and therefore the need for spilling. If there is still need for spilling, the interval that is selected for spilling is split just before the register pressure exceeds the number of available registers. The first part of the interval stays in a register. The remaining part of the interval is allocated to a stack slot. In addition to that, the interval is split again just before the next usage of the variable and is inserted into the list of unhandled intervals. In other words, the interval gets a *second chance* of getting into a register. Thus the name of the approach, *second-chance binpacking*. Due to the splitting technique, Traub et al.'s allocator selects the spill candidate based on the next use position of the interval. Interval splitting also leads to data-flow mismatches because a variable might reside in different locations (i.e., in different registers or in a register and in a stack slot) at both ends of a control-flow edge. Therefore, a *resolution phase* iterates the edges and fixes mismatches by inserting move instructions.

Wimmer and Mössenböck [2005] extended the *second-chance binpacking* approach to further improve the allocation quality. The *optimal split position* optimization moves the split position (i.e., the position where a value is moved from a register to a stack slot) out of loops. If this succeeds, the spill move is executed less often, which has a positive impact on the performance of the generated code. Wimmer and Mössenböck also proposed *register hints* as a light-weight alternative to coalescing. If two intervals are linked via a move instruction, the destination interval tries to reuse the register of the source interval, if it is available, i.e., if the interval ends at the move. Furthermore, they distinguish between usages of a variable that require a register and those which




(a) Control-flow graph (b) Before $b_1$ (c) Allocation finished

Allocation example with two registers, $reg_1$ (🔴) and $reg_3$ (🟢). Note, that without interval splitting we need a scratch register $reg_2$ (🔵) to temporarily reload spilled values.

**Figure 1.12:** Poletto and Sarkar-style linear scan allocation example with two registers

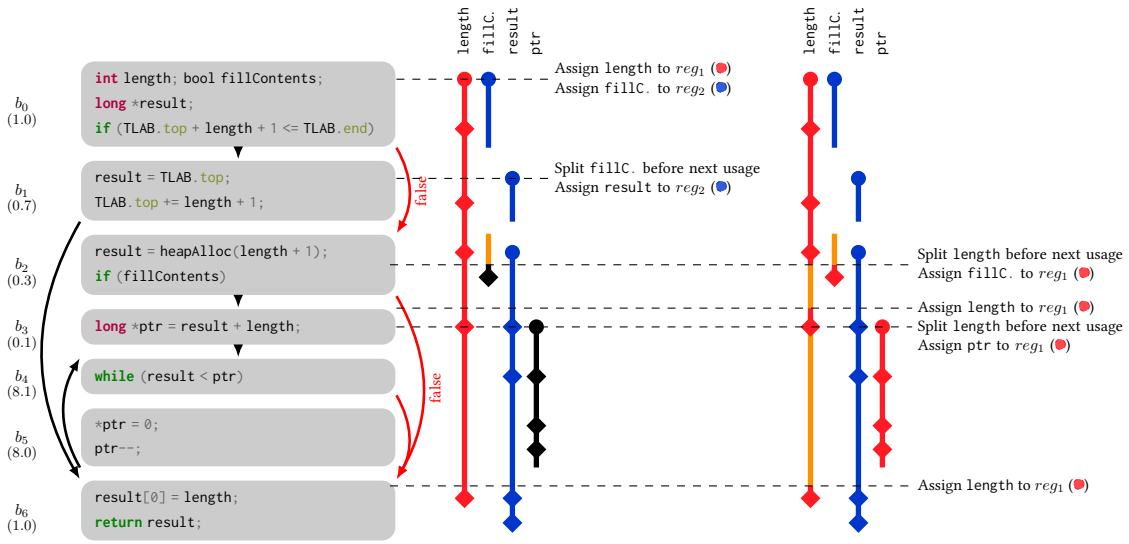
could directly address a variable on the stack. The latter type is common on CISC machines (see Section 7.3.1) like the AMD64. The optimization reduces the number of reloads of an interval, which would potentially cause more spilling.

Figure 1.13 shows an example of the interval-splitting linear scan allocation. In contrast to the original approach, linear scan with interval splitting does not require a scratch register and can allocate `allocArray` with two registers. The interval for `fillContents` illustrates how an already assigned interval is split and moved onto the stack ($b_1$) and later reloaded to a different register ($b_2$).

**Discussion** The interval splitting approaches are significantly more difficult to implement than the original approach. First, the interval representation has to support multiple ranges to accommodate lifetime holes and usage positions for guiding splitting decisions. Second, the data-flow mismatches call for a resolution phase. Finally, due to the extensions the time complexity of the algorithm is no longer linear [Wimmer and Franz, 2010].

Despite these issues, the interval splitting variant is commonly used in compilers today, for example Google's Java Script engine V8,[11] in the HotSpot client compiler,[12] or Apple's WebKit.[13] One reason is that in practice interval splitting still exhibits linear compile time behavior [Wimmer

[11] *Linear Scan Register Allocator in V8* by V8 [2017]
[12] *Linear Scan Register Allcoator in WebKit* by WebKit [2017b]
[13] *Linear Scan Register Allocator in C1* by OpenJDK [2017b]




(a) Control-flow Graph (b) After Activating `result` in $b_1$ (c) Allocation Finished

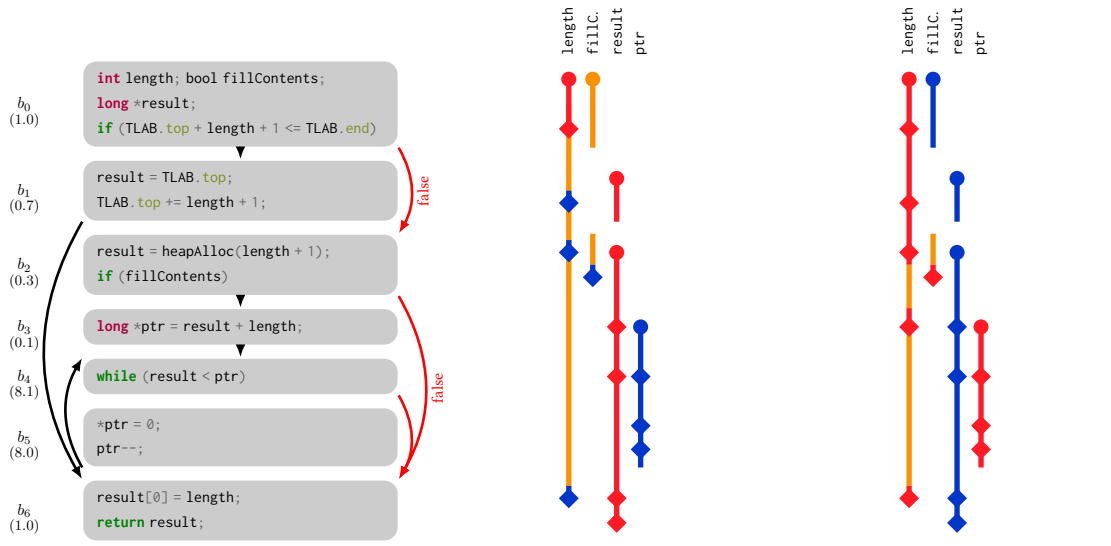
Allocating `allocArray` with an interval splitting linear scan allocator as described by Traub et al. [1998] or Wimmer and Mössenböck [2005] with two registers, $reg_1$ (🔴) and $reg_2$ (🔵). Interval activations (dashed line) occur from top to bottom.

**Figure 1.13:** Interval-splitting linear scan allocation of `allocateArray` with two registers

and Mössenböck, 2005]. The other, more compelling reason is that spilling the whole interval leads to worse allocation results, especially with increasing numbers of variables and lengths of intervals. For the rest of this work, unless otherwise noted, we refer to the Wimmer and Mössenböck approach when talking about linear scan.

## 1.4 Problems of Existing Register Allocation Approaches

Figure 1.14 visualizes the graph coloring and the linear scan allocation examples with two registers side-by-side. Both approaches introduce *spilling* in the *common case* ($b_0$, $b_1$ and $b_6$). Graph coloring spills `fillContents` and `length` in $b_0$ and reloads `length` in $b_6$. Linear scan performs better in this example and only reloads `length` before its usage in $b_6$. Of course, we have carefully selected this example to make our case. Our graph coloring example uses a rather simple splitting heuristic (reload at every usage). An advanced approach, for example as described by Bergner et al. [1997], would improve the result. Also linear scan would perform better if the block order were different, for example if the important blocks come first. However, these workarounds just underline our criticism of the approaches. They use "tricks" to such as block orderings to push the algorithm into the right direction or heuristics to reduce obvious deficiencies. But the underlying problem—distinguishing between common and uncommon cases—is not modeled ap-




(a) Control-flow graph (b) Graph coloring (c) Interval-splitting linear scan

Comparison of graph coloring and linear scan allocation with two registers, $reg_1$ (🔴) and $reg_2$ (🔵). See Figure 1.9 and Figure 1.13 for the full allocation example.

**Figure 1.14:** Graph coloring vs. linear scan of `allocateArray` with two registers

propriately. For small compilation units, like our example, most well-tuned approaches would probably come up with equally good results. When the method size increases, however, the shortcomings become more evident.

The issues are especially problematic in JIT compilers. First, there are compile time constraints due to compilation at run time. In addition, many JIT compilers perform *speculative optimizations* [Duboscq et al., 2013], based on profiling information (e.g., inlining a specific call target of a *virtual call*, although there might be multiple candidates). This allows the compiler to perform optimizations aggressively, which is not possible in a static compiler. The compiled code contains checks to verify that the *assumptions* on which the optimizations were based still hold. If not, *deoptimization* takes place, i.e., execution is continued in an unoptimized version or in an interpreter [Hölzle et al., 1992; Wimmer et al., 2017]. In order to do so, the system needs to reconstruct the unoptimized (interpreter) *state*. Therefore, all local variables in the current stack *frame* need to be kept alive, although they are not needed by the optimized code [Duboscq et al., 2014]. Multiple levels of inlining means that multiple frames need to be reconstructed. Due to this, compilation units in JIT compilers often consists of tens of thousands of instructions (see Figure 7.7) and variables [Chen et al., 2018]. This poses a challenge for register allocation.

### Issues

Let us summarize the issues we have uncovered so far with graph coloring, linear scan and the global approach in general.

**Graph Coloring** The inference graph used by graph coloring models register requirements on the level of variables, nicely. However, due to the quadratic worst-case graph size, the applicability of this approach to huge compilation units is limited [Sarkar and Barik, 2007]. Also, the graph coloring model loses a lot of its appeal when it comes to spilling and splitting, since the information that is necessary for doing it is not an inherent part of the graph. However, for the performance of the generated code, making good spilling and splitting decisions is highly important [Braun and Hack, 2009; Lozano et al., 2014]. Finally, the non-linear time complexity rules out graph coloring for most JIT compilers.

**Linear Scan** In linear scan, interference is only implicit in the entries of the *active* list, which represents a local snapshot of *live* variables currently stored in registers. Due to this local view, the scope for decisions is also limited. For example, spilling an interval might render a previous spilling decision unnecessary. On the other hand, splitting the lifetime of a variable is more natural in linear scan than in a graph coloring allocator. There is a closer connection between the instructions and the liveness representation. However, the linearization of the control flow is another issue of linear scan. Consecutive blocks might not have a control-flow dependency, even though they are processed as if they were executed sequentially. *Lifetime holes* are an incarnation of this dilemma.

**Global Scope** With increasing problem sizes, dealing with the whole compilation unit in a (global) register allocator becomes a problem. The allocation time depends on the size of the input, often in a non-linear way. Optimizations target the complete compilation unit. However, as already stated, not all parts are equally important. A bad global decision (i.e., spilling due to high register pressure in an infrequently executed part of the method) may cause degradation for the common case, as we have seen in our example in Figure 1.14. When compile time is scarce, we want to be more clever about where to spend it. We would like to invest more time on important parts of the compilation unit and less time on others. The global view makes this difficult.

## 1.5 Our Approach

The deficiencies outlined in the previous section motivated our investigations for finding a new register allocation approach that solves these problems. More specifically we envisioned the following goals.

*Register Allocation for JIT Compilers.* Striving for optimal register allocation is easier if compile time is not an issue. We explicitly target just-in-time compilers where the compile-time/peak-performance trade-off is a challenge.

*Non-global approach.* Instead of dealing with the whole compilation unit at once, we want to divide the input into sub-problems and solve them independently. This would allow us to deal with bigger problem sizes, provide more flexibility and enables concurrent allocation.

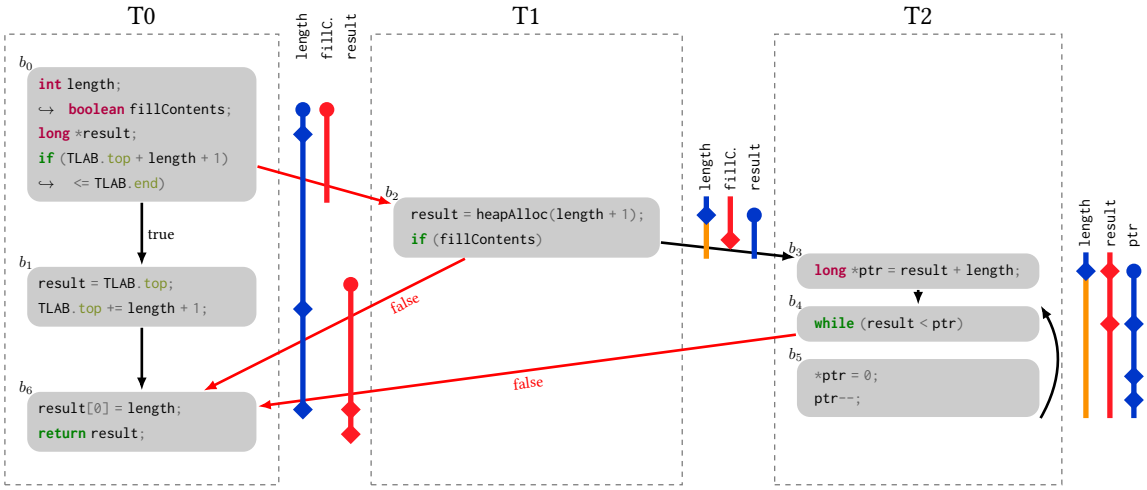
*"Focus on the Common Case"* is an often heard suggestion [Hennessy and Patterson, 2003, p. 38]. We want to concentrate on those parts which will contribute most to the overall performance of the generated code and find a good solution for those. The quality of infrequently executed parts of a method is of low interest. Since we target virtual machines that provide profiling information, we want to exploit this knowledge.

*Natural liveness model.* Liveness is the key information for a register allocator. We want a simple liveness model which is easy to reason about. In particular, we want to avoid lifetime holes. Therefore, the control flow of our sub-problems should to be linear.

*Same or better allocation quality.* Although we push for a simpler model, we still want to be able to achieve the same allocation quality as global approaches.

*Same or better compile-time behavior* for the same allocation quality. Compile time is our major concern. The minimum goal is to compete with linear scan with respect to allocation time for comparable allocation quality.

*Better scaling for large compilation units.* A main challenge is to cope with increasing sizes of compilation units. We want an allocator that scales well as the problem grows. It is especially important to work towards a linear or almost linear compile time with respect to the input size.




Allocating `allocArray` with trace register allocation using two registers, $reg_1$ (🔴) and $reg_2$ (🔵). Each trace (dashed rectangles) is allocated independently. Note that the locations of the variables at the trace boundaries do not always match. A *resolution phase* inserts *moves* to fix the data-flow.

**Figure 1.15:** Trace register allocation of `allocArray()` with 2 registers

*More flexibility* to control the trade-off between compile time and allocation quality. We want *means* to trade-off allocation quality for compile time on a fine-grained level. In particular, we want to be able to decide which parts of a method are particularly important for peak performance and where we thus want to spend more compile time.

**Idea** The requirements outlined above led a novel register allocation framework which we entitled *trace register allocation*. In contrast to *global register allocation*, which processes a whole method at once, *trace register allocation* divides the problem into smaller sub-problems, so-called *traces*, for which register allocation can be done independently. A *trace* is a list of basic blocks, which might be executed sequentially. Traces are non-empty and non-overlapping, and every basic block is contained in exactly one trace. We use profiling information provided by the virtual machine to construct long and important traces first.

For each trace, the algorithm selects an *allocation strategy*. Due to the explicit global liveness information, allocating a trace is completely decoupled from the allocation of other traces. The linear structure of traces makes the implementation of strategies significantly simpler than in a global algorithm. Most importantly, there are no lifetime holes. Since some traces are more important than others, we use different allocation algorithms (strategies), depending on the trace. For infrequently executed traces we use a strategy that is fast but produces sub-optimal code. For important traces, on the other hand, we want the best allocation quality and are willing to invest more compile time.

**Example** Figure 1.15 shows the control-flow graph of `allocArray` divided into traces. Each trace has been allocated independently with two available registers. In contrast to the graph coloring and linear scan example, there are no spill moves or reloads in the *common case* ($b_0$, $b_1$, $b_6$). At some inter-trace edges variables are stored in different locations. For example, `length` resides on the stack at the end of block $b_2$ but in register $reg_1$ at the beginning of $b_6$. The resolution phase inserts *moves* to correct the data-flow.

## 1.6 Contributions

In the course of this thesis, we contributed the following scientific results:

1. The *Trace Register Allocation Framework*. It is implemented in GraalVM, a production-quality Java virtual machine developed by Oracle. The code is open source and is publicly available for everyone.[14] As of version 9, the Graal compiler is an experimental part of the JDK. Therefore, our trace register allocation approach is now part of the most widely used Java environment (although it is not enabled by default).[15, 16] The implementation includes three register allocation strategies for processing individual traces.

   *Linear Scan* The *trace-based linear scan* strategy is an adaption of the global approach by Wimmer and Mössenböck [2005] and Wimmer and Franz [2010] to the properties of a trace. The main difference of our approach is that there is no need to maintain a list of live ranges for each lifetime interval, since there are no lifetime holes in trace intervals [Eisl et al., 2016].

   *Bottom-Up* To decrease compilation time, we implemented the *bottom-up* allocator [Eisl et al., 2017]. Its goal is to allocate a trace as fast as possible, potentially sacrificing code quality. Our experiments show that it is about $40\%$ faster than the linear scan strategy.

   *Trivial Trace* The *trivial trace* allocator is a special-purpose allocator for traces which have a specific structure. They consist of a single basic block which contains only a single *jump* instruction. Such blocks are introduced by splitting *critical edges* (see Section 2.2), and are quite common. For the DaCapo benchmark suite about $40\%$ of all traces are trivial [Eisl et al., 2016]. A trivial trace can be allocated by mapping the variable locations at the beginning of the trace to the locations at the end of the trace.

[14] `https://github.com/oracle/graal`
[15] On Java 10 and later:
`java -XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler -Dgraal.TraceRA=true ...`
[16] On Java 9 an additional `-XX:+EnableJVMCI` is required.

2. We showed that our trace register allocation approach achieves similar allocation quality as a state-of-the-art global linear scan allocator for common Java benchmarks on both AMD64 and SPARC processors [Eisl et al., 2016]. This suggests that high-quality register allocation can be achieved without a global approach.

3. We showed that we are *on par* with global linear scan with respect to allocation speed for the same allocation quality [Eisl et al., 2017].

4. Furthermore, we demonstrated the flexibility of using different strategies for allocating individual traces, which allows us to control the trade-off between allocation quality and compile time [Eisl et al., 2017]. We can, for example, save over 10% of register allocation time with a (geometric) mean performance degradation of only 1.5%. On the other end of the spectrum, we get over 40% faster allocation time at a mean slowdown of 11%. Depending on the requirements, all trade-offs between those two boundaries can be achieved via simple parameter settings. This flexibility is not possible with other approaches.

5. We prototyped an extension that enables parallel allocation of traces by multiple threads without a negative impact on allocation quality [Eisl et al., 2018]. This can reduce the *register allocation latency*, i.e., the duration until the register allocation result is ready. In our experiments, we decreased latency by up to 30% when using *four* threads instead of *one*. This is yet another showcase for the flexibility of the trace register allocation approach. To the best of our knowledge, it is the first time that parallelism was exploited for register allocation.

The trace register allocation results were published in the proceedings of multiple *peer-reviewed* venues. We presented a vision paper at the *Doctoral Symposium* of *SPLASH 2015*, where we motivated the trace register allocation idea [Eisl, 2015]. In 2016, we presented a full paper on allocation quality of a trace-based register allocator at the *International Conference on Principles and Practices of Programming on the Java Platform* (PPPJ) [Eisl et al., 2016]. At *ManLang 2017*, the *International Conference on Managed Languages & Runtimes*, we demonstrated the flexibility of our approach with a full paper on *Trace Policies*. We successfully submitted an extended abstract on trace register allocation to the *ACM Student Research Competition* at *CGO 2018* (*Symposium on Code Generation and Optimization*), where our poster and presentation won the *first price* in the *graduate category* [Eisl, 2018a]. The award made us eligible to compete in the *2018 Student Research Competition Grand Finals*, where we submitted a summary paper [Eisl, 2018b]. In 2018, a *work-in-progress paper* on parallel trace register allocation was accepted for the *International Conference on Managed Languages & Runtimes* (ManLang 2018) where we reported our experience with allocating traces by multiple threads [Eisl et al., 2018].

## 1.7 Outline

The rest of this thesis is organized as follows. In Chapter 2 "*Terminology*" we introduce notations used throughout this work. All terms are commonly used in related literature, so feel free to skip it. Chapter 3 "*The Graal Virtual Machine*" discusses GraalVM, the Java Virtual Machine in which we implemented our trace register allocation approach. Chapter 4 "*Trace Register Allocation*" introduces the core of our allocator in detail. It also describes notations, definitions and properties of traces which might not be commonly known. Chapter 5 "*Register Allocation Strategies*" outlines three allocation strategies, a linear-scan-based, a bottom-up and a trivial trace strategy. In Chapter 6 "*Inter-trace Optimizations*" we discuss three inter-trace optimizations needed for reaching peak performance. Chapter 7 "*Evaluation*" evaluates our implementation in GraalVM and argues why it solves the problems discussed in this introduction. In Chapter 8 "*Trace Register Allocation Policies*" we use various policies to control the trade-off between compile time and peak performance. It showcases the flexibility and uniqueness of our trace register allocation approach. Chapter 9 "*Parallel Trace Register Allocation*" describes how parallelization can reduce the compilation latency by allocating traces concurrently. In Chapter 11 "*Conclusion and Future Work*" we finally reiterate the main contributions, discuss future work, and conclude the thesis.

# Chapter 2

# Terminology

Before we get into the details of this thesis we need to agree on the terminology and notation used in this work. All terms are commonly used in the compiler construction community, so feel free to skip this chapter if you are familiar with them. However, some terms are used ambiguously in literature. Therefore, we give here a proper definition.

## 2.1 Instructions, Values, Locations

First we describe the *atomic* building blocks of our intermediate representation and how they interact to produce new results.

**Definition 1** (Register)**.** For our work, we assume that registers are unique and do not alias [Lee et al., 2007], i.e., that modifying one register will leave the contents of all other registers untouched.[1]

**Definition 2** (Stack Slot)**.** In our model we have an infinite number of stack slots for spilling. Assumptions are similar to registers. Stack slots do not overlap and there are no aliasing effects.[2]

**Definition 3** (Constants)**.** Constants are fixed numeric values that never change.[3] We can use them for rematerialization, i.e., for reloading the constant instead of spilling and reloading it from a stack slot.

[1] Register aliasing is a truly interesting and hard problem [Lee et al., 2007]. However, since there is no—or only very limited—support for describing aliases in our implementation platform Graal, respectively HotSpot, we did no research into this direction.

[2] In practice this is not entirely true. It is possible to get the address of a stack slot. However, these cases are handled very carefully by the compiler and they do not affect register allocation.

[3] Again, this is not always true. There are *Java Object constants*, i.e., the address of an object in memory. However, garbage collection might move the object and therefore change the address. Therefore, the compiler notifies the virtual machine about usages of object constants. If the object is moved, the machine code is patched accordingly.

**Definition 4** (Variable)**.** Others distinguish between *temporaries* (i.e., the intermediate results of expressions) and *variables in the source code* of the program. On the level of our intermediate representation, we do no longer (directly) see source variables. We call every non-physical, non-constant operand in the intermediate representation of the compiler a *variable*. In our terminology, variables are all operands that are subject to register allocation.

**Definition 5** (Value)**.** With *value* we refer to the content of a variable, register, stack slot, or constant. For example, we say that "a value is moved from one register to another."

**Definition 6** (Location)**.** The *location* of a variable is its *physical storage*, i.e., the register or stack slot after register allocation at a certain point in the program. A variable might get assigned to different locations at different program points.

**Definition 7** (Instruction)**.** An instruction represents an operation in the intermediate representation. Instructions can have one or multiple *input* and *output operands*. In most cases, the register allocator does not know the semantics of an instruction. However, there are exceptions. The allocator knows that *moves* copy the *value* of its *source* to its *destination operand*. *Constant load* instructions are a special kind of move where the source is a *constant*. In addition, we know about *labels*, which mark the beginning of a basic block, as well as *jumps*, which model unconditional control-flow transitions.

## 2.2 Control-flow Graphs

The *control-flow graph* is the compiler-internal representation of a *compilation unit*. A compilation unit is the input to the compiler. Usually, it represents a (Java) method, potentially with inlined methods calls. For our work, however, the origin of the input does not matter.[4] Therefore, we prefer the more generic term *compilation unit* over *method*. However, both terms are used interchangeably in this work. We will also talk about *program points*, which are certain positions (i.e., instructions) in the compilation unit.

**Definition 8** (Basic Block)**.** A basic block $b \in Blocks$ is a maximum-length sequence of branch-free instructions without side entries. In our case, all basic blocks start with a *label* instruction and end with a conditional or unconditional jump.

**Definition 9** (Edge)**.** An edge $(b_{source}, b_{dest}) \in Edges$ is a tuple of two basic blocks $b_{source}, b_{dest} \in Blocks$.

[4] A compilation unit can for example also stem from a *partially-evaluated* Truffle AST (see Chapter 3).


(a) Critical edge $(b_1, b_3)$ (b) After splitting: new block $b_c$

**Figure 2.1:** Critical edge splitting

**Definition 10** (Predecessors). $pred(b)$ is a list of *predecessor* blocks of $b$, i.e., $pred(b) = [b_p \in Blocks \mid (b_p, b) \in Edges]$.

**Definition 11** (Successors). $succ(b)$ is a list of *successor* blocks of $b$, i .e., $succ(b) = [b_s \in Blocks \mid (b, b_s) \in Edges]$.

**Definition 12** (Path). A path $\langle b_0, \ldots, b_n \rangle$ is a sequence of blocks where $b_{i+1}$ is a successor of $b_i$ for $i = 0, \ldots, n-1$. A path may consist of a single block, i.e., $\langle b \rangle$ for $b \in Blocks$.

**Definition 13** (Reachability). A block $b_y \in Blocks$ is *reachable* from a block $b_x \in Blocks$ if and only if there is a path $\langle b_x, \ldots, b_y \rangle$. We write $b_x \rightsquigarrow b_y$.

**Definition 14** (Control-Flow Graph). A control-flow graph is a tuple $CFG = (Blocks, Edges)$ consisting of a set of basic blocks and a set of edges. By definition, there is only one block in Blocks without a predecessor which we call the *entry block* $b_{entry}$. In addition, every block $b$ is reachable from the entry block, i.e., there is a path $\langle b_{entry}, \ldots, b \rangle$ for all $b \in Blocks$.

An example of the source code of `allocateArray` and the corresponding control-flow graph is shown in Figure 2.2.

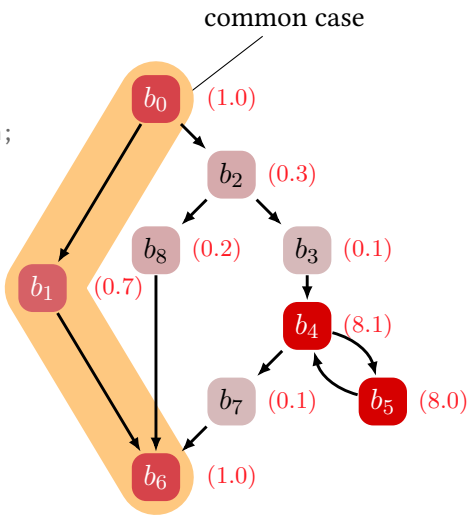
**Definition 15** (Critical Edge). An edge $e = (b_{source}, b_{dest})$ is *critical* if and only if $|succ(b_{source})| > 1$ and $|pred(b_{dest})| > 1$. In other words, the *source* block of $e$ has multiple successors and the *destination* block of $e$ has multiple predecessors.

See Figure 2.1a for an example of a critical edge. Critical edges can be removed by introducing new (empty) blocks (Figure 2.1b). For data-flow resolution, critical edge splitting is of vast importance. The newly introduced blocks provide a place for compensation code, i.e., moves from one location to another.

Our introductory example in Figure 1.1 contains two critical edges, from $b_2$ to $b_6$ and from $b_4$ to $b_6$. Figure 2.2 shows the example after critical edge splitting. Assume that there is a data-flow mismatch between $b_2$ and $b_6$. If the compensation code were placed at the end of $b_2$ it would also affect $b_3$. If it were placed at the beginning of $b_6$ it would also be executed for paths from $b_1$ and $b_4$. Therefore, the new block $b_8$ is required.

```
long *allocateArray(int length, boolean fillContents) {
        long *result;
/*B0*/  if (TLAB.top + length + 1 <= TLAB.end) {
/*B1*/      result = TLAB.top;
            TLAB.top += length + 1;
        } else {
/*B2*/      result = heapAlloc(length + 1);
            if (fillContents) {
/*B3*/          long *ptr = result + length;
/*B4*/          while (result < ptr) {
/*B5*/              *ptr = 0;
                    ptr--;
                }
/*B7*/          // critical edge
            } else {
/*B8*/          // critical edge
            }
        }
/*B6*/  result[0] = length;
        return result;
}
```

(a) C-like pseudo source code




(b) Corresponding control-flow graph

The method `allocateArray()` after *critical edge* splitting which introduced $b_7$ between $b_4$ and $b_6$, and $b_8$ between $b_2$ and $b_6$. See Figure 1.1 for more information about the code snippet.

**Figure 2.2:** The `allocateArray()` sample code snippet (without critical edges)

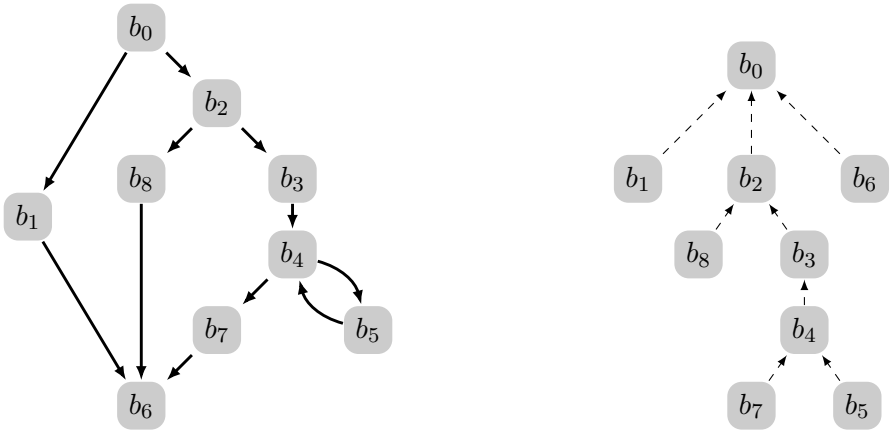
For the rest of this thesis, we assume that all critical edges in the CFG have been split, so that there are no more critical edges.

### 2.2.1 Dominance

**Definition 16** (Dominator)**.** A block $b_d \in Blocks$ is a *dominator* of a block $b \in Blocks$ if every path $\langle b_{entry}, \ldots, b \rangle$ includes $b_d$. We say that $b_d$ *dominates* $b$ and define $dom(b)$ as the set of all blocks that dominate $b$. Per definition, a block $b$ dominates itself. It is easy to see that *dominance* is a transitive relation.

**Definition 17** (Strict Dominator)**.** The strict dominators of a block $b$ are the dominators of $b$ without $b$ itself, i.e., $sdom(b) = dom(b) \setminus b$.

**Definition 18** (Immediate Dominator)**.** The immediate dominator of $b$, denoted as $idom(b)$, is the strict dominator of $b$ that is *dominated* by all other strict dominators of $b$. In other words, it is the strict dominator that is closest to $b$. Every node except the *entry node* $b_{entry}$ has exactly one immediate dominator.




(a) Control-flow Graph (b) Dominator Tree

On the left the control-flow graph of `allocateArray()`, on the right the *dominator tree* of that method. The dominator tree is a visualization of the *immediate dominator* relation. Source code in Figure 2.2a.

**Figure 2.3:** Dominator tree for `allocateArray()`

Figure 2.3b shows the dominator tree for the control-flow graph of `allocateArray` in Figure 2.3a. The edges represent the *immediate dominator* relation.

**Definition 19** (Dominance Order)**.** In dominance order, a block is visited only *after* all its dominators have been visited.

### 2.2.2 Loops

**Definition 20** (General Loops)**.** A loop is a set of basic blocks $L \subset Blocks$ so that there is a path of at least length 2 from every block $b_x \in L$ to every other block $b_y \in L$, i.e., $\forall b_x, b_y \in L \mid |\langle b_x, \ldots, b_y \rangle| \geq 2$.

**Definition 21** (*Reducible* Loops)**.** A *reducible* loop $L$ is a loop where there is a loop header $b_h \in L$ which dominates all blocks $b_i \in L$. As a result, $b_h$ is the only block in $L$ that can be targeted by an edge from a block outside of the loop. Edges from a block $b_e$ in the loop to the loop header are called *back edges*. Any such block $b_e$ is called a *loop end*. An edge from a block in the loop $L$ to a block not in $L$ is called a *loop exit*.

In this work we only deal with reducible graphs, i.e., with CFGs that only contain *reducible* loops.

**Theorem 1.** Every path $\langle b_x, \ldots, b_d \rangle$ from a block $b_x$ to a strict dominator $b_d \in sdom(b_x)$ contains a loop header and therefore a back edge.

*Proof.* Since $b_x \rightsquigarrow b_d$ and $b_d \rightsquigarrow b_x$, there is a loop $L$ which contains $b_x$ and $b_d$. Therefore there must be a loop header $b_h$ which dominates both and is on every path from $b_x$ to $b_d$. □

**Theorem 2.** Every loop has at least one back edge.

*Proof.* By definition, there is a path from every block $b$ in a loop to every other block in this loop; in particular, there must be a path $\langle b, \ldots, b_e, b_h \rangle$ from $b$ to the loop header $b_h$. By definition, the edge $(b_e, b_h)$ is a back edge. □

**Theorem 3.** Every loop header has at least one predecessor from outside the loop, which is called the loop entry.

*Proof.* By Definition 14, $b_{entry}$ has no predecessor, so it cannot be a loop header. Since every block is reachable from the entry block $b_{entry}$, there must be a path $\langle b_{entry}, \ldots, b, b_h \rangle$ of at least length 2 where $b \notin L$, otherwise $b_h$ would not be the header of $L$. Thus, there is an edge $(b, b_h)$ that is not a back edge. □

**Corollary 1.** Let us assume that a CFG has no critical edges, every predecessor of a loop header has a single successor, namely the loop header.

*Proof.* From Theorem 2 (at least one back edge) and Theorem 3 (at least one loop entry) we know that a loop header has at least two predecessors. Therefore, all predecessors must have a single successor otherwise there would be a critical edge. □

**Definition 22** (Reverse Postorder)**.** In reverse postorder, a block is visited before any of its successor blocks are visited unless the successor is a loop header. Reverse postorder implies dominance order.

## 2.3 Liveness and Lifetime Intervals

Liveness is the most important concept for register allocation. The liveness of the variables in the compilation unit determines whether the problem is hard or simple to solve.

**Definition 23** (Instruction Path)**.** An instruction path is a list of instructions that occur between a *start instruction* $i_s$ in block $b_s$ and an *end instruction* $i_e$ in block $b_e$. If both instructions are in the same block, the instruction path consists of all instructions between the two instructions (inclusive). If there is no (basic block) path between $b_s$ and $b_e$, then the instruction path is empty.

Otherwise, the instruction path contains all instructions from $i_s$ to the end of $b_s$, all instructions from the blocks along the path $\langle b_s, \ldots, b_e \rangle$ excluding $b_s$ and $b_e$, and all instructions from the beginning of $b_e$ to $i_e$.

**Definition 24** (Liveness)**.** A variable is *live* at every instruction that is part of an instruction path between *any* definition of the variable and *any* usages of the variable, for all instruction paths that do not contain another definition of that variable. Note that we exclude the defining instruction itself from the instruction path, i.e., a variable is live right *after* its definition.

Figure 1.3a in the introduction already showed *lifetime intervals*, which are a representation of *liveness*.

**Definition 25** (Interference)**.** Two variables *interfere*, it they are *live* at the same instruction. Since we exclude the defining instruction from *liveness*, a usage of variable $x$ in instruction $i$ does not interfere with the variables *defined* by $i$, unless $x$ is *live* after $i$.

**Definition 26** (Interference Graph)**.** An interference graph is a graph where the nodes represent variables and an edge between two nodes indicates that the variables of these nodes interfere.[5]

In Figure 1.3b in the introduction, we saw an example of an interference graph for the method `allocateArray`.

**Definition 27** (Lifetime Hole)**.** A *lifetime hole*, a term proposed by Traub et al. [1998], is a consequence of the linearization of the basic blocks of the control-flow graph. It occurs whenever a variable is *not live* in a block $b_h$, but live in a block $b_b$ before $b_h$ and live in a block $b_a$ after $b_h$, with respect to the linear block order $[\ldots, b_b, b_h, b_a, \ldots]$.

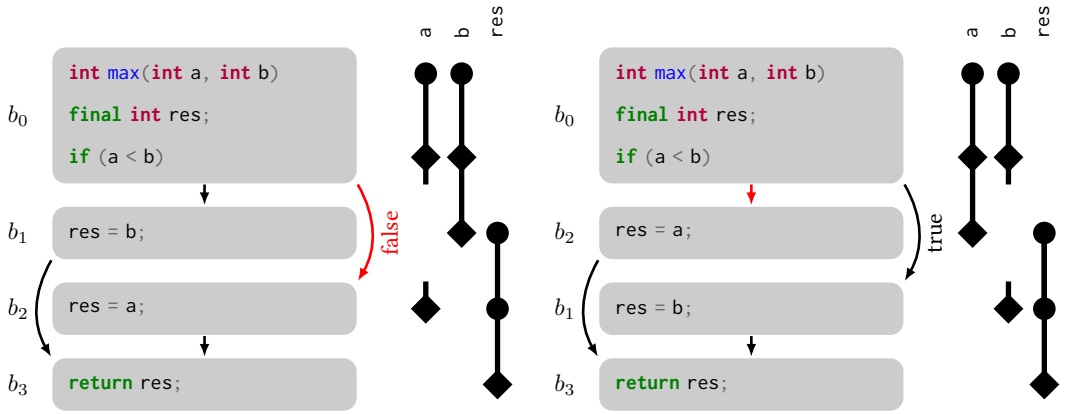
Figure 2.4 demonstrates an example for an *inevitable* lifetime hole. For every possible block order, there is at least one lifetime hole.

**Definition 28** (Live Range)**.** In a linear list of instructions, a *(live) range* denotes a subset of the instructions where a variable is continuously *live*, i.e., there is no lifetime hole. A range can be described by a *from* and *to* position.

**Definition 29** (Lifetime Interval)**.** The term *(lifetime) interval* denotes a collection of one or more unconnected and non-overlapping live ranges.

We introduced the notions of liveness in terms of *variables*. However, the same definitions can be applied to *registers* and *stack slots*.

[5] To be more precise we should add that they also need to belong to the same *register class*, i.e., they compete for the same set of registers. For readability reasons, we assume that there is only a single register class.




(a) `false` case after `true` case

(b) `true` case after `false` case

Example for an inevitable *lifetime hole*. For every block ordering the intervals for either `a` or `b` will be non-continuous. `final int res` in $b_1$ ensures a single assignment.

**Figure 2.4:** Example for an inevitable lifetime hole

## 2.4 Static Single Assignment Form

Static single assignment form (SSA form, Cytron et al., 1991) is a flavor of an intermediate representation that is very common in modern compilers [Rastello, 2013]. There are different SSA variants with different requirements, for example regarding *minimalism*. We will only describe the most basic properties, which are common in all of them, since they are what we rely on later.

**Property 1** (Single Definition Property)**.** The *static single assignment* property mandates that for every variable there is exactly *one* definition in the program.

This property is sometimes called *referential transparency* [Rastello, 2013, Chapter 1.1], i.e., the value of a variable does not depend on its position in the code. It is important to grasp that this is a *static* property of a program. *Dynamically*, the value of a variable can change, for example if the *same definition* is executed multiple times in a loop. Still, this property tremendously simplifies reasoning about variables. For example, every variable can be unambiguously identified by its definition.

Every program can be transformed into SSA form. Many algorithms have been proposed, for example by Cytron et al. [1991], Brandis and Mössenböck [1994], or Braun et al. [2013]. Intuitively, we introduce a new name for every assignment to a variable. Figure 2.5 shows an example. The situation is getting more complicated when we have control flow. See Figure 2.6a for example. The variable `res` is assigned in both branches, $b_1$ and $b_2$. In block $b_3$, however, the value of `res` depends on the dynamic predecessor. To solve this issue, SSA form introduces $\varphi$-functions. For

```
x = 1;
y = x + 1;
x = 2;
z = x + 1;
```

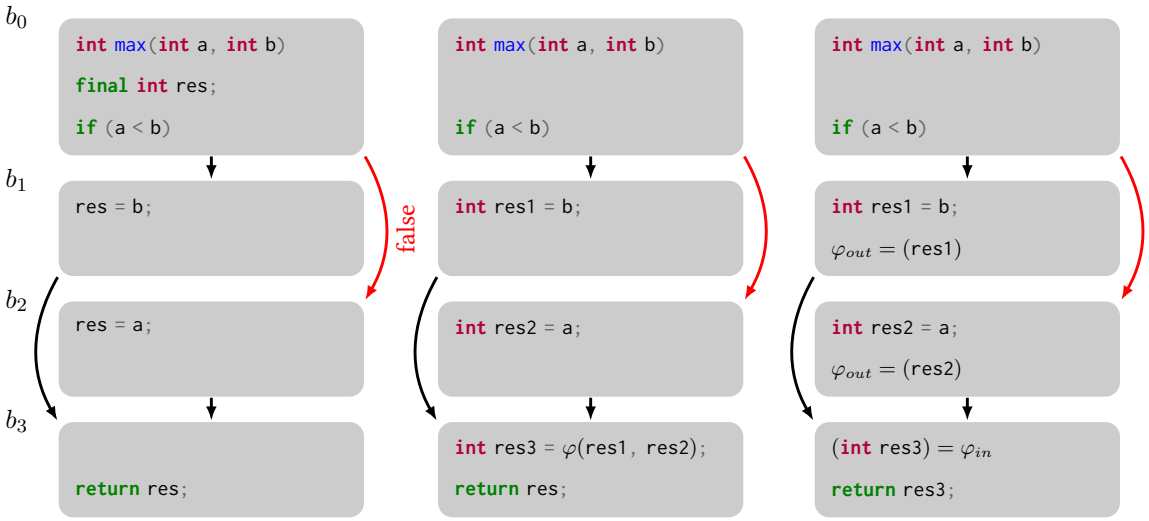
(a) Before SSA

```
x1 = 1;
y  = x1 + 1;
x2 = 2;
z  = x2 + 1;
```

(b) After SSA

Without SSA form one might assume that both occurrence of `x + 1` calculate the same value and that therefore `y` equals `z`. After renaming, it is clear that this is not the case. Example taken from Rastello [2013].

**Figure 2.5:** Variable renaming in Static Single Assignment form



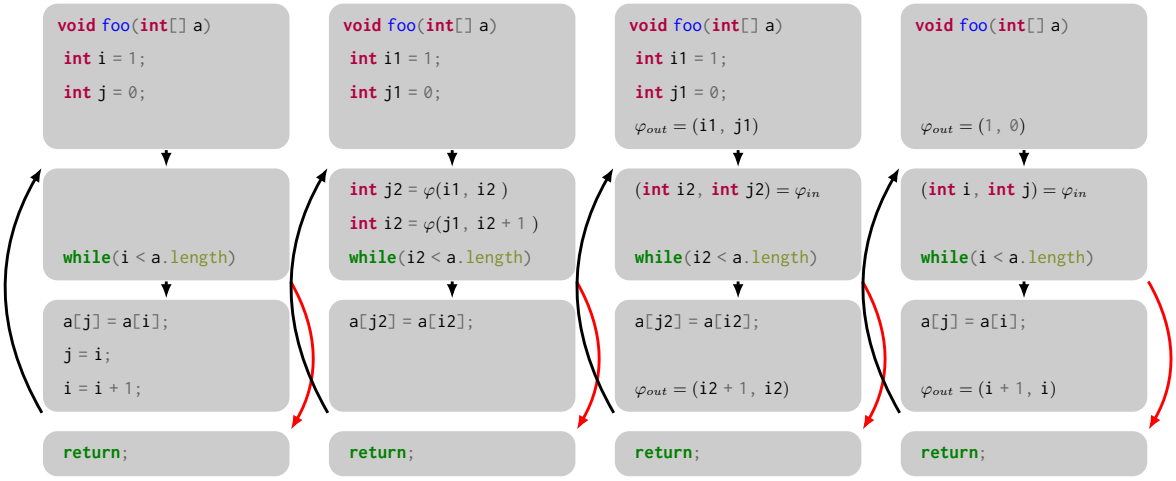

(a) Before SSA (b) With $\varphi$-functions (c) With $\varphi$-splitting

**Figure 2.6:** $\varphi$-function in Static Single Assignment form

every variable that has different values in different branches, a $\varphi$-function is introduced at the merge block. It defines a new unique name for the variable. Its value depends on the actual predecessor. In Figure 2.6b, `res3` has the value `res1` if $b_3$ is reached via $b_1$, or `res2` if reached via $b_2$.

**Property 2** (Dominance Property)**.** The *dominance property* requires the (single) definition of a variable to *dominate* all its usages. If the definition and the usages are in the same block, the definition must occur *before* the usage in the instruction list of the block. If the blocks are different, the definition block must dominate the block of the usage.

The dominance property simplifies liveness analysis significantly, since we know that a variable can only be live in blocks *after* its definition, if visited in *dominance order*. This allows building lifetime intervals in a linear pass over the blocks, as demonstrated by Wimmer and Franz [2010].




(a) Before SSA (b) With $\varphi$-function (c) With $\varphi$-splitting (d) Copy propagation

**Figure 2.7:** Parallel $\varphi$-copies

### 2.4.1 $\varphi$-notation

There are multiple ways of notating $\varphi$s. The most common way is the use of $\varphi$-functions, i.e., a pseudo instruction at the beginning of a *merge block* that defines a new variable and has exactly one input from every predecessor block. Figure 2.7b shows an example. This notation suffers from two problems. First, it seems like the inputs to the $\varphi$ are *live* in the merge block and therefore *interfere*. Second, one might assume that the $\varphi$-variable definition order is fixed. Both observations are not true. The inputs of a $\varphi$-function are *live* until the end of the predecessor block and all $\varphi$-variables are defined in parallel.

To make this more explicit, we use a notation that was, for example, described by Hack [2007, see Chapter 2.3.1]. As shown in Figure 2.7c, we split the $\varphi$s into two parts. In the merge block we have a $\varphi_{in}$ instruction, that defines all $\varphi$-variables atomically. At the end of every predecessor there is a $\varphi_{out}$ instruction that has as many inputs as there are $\varphi$-variables. This notation solves both problems. First, the definitions occur in parallel. Second, the pseudo usages in $\varphi_{out}$ keep the values *live* until the end of the block. In contrast to the conventional notation, the liveness analysis does not need modifications compared to a non-SSA variant.

The $\varphi_{in}/\varphi_{out}$ notation allows us to see $\varphi$s from a different perspective, namely as *parameters* of a basic block, a mental model that was also suggested by Appel [1998]. The variables defined by $\varphi_{in}$ of a block can be seen as the *formal parameters*[6] of a block. The values in $\varphi_{out}$ are the *actual parameters* when *"calling"* a block. Since this model conceptionally decouples the outgoing and

[6]See Cooper and Torczon [2011] for more information about *formal* and *actual* parameter.

incoming values, the renaming of variables is often not necessary. Therefore, we sometimes omit the newly introduced variables and use values directly in $\varphi_{out}$, i.e., we perform a kind of *copy propagation* [Aho et al., 2006]. Figure 2.7d depicts an example.

### 2.4.2 SSA destruction

Before emitting machine code, $\varphi$-instructions need to be deconstructed. This means, they are replaced by move instructions in the respective predecessor block. SSA destruction can be done explicitly, as for example described by Sreedhar et al. [1999]. However, register allocators that work on SSA form often perform SSA destruction as part of the data-flow resolution (e.g., the SSA-based linear scan allocator by Wimmer and Franz [2010]).

# Chapter 3

# The Graal Virtual Machine

We used GraalVM, a Java Virtual Machine, as a platform to experiment with our register allocation approach. GraalVM is a project done at Oracle Labs, in cooperation with academic collaborators, with the goal of developing a polyglot runtime based on the JVM. We implemented our register allocator in the Graal compiler (or simply Graal), the just-in-time compiler that is the core of the virtual machine. In addition to the compiler, GraalVM includes Truffle, a language implementation framework [Würthinger et al., 2017]. Truffle allows developers to create an execution system for a certain programming language by implementing an *abstract-syntax-tree* (AST) interpreter for that language. During interpretation, the AST specializes to the observed input (e.g., to the observed variable types). When the execution frequency of a method exceeds a certain threshold, the Graal compiler uses *partial evaluation* [Würthinger et al., 2012] to compile the specialized AST into high-performance machine code. In addition, Truffle features high-performance cross-language interoperability. The GraalVM can be executed in two different modes. The first mode is in the context of the *HotSpot VM*, the Java Virtual Machine (JVM) maintained by *Oracle*. The second deployment mode is via *Substrate VM* [Wimmer et al., 2017], which allows *ahead-of-time* (AOT) compilation of a restricted Java application—for example a Truffle interpreter and the Graal compiler—into a native image. Figure 3.1 gives an overview of the GraalVM ecosystem.

Since we evaluated our approach mainly with Java and Scala workloads on GraalVM on top of HotSpot VM, we will not go into the details of Truffle and Substrate VM, although they are interesting projects on there own. All the core components of GraalVM, including the implementation of our approach, are open source.[1] In addition, there is a pre-built *enterprise edition* bundle available from the GraalVM website.[2] It includes optional compiler optimizations, for example code duplication [Leopoldseder et al., 2018], that are not open source. We, however, evaluated our approach with the open source parts.

[1] `https://github.com/oracle/graal`
[2] `https://www.graalvm.org`

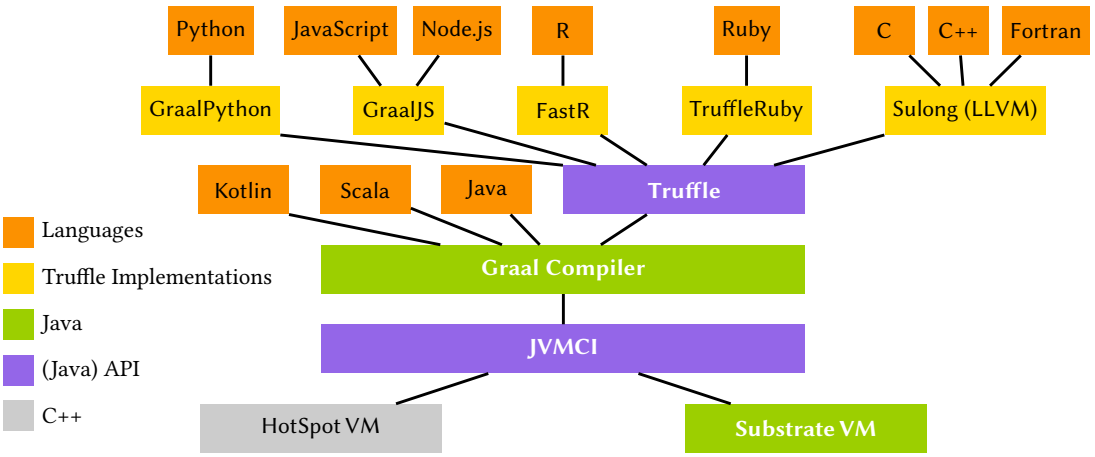



**Figure 3.1:** The GraalVM ecosystem

## 3.1 The Java Virtual Machine

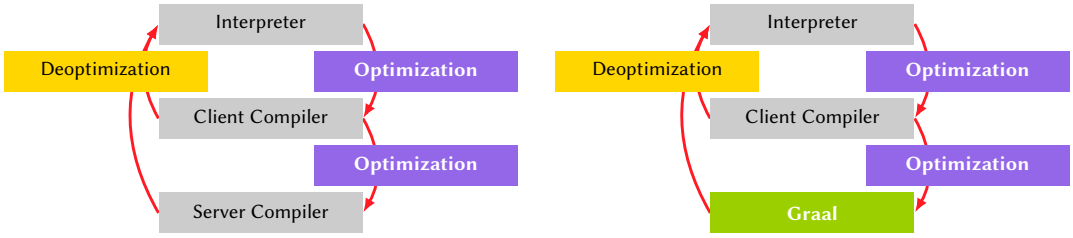
The Java Virtual Machine Specification [Lindholm et al., 2015] defines an abstract machine that executes Java *bytecodes* [Lindholm et al., 2015, Chapter 6]. Bytecodes are deployed in so-called *class files*, a format which is also specified in the JVM specification [Lindholm et al., 2015, Chapter 4]. Class files and bytecodes are machine-independent and can be executed on every host where there exists an implementation of the JVM. The JVM does not only deal with the execution of bytecodes but also offers support for threading, garbage collection and reflection.

Although the JVM is designed in close ties with the specification of the *Java programming language* [Gosling et al., 2015], the JVM can execute all languages that are compiled to bytecodes. Examples include Scala,[3] Clojure[4] and Kotlin,[5] which are ahead-of-time compiled to Java class files. Other languages, especially dynamically typed ones, generate bytecodes on-the-fly during execution of the program, for example JRuby[6] or Jython.[7]

## 3.2 The HotSpot VM

The HotSpot VM [Oracle Corporation, 2016] is an implementation of the JVM specification originally developed by *Sun Microsystems* and now maintained by *Oracle*. It executes bytecodes using an interpreter, or it just-in-time compiles them to native machine code. The bytecodes are first executed in the interpreter. The VM tracks the number of invocations and loop iterations

[3] http://www.scala-lang.org/
[4] https://clojure.org/
[5] https://kotlinlang.org/
[6] http://www.jruby.org/
[7] http://www.jython.org/




(a) HotSpot VM (b) GraalVM on HotSpot

**Figure 3.2:** Tiered-compilation on the JVM

of a method. Once a *threshold* is reached, the method is queued for JIT compilation.[8] We say the method is *hot*. The compiled code is orders of magnitude faster than the interpreter. Most methods are only executed rarely and will thus never be compiled. During the execution of bytecodes, the interpreter collects *profiling information* including branch probabilities or the types of receiver objects at a *virtual call*. The compiler exploits this knowledge about the run-time behavior of the program to optimize the machine code for the (expected) use case. Many of these optimizations *speculate* that the program will exhibit the same behavior in the future [Duboscq et al., 2013]. For instance, if the profile shows that a branch was never taken, the compiler might not produce code for the branch target. Another example are virtual call sites which might be *inlined* if they were always called with the same receiver types. Since the program might change its behavior, e.g., take an untaken branch or call a method with a different receiver type, the compiled code is not valid in such a case. To ensure the semantic correctness of the machine code, the compiler inserts checks to verify that the *assumptions* still hold. In case they do not, the compiled code triggers a so-called *deoptimization*, which means that execution is continued in the interpreter [Hölzle et al., 1992; Wimmer et al., 2017]. Since interpreted code has a different stack and register layout than compiled code, the VM has to create an *interpreter frame* and populate it with the contents of machine registers and stack slots.

The HotSpot VM includes two compilers, the *client compiler* (C1) [Kotzmann et al., 2008] and the *server compiler* (C2) [Paleczny et al., 2001]. The goal of the client compiler is to provide fast compilation speed and rapid start-up. The server compiler, in contrast, aims at good code quality at the cost of a higher compilation time. To combine the advantages of both, the HotSpot VM uses an approach called *tiered compilation* [Oracle Corporation, 2017], which utilizes both compilers. Execution starts in the interpreter, which collects profiling information. Hot methods are then compiled using the client compiler, which continues to collect profiling information. If the method execution count exceeds another threshold, the method is scheduled for compilation with the server compiler. Figure 3.2a illustrates the system. The client-compiled code with profiling is still a lot faster than the interpreter. Therefore, more information is collected before the method is processed by the server compiler, which can therefore produce better optimized code.

[8] The default threshold is 10000 [Oracle Corporation, 2018].

**JVM and Application Warmup** When the JVM starts executing an application, all methods are interpreted in the beginning. Eventually, the *hot* parts are compiled and executed via native machine code. That means the execution of the application gains speed after a while. We say the application *warms up*. For deterministic programs, especially benchmarks, it is often assumed that after the warmup phase the application will reach a *steady state* of *peak performance*. In reality, however, determining if an application has reached its *steady state* is hard or even impossible, as demonstrated by Barrett et al. [2017].

### 3.2.1 Graal on the HotSpot VM

In the *Graal-on-HotSpot* deployment model, the server compiler is replaced by the Graal compiler as the second-tier compiler, as depicted Figure 3.2b. The Graal compiler is itself written in Java, which enables a fast development cycle and allows us to use advanced debugging techniques such as *hot code replacement*.[9] It is implemented in a modular way so that its components, e.g. the register allocator, can be easily replaced with a different implementation. This makes it a practical environment for (dynamic) compiler research. The Graal compiler communicates with the HotSpot VM, which is implemented in C++, via the JVM Compiler Interface [JVMCI, 2014]. JVMCI is part of Java 9 and later releases, however, all results presented in this work are based on a JVMCI enabled development version of Java 8.[10,11]

#### A Note on Graal and Meta-circularity

The fact that Graal on the HotSpot VM is a *meta-circular* system—meaning that Graal is itself implemented in Java—has some interesting consequences.

Like any other Java code, the source code of the compiler is compiled to bytecodes which are then executed by the JVM. If the JVM decides that a method should be compiled with the second-tier compiler, Graal is invoked. Like other bytecodes, Graal is first executed by the interpreter. The hot parts of Graal are scheduled for compilation and might even be compiled by Graal itself. Therefore, Graal starts slow (interpreted) and gets faster over time (JIT compiled). But Graal also competes with the application for compiler threads and might delay the *warmup* of the application. The situation is two-fold. On the one hand, the application is what we are actually interested in, so we should compile it early. On the other hand, if Graal gets compiled the compilation of application code will be faster. The current trade-off is that compilation of Graal methods will stop after the first compiler tier. This means that Graal will only be compiled by the

[9] `https://wiki.eclipse.org/FAQ_What_is_hot_code_replace%3F`
[10] `http://hg.openjdk.java.net/graal/graal-jvmci-8/`
[11] `http://www.oracle.com/technetwork/oracle-labs/program-languages/downloads/index.html`

client compiler. Therefore, the Graal coding style favors patterns that are easy to optimize. For example, it prefers virtual calls over interface calls and avoids the use of *lambda expressions*[12] or *streams*.[13]

Another unfavorable effect of meta-circularity is *profile pollution*. Graal makes use of features from the Java standard library, for example, collections or sorting. Since profiling in HotSpot is not context sensitive, the profile of the shared code includes information from the application as well as about Graal. This might misguide the compiler and produce code that is not optimal for the application. For example, an application always calls `Arrays.sort(T[], Comparator<T>)` with the same comparator, Due to the profiling information the JIT compiler can inline the `compare()` method of that comparator. Unfortunately, Graal also uses this function, so the profile will show two receiver types and the compiler refuses to optimize the call.[14] However, Graal reduces the effect of profile pollution by using custom data structures.

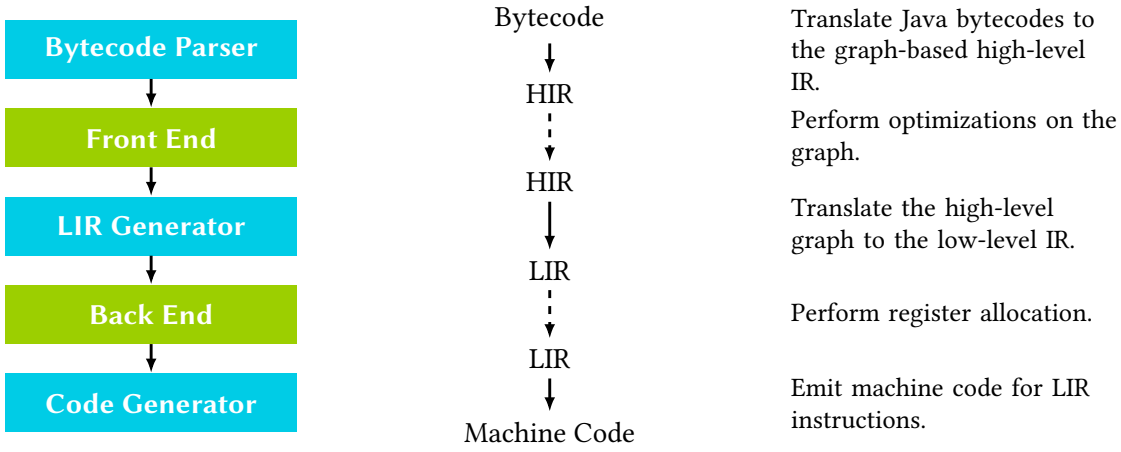
For both problems there are proper solutions. One would be to run Graal in a different JVM context with a separate compilation queue and distinct profiling information. Another solution would be to compile Graal ahead-of-time. Substrate VM already applies the second solution, i.e., it contains a pre-compiled version of Graal. *Project Metropolis* aspires towards the same for the HotSpot VM [OpenJDK, 2018].

## 3.3 The Graal Compiler

To translate Java bytecodes to machine code, the Graal compiler involves two *transformation* and three *translation* stages. During the process, the compilation unit is represented in four different shapes, as shown in Figure 3.3.

**Bytecode Parser** First, the *bytecode parser* translates the bytecodes into a *high-level intermediate representation* (HIR), which is graph-based [Duboscq et al., 2013; Click and Paleczny, 1995] and in static single assignment form (SSA, see Section 2.4). Many of the instructions, called *nodes* in HIR, are *floating*, which means that they are not bound to a specific basic block. Their *position* is only defined by their *inputs* (dependencies) and *users* (dependants). Figure 3.4 shows an example of the high-level IR. Although Java bytecode can describe irreducible programs, Graal

[12] `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`
[13] `http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html`
[14] Note that static methods such as `Arrays.sort()` are often inlined into the caller, where we might have precise receiver type knowledge and could optimize the comparator call after all.




(a) Phases (b) Program representations (c) Responsibility

**Figure 3.3:** Graal compiler pipeline

handles only reducible control flow (Section 2.2.2). This assumption simplifies all control-flow-sensitive phases. Since Java programs are always reducible [Appel and Palsberg, 2003, Chapter 18], this restriction is not an issue in practice.[15]

**Front End** The *front end*[16] performs a number of optimizations [Stadler et al., 2013; Prokopec et al., 2017], including *polymorphic inlining* [Deutsch and Schiffman, 1984], *partial escape analysis and scalar replacement* [Stadler et al., 2014], *loop unrolling* [Stadler et al., 2013], *global value numbering* [Click and Paleczny, 1995], *strength reduction*, *constant folding*, *branch* and *dead-code elimination* [Cooper and Torczon, 2011] as well as *speculative* JVM-specific optimizations [Duboscq, 2016]. In addition, the front end transforms the HIR from JVM-specific concepts (e.g., an instruction representing a `getfield` bytecode) to machine-level constructs (a `null`-check and a raw memory access). The *scheduler* is the final phase of the front end. All remaining floating nodes are assigned to basic blocks and the order of the nodes within the blocks is finalized. Refer to Click [1995, Chapter 6.3] for more details on this procedure.

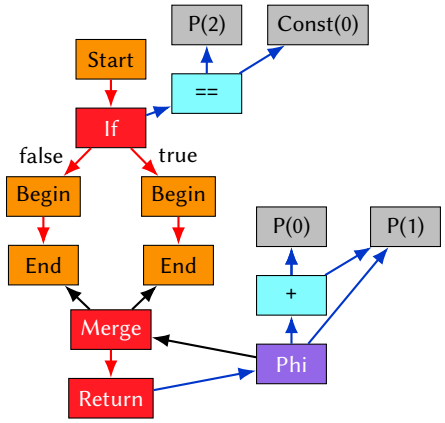
**LIR Generation** The scheduled graph is then translated to a low-level intermediate representation (LIR ) before entering the *back end*. The LIR uses a conventional CFG representation with basic blocks (see Section 2.2). Critical edges are split (see Definition 15). Every block is associated with a list of LIR instructions, which are close to the actual machine instructions.

[15] Note, however, that it is possible to create irreducible loops in Java bytecode.

[16] The use of the term *front end* might seem strange in this context since it is often associated with the component that deals with the programming language. Clang [2018] for example is the *C/C++* front end for the LLVM compiler [Lattner and Adve, 2004]. Following the definition of Cooper and Torczon [2011] "the front end focuses on understanding the source-language program." In the case of Graal, the source-language is *Java bytecode* and the front end deals with the semantics of bytecodes. Therefore, the term *front end* is indeed appropriate.

```
int foo(int value0, int value1,
        boolean cond) {
    final int result;
    if (cond) {
        result = value0 + value1;
    } else {
        result = value1;
    }
    return result;
}
```

(a) Source code




(b) HIR

The Java code snippet (left) and the corresponding high-level graph (right). Example borrowed from Duboscq et al. [2013]. A *downwards facing* edge (red) represents a *control-flow* successor of a node. For example, the *true* and the *false* successors of the If node. *Upwards facing* edges are *dependencies* and come in two forms: They are either *input* edges (blue), which represent data dependencies. For example, the + node calculates the sum of the two inputs, the parameter nodes P(0) and P(1). In addition, there are *association* edges (black). They represent non-data dependencies. For example, a Phi node is always attached to a Merge node. Duboscq et al. [2013] give more details on this IR.

**Figure 3.4:** Graal High-level Intermediate Representation (HIR)

Although they are specific to the target architecture, the back end phases are implemented in a machine-independent manner. LIR instructions may *define* (output) or *use* (input) operands. These operands can be *variables*, *registers*, virtual or physical *stack slots*, or *constants*. All operands are associated with a type, called *LIR kind*, which determines the register class that can hold the content. In addition, LIR kinds store *reference information*, that is, whether the operand is a pointer to a heap object. This is information needed for the garbage collector, in order to find all referenced objects. The *LIR generator* also features a simple DAG[17] matcher for common patterns. Nodes that are not processed by the matcher are translated to one or several LIR instructions. Figure 3.5 shows an example of the LIR.

**Back End** The main responsibility of the *back end* is register allocation. Before register allocation, the LIR still adheres to the SSA form, so every variable has a single definition. For $\varphi$s, the LIR uses the representation described in Section 2.4.1. The $\varphi_{in}$ variables are *defined* by the label instruction, which can be found at the beginning of every block. The $\varphi_{out}$ values are attached as *usages* of the unconditional jump instruction at the end of the predecessors of the merge block. For *fixed register constraints*, for example as required by calling conventions, the LIR instructions use register operands directly. These usages do not adhere to the *single definition property* of the SSA form. However, a fixed register is never live across a basic block boundary, so these requirements can be handled locally. The register allocator replaces variables with machine registers

[17] Directed Acyclic Graph.

```
B0 -> B1,B2
   [rsi|DWORD, rdx|DWORD, rcx|DWORD, rbp|QWORD] = LABEL
   v5|QWORD = MOVE rbp|QWORD
   v0|DWORD = MOVE rsi|DWORD
   v1|DWORD = MOVE rdx|DWORD
   v2|DWORD = MOVE rcx|DWORD
   TEST (x: v2|DWORD, y: v2|DWORD)
   BRANCH condition: =0 trueDest: B1 falseDest: B2
B2 <- B0 -> B3
   [] = LABEL numbPhis: 0
   v3|DWORD = ADD (x: v0|DWORD, y: v1|DWORD)
   JUMP ~[v3|DWORD] destination: B3
B3 <- B2,B1
   [v4|DWORD] = LABEL numbPhis: 1
   rax|DWORD = MOVE v4|DWORD
   RETURN (savedRbp: v5|QWORD, value: rax|DWORD)
B1 <- B0 -> B3
   [] = LABEL numbPhis: 0
   JUMP ~[v1|DWORD] destination: B3
```

(a) Before register allocation

```
B0 -> B3,B2
   [rsi|DWORD, rdx|DWORD, rcx|DWORD, rbp|QWORD] = LABEL
   TEST (x: rcx|DWORD, y: rcx|DWORD)
   BRANCH condition: =0 trueDest: B3 falseDest: B2
B2 <- B0 -> B3
   [] = LABEL numbPhis: 0
   rsi|DWORD = ADD (x: rsi|DWORD, y: rdx|DWORD)
   rdx|DWORD = MOVE rsi|DWORD // phi resolver
   JUMP ~[] destination: B3
B3 <- B2,B0
   [] = LABEL numbPhis: 1
   rax|DWORD = MOVE rdx|DWORD
   RETURN (savedRbp: rbp|QWORD, value: rax|DWORD)
```

(b) After register allocation

The low-level intermediate representation (LIR) of the `foo()` method from Figure 3.4 for AMD64. Representation as shown by the *Client Compiler Visualizer* [Wimmer, 2007]. Operands have the form `v1|DWORD`, where the first part is the variable or register name and the second the *kind*, i.e., the *width* and register class. On the left, the LIR *after LIR generation*. Note the representation of $\varphi_{in}/\varphi_{out}$ as operands of the `LABEL` and `JUMP` instructions in the blocks `B3`, `B2` and `B1`. The right figure shows the LIR *after register allocation*. The block `B1` was empty and has therefore been deleted. The $\varphi$ has been replaced by a move in block `B2`.

**Figure 3.5:** Graal Low-level Intermediate Representation (LIR)

or stack slots and also destructs the SSA form. Machine instructions in modern architectures can often directly address memory. Therefore, a LIR instruction differentiates between usages that *must have* a register and those that could also use a memory operand. The register allocator uses this information to reduce the register pressure. After register allocation, the back end performs simple peephole optimizations such as removing empty blocks (introduced by critical edges splitting) or eliminating redundant explicit `null`-checks. Figure 3.5b shows the LIR example after register allocation.

**Code Generation** *Code generation* is the last phase in the compiler pipeline. Every LIR instruction has an `emit()` method that writes machine instructions into a code buffer. Usually, a LIR instruction will emit a single machine instruction, although it could also produce more than one. The machine instructions are directly emitted to a byte array in their binary representation. In addition to the code array, the code generator collects meta-information needed by the virtual machine, for example, the frame layout for deoptimization or embedded object constants which might be moved by the garbage collector.

# Chapter 4

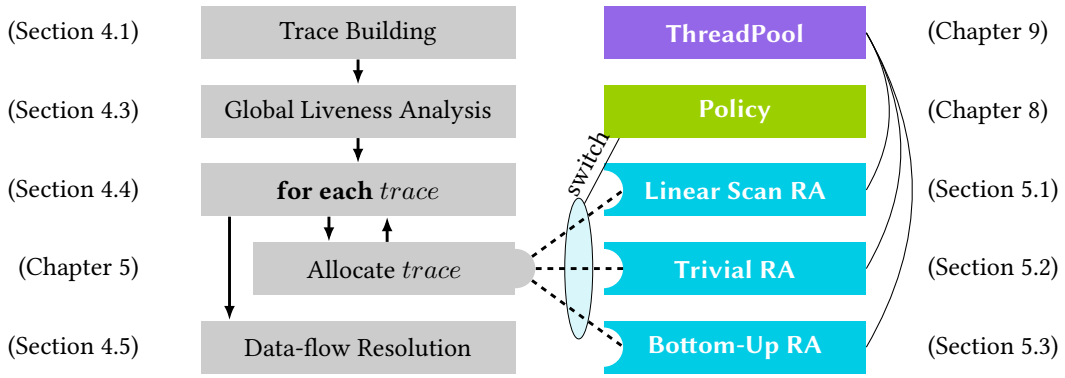
# Trace Register Allocation

Trace register allocation relinquishes the idea that register allocation needs to be done globally, i.e., for the whole compilation unit at once, using the same allocation algorithm. Instead, the idea is to focus on those parts of the compilation unit which are considered the most important. For those parts we invest compile time to find a good allocation. For the remaining parts of the compilation unit, *any* valid allocation is fine. To achieve this, we partition the control-flow graph into lists of sequentially executed basic blocks, based on their execution frequency. We call those lists *traces*. We allocate each of these traces independently, potentially using different allocation *strategies*, i.e., allocation algorithms. Due to their simple structure, traces are easy to allocate, so the strategies are simpler to implement than a global allocation approach. The allocation results of all traces are later combined to a valid global allocation.

**Why are they called *Traces*?** Many research directions in the field of programming language implementations utilize the term *trace*. Examples from our research group include compiling *traces* of executed bytecodes [Häubl and Mössenböck, 2011], *tracing* the execution of programs [Schatz and Prähofer, 2013], or tracking memory allocations via memory *traces* [Lengauer et al., 2016]. Although some of these works relate to our approach, we use the term *trace* in the legacy of *trace scheduling*, for example by Fisher [1981], Ellis [1985] or Lowney et al. [1993], which operates on the same structure as our approach.

The remainder of this section gives an overview of the main components of our trace register allocation approach. Figure 4.1 shows the components of the trace register allocation framework. We will first give a brief overview of the whole picture before going in to the details in the following sections and chapters.

**Trace Building** The *trace builder* takes a control-flow graph and partitions it into distinct traces. See Figure 4.2 for an example. It uses profiling information gathered by the VM to find long *hot* traces with high execution frequencies.




The left-hand side (gray boxes) shows the main components of the trace register allocation framework, i.e., the *trace builder*, the *global liveness analysis*, the *allocation loop* and the *data-flow resolution*. They work on a global scope, i.e., on the whole compilation unit. The *allocation strategies* on the bottom right (blue) find an allocation result for a single trace. For every trace, the framework can switch between different strategies. A *policy*, depicted above the strategies (green), decides which strategy should be used for a given trace. Traces can be processed in parallel, e.g., by using a *thread pool* as visualized on the top right (purple).

**Figure 4.1:** Trace register allocation overview

**Global Liveness Analysis** To enable independent processing of traces, we need liveness information of variables at trace boundaries. Therefore, a global liveness analysis is required. This analysis also operates on the control-flow graph and calculates the set of *live* variables at every inter-trace edge.

**Allocate Traces** Using the information from the trace builder and the global liveness analysis, the *registers* are *allocated*. For every trace we are free to choose one of several *allocation strategies*, based on the properties of the trace. Although we could process the traces in arbitrary order, we allocate important traces first. We exploit information about already processed traces to reduce the amount of data-flow resolution at trace boundaries.

**Data-flow Resolution** Since the location of a variable might be different across an inter-trace edge, *data-flow resolution* is needed for these edges. That phase will insert move operations from one location to another whenever needed.

## 4.1 Trace Building

The aim of a trace building algorithm is to find traces in the control-flow graph. A trace is a sequence of basic blocks connected with edges. Traces have interesting properties that we can exploit in a register allocator. Let us define a *trace* more formally and introduce some notations to make reasoning simpler.

**Definition 30** (Trace)**.** A *trace* $T$ is a sequence of basic blocks $\langle b_0, \ldots, b_n \rangle$. The *trace length* $|T|$ denotes the number of blocks in a trace. We use the notation $T[i]$ to refer to the $i^{th}$ block in a trace with $i$ ranging from 0 to $|T| - 1$. We call the first block $T[0]$ in a trace the *trace head*. There is an edge between two consecutive blocks in a trace, $T[i] \in succ(T[i-1])$ for $i > 0$. Therefore, the blocks might be executed sequentially. We also define that the edge $(T[i-1], T[i])$ must not be a back edge. This implies that all blocks of a trace $T$ are distinct, so $T[i] \neq T[j]$ for $i \neq j$.

The set of *Traces* is a *partition* of the blocks of a CFG. Every block is in exactly one trace, so $T \cap T' = \emptyset$ for $T, T' \in Traces$ and $T \neq T'$. We write $T = traceOf(b)$ if $T$ is the trace that contains block $b$ and $b \in T$ if $T = traceOf(b)$. Traces are non-empty, i.e., $|T| > 0$ for $T \in Traces$.

We call an edge $(b_a, b_b)$ between two blocks of the same trace an *intra-trace edge*, i.e., if $b_a, b_b \in T$. If not, i.e., $traceOf(b_a) \neq traceOf(b_b)$, it is an *inter-trace edge*. Any block in a trace can be the source or the target of an inter-trace edge. In other words, a trace can have multiple *entries* and *exits*.

There are many possible ways of finding trace partitions in the CFG.[1] In the simplest case, every block can be treated as a separate trace. In this case trace register allocation would be equivalent to local register allocation. However, longer traces offer more opportunities for optimization, for example, by moving spill code out of loops, and thus to improve the quality of the register allocation. Therefore, we need more sophisticated trace building strategies. In the following, we propose two trace building strategies that utilize the run-time feedback provided by the execution environment.

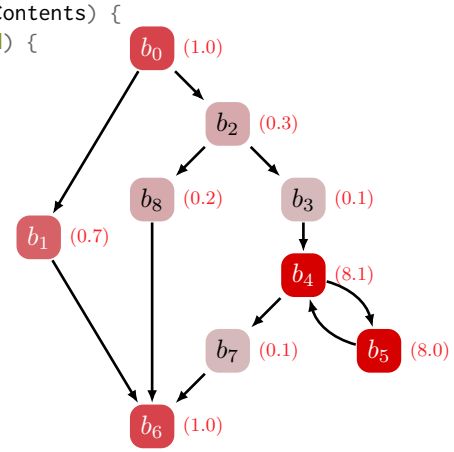
### 4.1.1 Unidirectional Trace Building

The unidirectional trace builder tries to minimize the trace exit probability. The algorithm, as shown in Algorithm 1, starts a new trace by selecting a block where all predecessors are already part of some other trace. If there are multiple candidates, we choose the block with the highest execution frequency. At the beginning, the only eligible block is the method entry. The block is selected as the header of the new trace. Next, we select the block with highest execution frequency from the set of successors that are not yet in any trace and add it to the current trace. We continue adding blocks in this way as long as possible. If there are no more successors the trace is complete and we proceed with the next trace. This is repeated until all blocks are assigned to a trace. Figure 4.2c depicts the result of the algorithm for the code in Figure 4.2a.

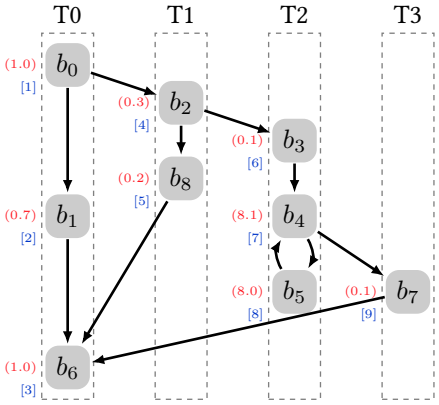
[1] The number of possible partitions of a set is called the *Bell number* [Flajolet and Sedgewick, 2009]. However, only some of them fulfill all the properties we require for a *trace partition*.

```
long *allocateArray(int length, boolean fillContents) {
/*B0*/  if (TLAB.top + length + 1 <= TLAB.end) {
/*B1*/      result = TLAB.top;
          TLAB.top += length + 1;
        } else {
/*B2*/      result = heapAlloc(length + 1);
          if (fillContents) {
/*B3*/          long *ptr = result + length;
/*B4*/          while (result < ptr) {
/*B5*/              *ptr = 0; ptr--;
                } /* critical edge B7*/
            } else { /* critical edge B8*/ }
        }
/*B6*/  result[0] = length;
        return result;
}
```

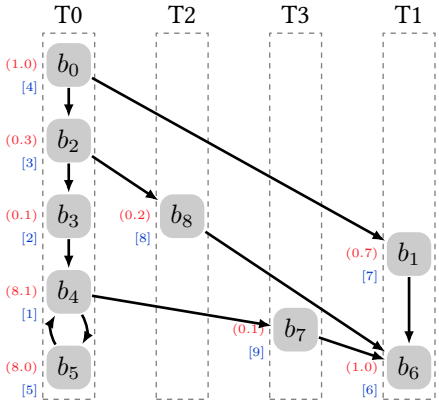
(a) C-like source code




(b) Control-flow graph




(c) Unidirectional trace builder




(d) Bidirectional trace builder

Source code, control-flow graph and trace-building results of `allocateArray`. The values in parentheses (e.g. (0.8)) denote relative execution frequencies. The numbers in square brackets (e.g. [2]) denote the order in which the trace builder processed the block.

**Figure 4.2:** Trace-building example for `allocateArray`

Although the algorithm is simple, it is not trivial to see that it terminates under all circumstances. Let us therefore analyze it in more depth.

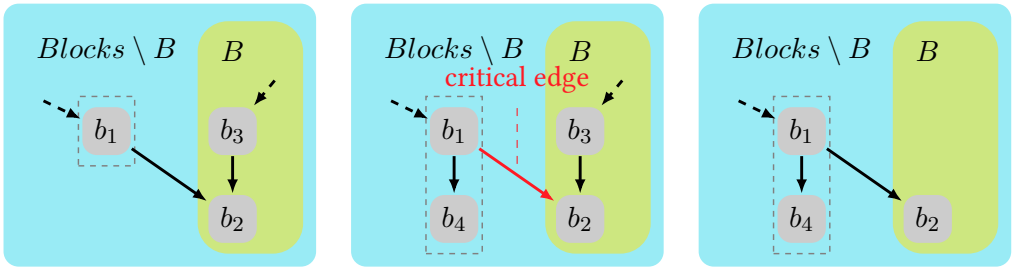
**Theorem 4.** The unidirectional trace builder described in Algorithm 1 terminates for every (*reducible*) CFG without *critical edges*.

*Proof.* Let us first discuss the *termination condition* of Algorithm 1. The inner loop on line 9 continues until the *eligible* set $E$ is empty. In the loop, $E$ is redefined on line 13. It is easy to see that its size is bound by $B$. The size of $B$, however, is reduced by *one* in every iteration of the inner loop due to line 12. Therefore, the inner loop will eventually terminate.

**Algorithm 1** Pseudocode for the Unidirectional Trace Builder

```
1:  procedure UnidirectionalTraceBuilder(CFG)
2:      (Blocks, Edges) ← CFG
3:      Traces ← {}
4:      B ← Blocks                                   ▷ Block not yet in a trace
5:      while |B| > 0 do                             ▷ There are blocks not yet in a trace
6:          T ← ⟨⟩                                   ▷ New trace
7:          E ← {b ∈ B | pred(b) ∩ B = ∅}
8:                                                   ▷ Eligible blocks where all predecessors are in a trace
9:          while E ≠ ∅ do
10:             b ← b_max ∈ E where freq(b_max) is maximal   ▷ Block with highest frequency
11:             T ← T + ⟨b⟩                          ▷ Add block to trace
12:             B ← B \ b                            ▷ Block in a trace
13:             E ← succ(b) ∩ B                      ▷ Eligible blocks, successor not in a trace
14:         end while
15:         Traces ← Traces ∪ T                      ▷ Add T to the set of traces
16:     end while
17:     return Traces
18: end procedure
```

How about the outer loop? Due to the condition on line 5, the loop terminates if all blocks are in a trace, so that $B$ is empty. For termination we need to ensure that the size of $B$ is reduced in every iteration. We have already seen that the inner loop, if entered, will always terminate *and* reduce $B$ in every iteration. It remains to show that we will enter the inner loop at least once in every outer-loop iteration. In other words, we have to show that the set $\{b \in B \mid pred(b) \cap B = \emptyset\}$ on line 7 is never empty. In the first iteration of the outer loop all blocks are in $B$, so $pred(b) \cap B$ will be non-empty for every block unless $pred(b)$ is empty. Per definition, the only block without predecessors is the *method entry* block $b_{entry}$. This means that we will find at least one trace containing $b_{entry}$. In the general case, $B$ is a proper subset of the blocks ($B \subset Blocks$). So some blocks are already in a trace ($Blocks \setminus B$) and some are not ($B$). Let us now assume that $E$ on line 7 is empty. Since we know that $b_{entry} \notin B$ and all blocks are per definition reachable by $b_{entry}$, there must be an *inter-trace edge* from a block $b_1 \notin B$ to a block $b_2 \in B$. Also, for $E$ to be empty, $b_2$ must have another predecessor $b_3 \in B$. See Figure 4.3a for an illustration. Note, that the building algorithm is *greedy*, which means that it will extend a trace as long as possible. If block $b_2$ would be the single successor of $b_1$, it would have been added to the trace due to the construction of the eligible set on line 13. Therefore, since $b_1 \notin B$, there must be another block $b_4 \notin B$ and an edge $(b_1, b_4)$. However, as we can see in Figure 4.3b, this would mean that the edge $(b_1, b_2)$ is *critical* which would violate our assumption that there are no such edges in $CFG$. Since $b_4$ must exist, $b_2$ can have only a single predecessor $b_1$. More specifically, $b_3$ cannot exist as shown in Figure 4.3c. Thus, no predecessor of $b_2$ is in $B$ and the set $E$ is not empty (it includes at least $b_2$) which violates our assumption. □




(a) Non-greedy (b) Greedy with Critical Edge (c) Conclusion

Attempt to construct a non-terminating example for Algorithm 1. $Blocks$ are all block of the $CFG$. The blocks in $B$ are not yet assigned to a trace. Due to the greedy nature of the algorithm the situation in sub-figure (a) cannot occur since $b_2$ would be appended to the trace containing $b_1$. So there must be a block $b_4$ as shown in sub-figure b. This, however, violates our assumption that there are no *critical edges*. Thus, $b_3$ cannot exist c and $b_2$ is *eligible*.

**Figure 4.3:** Termination of the unidirectional trace building algorithm

### 4.1.2 Bidirectional Trace Building

The bidirectional trace builder creates traces in decreasing order of their maximum block execution frequency. Algorithm 2 outlines the idea in pseudocode. It first selects the block with the highest execution frequency from all blocks that are not yet part of a trace. From this initial block, it first *grows* the trace upwards. Among all predecessors from the candidate set we select the one with the highest execution frequency and prepend it to the trace. Note that we exclude predecessors that are the source of a loop back-edge. This means that after we processed a loop header we always continue with the block entering the loop, never with the *loop end* block. Once there is no candidate left we start the downwards pass, again starting at the initial block. We proceed in a way that is similar to the unidirectional trace builder. Bidirectional trace building has already been described by Ellis [1985] and Lowney et al. [1993]. Figure 4.2d shows the traces formed by the bidirectional trace builder. The example illustrates that both strategies potentially lead to different results.

Since the bidirectional trace builder imposes less restrictions for starting a new trace, the termination of the algorithm is easier to see.

**Theorem 5.** The bidirectional trace builder (Algorithm 2) terminates for every CFG that is *reducible* and has no *critical edge*.

*Proof.* Both inner loops of Algorithm 2 terminate if the respective *eligible* set $E$ becomes empty. Both sets are bound by $B$ (line 11 and line 20). In every inner loop, $B$ is decreased by one so the inner loops terminate eventually. The outer loop terminates if $B$ is empty. Since we pick a new $s \in B$ for every iteration and remove $s$ from $B$ in line 8 the outer loop is also guaranteed to terminate. □

**Algorithm 2** Pseudocode for the Bidirectional Trace Builder

```
1:  procedure BidirectionalTraceBuilder(CFG)
2:      (Blocks, Edges) ← CFG
3:      Traces ← {}
4:      B ← Blocks                                          ▷ Block not yet in a trace
5:      while |B| > 0 do                                    ▷ There are blocks not yet in a trace
6:          s ← b_max ∈ B where freq(b_max) is maximal      ▷ Block with highest frequency
7:          T ← ⟨s⟩                                         ▷ New trace with s
8:          B ← B \ s                                       ▷ Block s in a trace
9:          b ← s                                           ▷ Start with block s
10:         loop                                            ▷ Go upwards
11:             E ← {b' ∈ (pred(b) ∩ B) | (b', b) ∈ Edges not a backedge}
12:                 ▷ Eligible blocks, predecessors that do not form a loop and are not in a trace
13:             if E = ∅ break                              ▷ No more eligible blocks, leave loop
14:             b ← b_max ∈ E where freq(b_max) is maximal  ▷ Block with highest frequency
15:             T ← ⟨b⟩ + T                                 ▷ Prepend block to trace
16:             B ← B \ b                                   ▷ Block in a trace
17:         end loop
18:         b ← s                                           ▷ Reset current block to s
19:         loop                                            ▷ Go downwards
20:             E ← succ(b) ∩ B                             ▷ Eligible blocks, successor not in a trace
21:             if E = ∅ break                              ▷ No more eligible blocks, leave loop
22:             b ← b_max ∈ E where freq(b_max) is maximal  ▷ Block with highest frequency
23:             T ← T + ⟨b⟩                                 ▷ Append block to trace
24:             B ← B \ b                                   ▷ Block in a trace
25:         end loop
26:         Traces ← Traces ∪ T                             ▷ Add T to the set of traces
27:     end while
28:     return Traces
29: end procedure
```

**Discussion**

Figure 4.2 shows that the traces formed by the two trace building algorithms may differ. While the unidirectional algorithm favors the execution paths that are more likely, for example the common case $b_0$, $b_1$ and $b_6$, the bidirectional trace builder creates traces where most time is spent. In the `allocateArray` example, this is the *initialize loop* of the *heap allocation* case. Both methods lead to interesting results. Since our goal is to focus on the common case, the unidirectional trace builder is our algorithm of choice. We evaluated both approaches and came to the conclusion that the unidirectional trace builder is not only simpler but also achieves slightly better results than its bidirectional counterpart (see Figure 7.12).

## 4.2 Trace Properties

Although it seems intuitive that register allocation for a trace is simpler than for a general control-flow graph, traces have properties that are not self-evident. Some originate from the general definition of traces (Definition 30), others are due to our trace building strategies. Many of these properties can be exploited by a register allocation strategy to simplify its algorithmic complexity. In the following, we summarize these observations. Keep in mind that we assume that the CFG is reducible and does not contain critical edges.

**Definition 31** (Greedy Trace Builder)**.** A trace builder is upwards (downwards) greedy if it prepends (appends) blocks to a trace until there is no more predecessor (successor) that is not already in a trace.

**Theorem 6.** The unidirectional and bidirectional trace builders are *greedy*.

*Proof: Unidirectional.* This directly follows from the calculation of the eligible sets on line 7 and line 13 of Algorithm 1. □

*Proof: Bidirectional.* This directly follows from the calculation of the eligible sets on line 11 and line 20 of Algorithm 2. □

### 4.2.1 Greedy Trace Properties

The following properties hold for all greedy trace builders.

**Theorem 7** (Start Trace)**.** The *first* trace of a *greedy trace builder* always contains the start block $b_{entry}$ of the CFG.

*Proof.* Predecessors are prepended as long as they are not the source of a loop back edge. Since all blocks are reachable from the start block $b_{entry}$ the algorithm will eventually prepend $b_{entry}$ and stop the upwards pass. □

**Theorem 8** (Non-loop-header Trace Head)**.** A *loop header* $b_h \in L$ of a loop $L$ in a trace $T$ is always preceded by a block that is not in the loop, the loop entry. In other words, a loop header is never a trace head, or if $b_h = T[i]$ then $i > 0$ and $T[i-1] \notin L$.

*Proof.* By definition of a trace (Definition 30), the predecessor of a loop header $b_h$ in trace T cannot be a loop end. From Theorem 3 we know that there is at least one loop entry $b_e$. Due to *upwards greediness*, $b_e$ would be added to T if not already in a trace. However, due to *downward greediness* and the fact that $b_e$ has only $b_h$ as its successor (Corollary 1), $b_h$ would have been appended to $traceOf(b_e)$. Therefore, $b_e$ and $b_h$ are in the same trace, next to each other. □

**Theorem 9** (Single-Predecessor Trace Head)**.** If the trace builder is *greedy*, every trace head has no or a single predecessor.

*Proof.* We already that know that a loop header cannot start a trace (Theorem 8). Let us assume there is a trace head $b_h$ of trace T with more than one (non-back-edge) predecessor. Since the trace builder is *upwards greedy*, all predecessors of $b_h$ must be already in traces before T is built, otherwise they would be prepended to T. Since the trace builder is *downwards greedy*, a trace containing a predecessor $b_p$ of $b_h$ would have added $b_h$ to itself, unless $b_p$ has another successor. However, if $b_p$ has more than one successors (and we assumed $b_h$ has more than one predecessors) the edge $(b_p, b_h)$ would be *critical*. Therefore, either $b_h$ has only one successor or $b_h$ is not a trace header. □

See also Figure 4.3 for a visualization of this argument.

### 4.2.2 Dominance Properties of Traces

**Theorem 10** (Trace Block Dominance)**.** Blocks in a trace are ordered according to their dominance. This means that if two blocks $T[i], T[j]$ are in the same trace T and they are in a strict dominance relation $T[i] \in sdom(T[j])$, the dominating one is comes earlier in the trace ($i < j$).

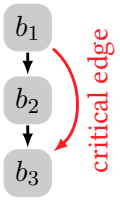



**Figure 4.4:** Non-sequential forward edges are critical edge

*Proof.* By contradiction. Let $T[i]$, $T[j]$, $i < j$ and $T[j] \in sdom(T[i])$. From Theorem 1 we know that every path from a block to its strict dominator contains a back edge. So the path $\langle T[i], T[i+1], \ldots, T[j] \rangle$, which is a subset of the trace $T$ contains a back edge which is prohibited by the definition of a trace (Definition 30). □

**Theorem 11** (Consecutive Intra-trace Edges)**.** If a control-flow graph does not contain critical edges, an intra-trace edge either connects two blocks that are next to each other in the trace, or it is a back edge.

*Proof.* By contradiction. Due to Definition 30 we know that every block in a trace $T[i]$, with exception of the last one, has at least one successor, namely the next block in the trace $T[i+1]$. Analogously, every block $T[i]$, that is not the trace head, has at least one predecessor, namely the previous block in the trace $T[i-1]$. Without loss of generality we assume that the trace consists of three blocks $\langle b_1, b_2, b_3 \rangle$. Figure 4.4 illustrates such a trace. Now, let there be a forward edge $(b_1, b_3)$. However, this means that the source block $b_1$ has two successor, namely $b_2$ and $b_3$, and the target block $b_3$ has more than one predecessor, namely $b_1$ and $b_2$. Therefore, the edge $(b_1, b_3)$ is a *critical edge*, which contradicts our assumption that there are no critical edges. □

**Corollary 2** (Loop End/Trace End)**.** If a control-flow graph does not contain critical edges, a back edge can only occur at the end of a trace.

*Proof.* From Corollary 1 we know that a loop end has only one successor, the loop header. However, by Definition 30, the trace builder does not follow a back edge. Therefore, there is no eligible successor and the trace must end at a loop end. □

**Theorem 12** (Trace Head Dominator)**.** Let $b_i$ be a block of trace T. Every dominator $b_d \in dom(b_i)$ is either in T or $b_d$ also dominates the trace head $b_h$ of T.

*Proof.* Assume that $b_d \in dom(b_i)$, $b_d \notin T$ and $b_d \notin dom(b_h)$ where $b_h$ is the trace head. This would mean that there is a path $\langle b_{entry}, \ldots, b_h \rangle$ which does not include $b_d$. Since $b_h \rightsquigarrow b_i$, there would also have to be a path $\langle b_{entry}, \ldots, b_i \rangle$ that does not include $b_d$. Therefore, $b_d$ cannot dominate $b_i$. □

**Theorem 13** (Trace Head Liveness)**.** Assuming SSA form, a value which is *live* at a certain point in a trace T is either defined in T or *live* at the beginning of the trace head of T.

*Proof.* Due to the *dominance property* (Property 2) of the SSA form we know that a value $v$ that is used in block $b_i$ is defined in a dominator $b_d$ of $b_i$. If $v$ is not defined in T then there is a path from the definition in $b_d$ to the usage that goes through the trace head $b_h$. Therefore, $v$ must be live at the beginning $b_h$. □

**Theorem 14** (No Lifetime Holes in Traces)**.** The lifetime interval for a variable in a trace does not contain lifetime holes.

*Proof.* By Definition 27, a lifetime hole can only occur if there are non-consecutive blocks in the linearized form of the CFG. By Definition 30, this does not hold for a trace. □

**Definition 32** (Interval Graph)**.** Following Brisk and Sarrafzadeh [2007, Definition 1], which originates from Lekkeikerker and Boland [1962], a graph is an *interval graph* if it is the interference graph of some set of *live ranges*.

**Corollary 3** (The Interference Graph of a Trace is an Interval Graph)**.** By the definition of an interference graph and Theorem 14, the interference graph of a trace is an interval graph.

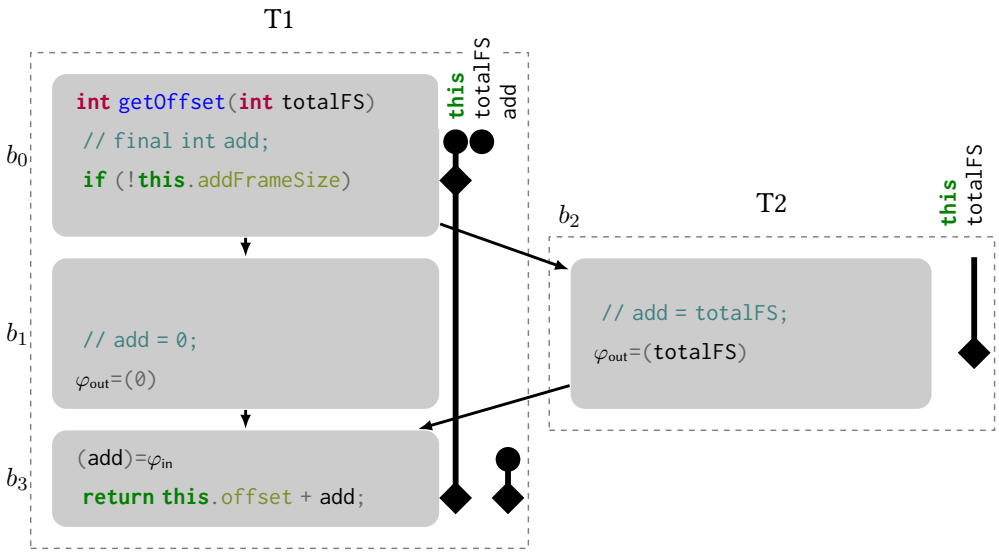
**Theorem 15** (Trace Domination)**.** Assume that a *greedy* trace builder builds traces in the order $T_0, T_1, \ldots, T_n$. Then we know that for a block $b$ in trace $T_k$ all dominators $dom(b)$ of $b$ are in traces $T_0, \ldots, T_k$.

*Proof.* Due to *upwards greediness*, the single predecessor $b_p$ (Theorem 9) of trace header $b_h$ of a trace $T_k$ is already in a trace $T_0, \ldots, T_{k-1}$. Since $b_{entry} \in T_0$ (Theorem 7) there is a path $\langle b_{entry}, \ldots, b_p, b_h \rangle$ which contains all dominators of $b_h$. So, due to Theorem 12, all dominators of all blocks in $T_k$ are either in $T_k$ or in $dom(b_h)$. □

## 4.3 Global Liveness Analysis

A core feature of our trace register allocation approach is that traces can be processed independently by the allocator, potentially with different strategies. However, a trace in general does not have enough information about the liveness of variables.

Let's have a look at Figure 4.5. There is no usage of `totalFS` in T1, so we could conclude that we do not need a register for storing its value. In trace T2, on the other hand, it seems that only `totalFS` is *live*. However, when looking at the whole method, it becomes clear, that this is




Example based on the code in Listing 2.

**Figure 4.5:** Traces without global liveness information

wrong. In T1, `totalFS` is live until the end of $b_0$, as shown in Figure 4.6. Additionally, **`this`** is live throughout trace T2, although it is not used there at all. So we need to exhibit *global liveness information* to the traces in order to perform register allocation properly.

Different register allocation algorithms use different data structures for modeling *liveness*, for example *intervals* in linear scan or an *interference graph* in graph coloring. Since we want to make the framework as flexible as possible, we do not force a *strategy* to use a certain model of liveness. Therefore, we maintain information about liveness at trace boundaries in a generic form. Every block in a trace has two sets, $live_{in}$ and $live_{out}$: $live_{out}$ stores all variables that are *live* at the end of the block, and $live_{in}$ stores those that are live at the beginning of the block. See Figure 4.6 for an example. The set $live_{out}$ can be seen as a *pseudo usage* of all variables in this set. It keeps those variables alive in case they are needed in other traces that branch off from this blocks. The set $live_{in}$ at the header of a trace is a *pseudo definition* of all variables in this.

### 4.3.1 Liveness Analysis

To compute these sets, we perform a *liveness analysis*. In the first prototype we used a standard *iterative data-flow analysis* [Cooper and Torczon, 2011, Chapter 9.2.2]. However, since the variables in LIR are in SSA form, we can exploit the *dominance* and *single definition* properties. We thus follow an approach described by Wimmer and Franz [2010] for SSA-based linear scan register allocation. The analysis is done in a single backwards iteration over the blocks in reverse postorder (see Definition 22). We maintain a set of live variables at block boundaries which we

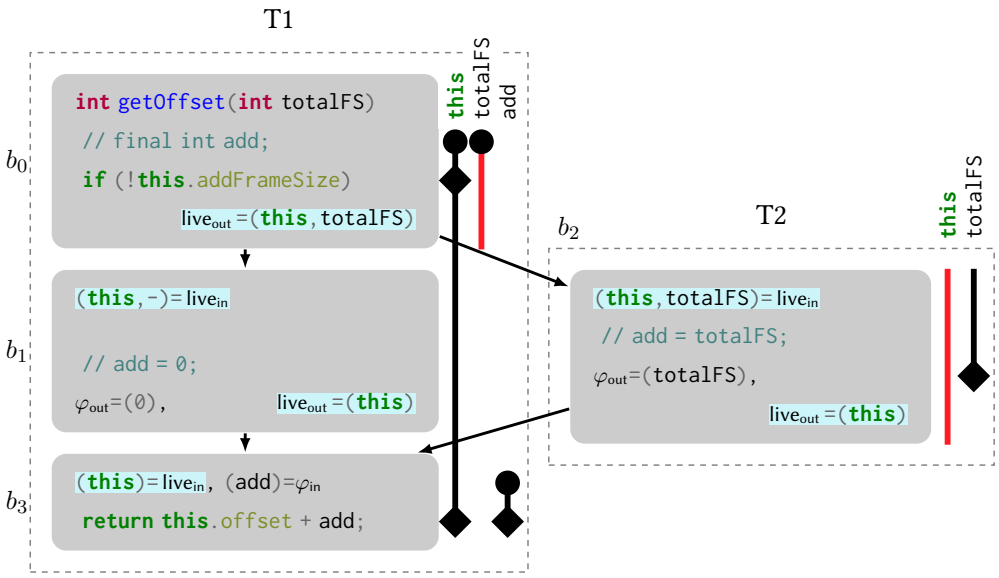


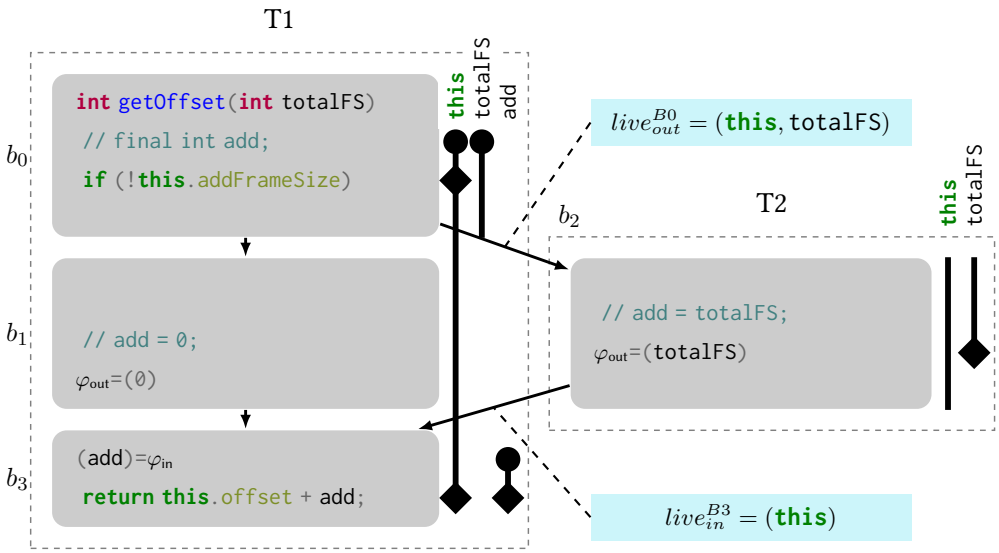

**Figure 4.6:** Traces with global liveness information

later use for the $live_{in}$ and $live_{out}$ sets. For loop headers, the live sets in the loop need to be updated, as proposed by Wimmer and Franz [2010]. Nevertheless, this approach offers significant speedup over the iterative approach.

### 4.3.2 Representation of Global Liveness

The most convenient way to represent liveness information is to add pseudo usages and definitions directly to the LIR, as shown in Figure 4.6. The representation is similar to the notion of $\varphi$s. The advantage is that phases can process the pseudo usages just as any other usage without special handling. Pseudo usages will be picked up by (trace-local) liveness analysis and the register allocator will assign locations to them. These locations are then used by the global data-flow resolution (see Section 4.5). The downside is that the *in-LIR* variant poses overhead in terms of space, because the $live_{in}$ and $live_{out}$ sets at both ends of an edge are redundant, and in terms of time, since the iteration of LIR instructions gets slower due to the increased number of operand references. In addition, it increases the number of indirections, since all operands in LIR are objects (of class `Variable` in this case), although a primitive integer value would be sufficient for encoding variables.

Therefore, we decided to use an external data structure to capture only the required information. Our first observation was, that for every control-flow *merge*, the liveness information is the same at the end of all predecessors. This is only true because there are no critical edges. Also note that $\varphi$s are still represented in the LIR. Therefore, it is sufficient to store only $live_{in}$ at the merge, and use this set also for all predecessors. We can use a similar optimization for control-flow




The set $live_{out}$ is stored at the end of split blocks and the set $live_{in}$ at the beginning of merge blocks.

**Figure 4.7:** Representation of liveness

*splits*, although a variable that is live at the end of a branching block might not be needed in all successors. The edge $(b_0, b_1)$ in T1 of Figure 4.6 demonstrates the situation. The variable `totalFS` is live at the end of $b_0$ but not at the beginning of $b_2$. However, this is usually not a problem. During trace-local liveness analysis—assuming that we do it via a backwards pass—we would see a *pseudo* definition without usage, which we can simply ignore. Thus, it is sufficient to store $live_{out}$ at the end of splitting blocks, and $live_{in}$ at the beginning of merge blocks, which reduces the memory requirement to at least a third.[2] Figure 4.7 shows the improved global liveness information for the previous example.

## 4.4 Allocating Registers

The *allocation step* is the core of the framework and the basis for its flexibility. Essentially, as long as there is an unprocessed trace, the framework selects one and allocates registers for it. *How* registers are allocated is not relevant to the framework. We implemented three different allocation *strategies* for different purposes. The details of them are described in Chapter 5. For every trace, the framework can use a different strategy. The decision *which* strategy is most suited is based on heuristics, which are discussed in Chapter 8. Note that traces could be processed in

[2] Also, we can use an `int[]` instead of a `Variable[]` array. This means that in addition to removing one indirection, only half of the memory is needed assuming that `int` requires 32 bits and a reference 64 bits.

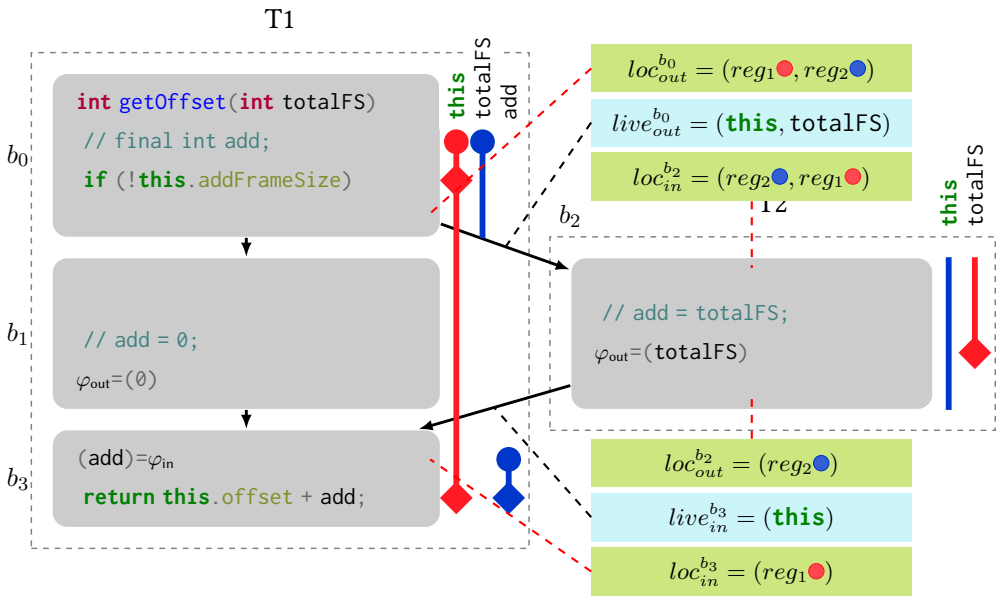


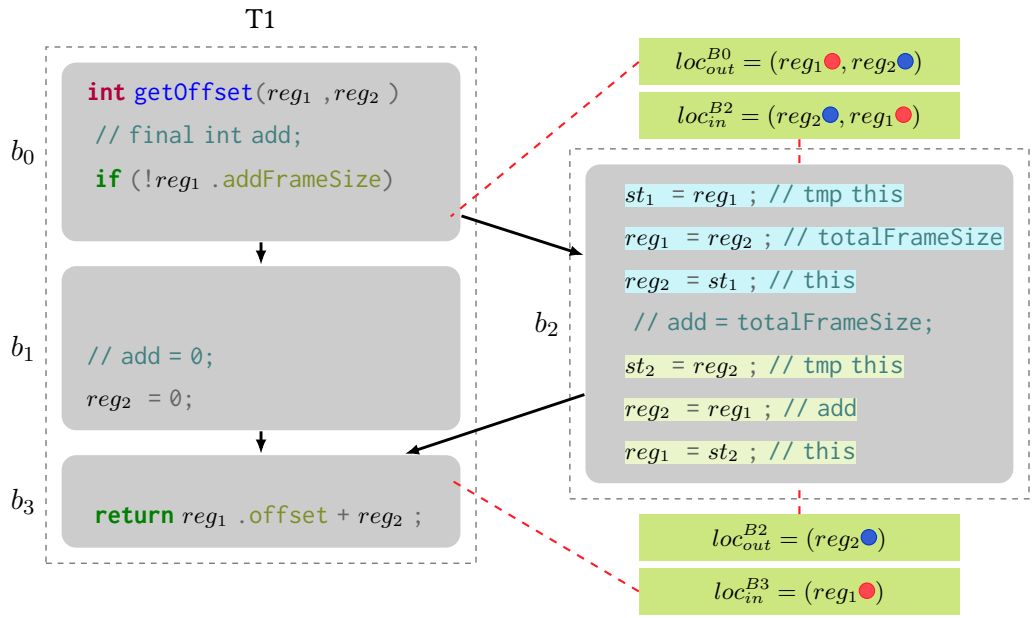

**Figure 4.8:** Representation of the $loc_{in}/loc_{out}$ sets

arbitrary order, even in parallel as demonstrated in Chapter 9. However, traces that are processed later can exploit information about already processed traces. Therefore, traces are processed in order of their *importance*.

In addition to allocating registers in the trace, the allocation strategy needs to record the *location* of variables at inter-trace edges. This information is stored in $loc_{in}$ and $loc_{out}$ sets. In contrast to the $live_{in}/live_{out}$ sets, the location sets are needed for both ends of an edge, since variables might be in different locations in different traces. Figure 4.8 shows the location sets for the `getOffset` example.

## 4.5 Global Data-flow Resolution

Since traces are processed independently, the location of a variable might differ at the two ends of an inter-trace edge. See the edge $(b_0, b_2)$ in Figure 4.8, for example. In $b_0$ of T1, `this` is stored in $reg_1$. However, in $b_2$ the value is in $reg_2$. Therefore, when crossing the edge, we need to adjust the locations to get a valid allocation. To do so, we need to insert moves which are executed when following the edge. For our example this means that we have to insert the moves at the beginning of $b_2$. This *global data-flow resolution* is done for every inter-trace edge. Figure 4.9 shows the example with the inserted move instructions.




Result of data-flow resolution for the allocation of `addOffset` in Figure 4.8. There are two inter-trace edges that need data-flow resolution, namely $(b_0, b_2)$ and $(b_2, b_3)$. For the first edge we need to insert moves from $reg_1$ to $reg_2$ for **this** and from $reg_2$ to $reg_1$ for `totalFS`. Since the moves are cyclic we introduce a temporary stack slot $st_1$ to break the cycle. The moves are inserted at the beginning of $b_2$. For the edge $(b_2, b_3)$ we need to fix the data-flow mismatch for **this** with a move from $reg_2$ to $reg_1$. In addition, we need to destruct the $\varphi$ which assigns `add` to `totalFS`. To do so we insert a move from $reg_1$ to $reg_2$. As this example demonstrates, $\varphi$s and data-flow mismatches need to be resolved simultaneously since they may also introduce a cycle. We therefore introduce another temporary stack slot $st_2$. The resolution code is inserted at the end of $b_2$.

**Figure 4.9:** Inserted move instructions for global data-flow resolution

Where should the phase place the resolution code? As suggested in the above example, we would want to insert it along a control-flow edge. Since there are no critical edges, this is easy. Either the source of an edge has only a single successor, then we can insert the code at the end of the source block. Otherwise, the target must have only a single predecessor. Therefore, we can place the code in the target block.

Conceptionally, all resolution moves are done in parallel. In reality, however, most processors force us to place the moves in sequential order.[3] Since there might be write-after-read dependencies (also called anti-dependencies) [Cooper and Torczon, 2011, Chapter 12.2] between move instructions, we need to be careful with the order in which we insert the moves. In the worst case, those dependencies are *cyclic*, that means that we cannot find an order so that all dependencies are satisfied. The edge $(b_0, b_2)$ in Figure 4.8 shows such an example. To resolve the location mismatch for **this** we need to insert a move from $reg_1$ to $reg_2$. However, to fix the data flow `totalFS`, a move from $reg_2$ to $reg_1$ is required. No matter how we place the moves, we would

[3] Mohr et al. [2013] proposed special hardware to speed up the *shuffle code*. While this would avoid the move emission, compiler support is still required [Buchwald et al., 2015].

overwrite one of the two values. Note, that the cycle can consist of an arbitrary number of move instructions. To break such a cycle, we introduce a new temporary stack slot. Figure 4.9 shows the result for the example.

In addition to a location mismatch, there might be $\varphi_{in}/\varphi_{out}$ pairs that need to be resolved. We need to handle them in parallel to the locations, since they might have interdependencies. The edge $(b_2, b_3)$ in Figure 4.8 shows an example. Due to the location mapping for `this`, we need a move from $reg_2$ to $reg_1$. To resolve the $\varphi$ for add, the content of $reg_1$ should be copied to $reg_2$. Again, this forms a cycle and requires a temporary stack slot to break it.

# Chapter 5

# Register Allocation Strategies

The task of a register allocation strategy is to solve the register allocation problem, i.e., to replace variables with registers, for a single trace. Since a trace is a list of sequential blocks, allocating registers for a trace is significantly simpler than for a method with arbitrary control-flow. First, there are no holes in the lifetime intervals of variables (see Theorem 14). Since variables adhere to the SSA form, we know that a variable is *live* from its definition in the trace (either the explicit definition, or the $live_{in}$ set of the trace header, see Theorem 13) to the last usage in the trace, which might be a real usage or an entry in the $live_{out}$ set. In addition, a strategy must resolve $\varphi$-functions for the predecessor that is in the same trace as the merge block. (Inter-trace edges with $\varphi$s are handled by the global data-flow resolver, which was discussed in Section 4.5.)

In the course of this work, we implemented three strategies, a *linear scan* allocator (Section 5.1), a special *trivial trace* allocator (Section 5.2) and a fast *bottom-up* allocator (Section 5.3).

## 5.1 Linear Scan Allocator

The trace-based linear scan algorithm originated from an adaption of the global approach by Wimmer and Mössenböck [2005] and Wimmer and Franz [2010], which is the default register allocator in Graal. However, due to the straightforward structure of traces, the implementation could be significantly simplified. As a result, large parts of the code have been rewritten. In this section, we given an overview of our trace-based linear scan strategy with a focus on the differences to the global variant.

Figure 5.1 shows the phases of the trace-based linear scan strategy. The general idea is that first an interval representation of *liveness* is created. These *intervals* capture all information necessary to perform the actual allocation. Working with intervals as an intermediate data structure is more efficient than modifying the LIR directly. This is especially important, since decisions

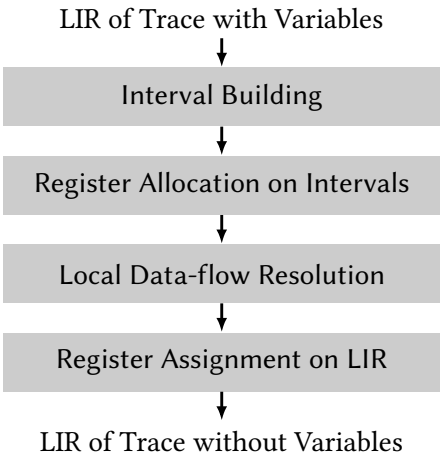


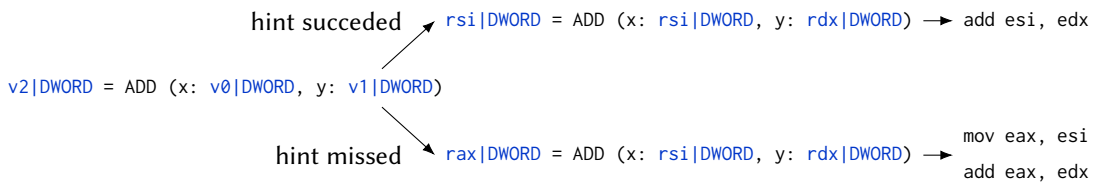

**Figure 5.1:** Trace-based linear scan

during allocation are not final in general. For example, the spill position, i.e., the position where a variable is spilled to the stack, might change during allocation. After the allocation pass is finished, a data-flow resolution phase handles data-flow mismatches and $\varphi$s across intra-trace edges, which might have been introduced by interval splitting. Last but not least, the variables in the LIR are replaced by the locations stored in the corresponding intervals.

### 5.1.1 Interval Building

First, we number the instructions of the traces. For technical reasons we only use *even* numbers, and start at 2. The *interval builder* performs a backwards liveness analysis. It creates an interval for every *variable* and every *register* in the LIR instructions of the trace. *Variable intervals* are used to perform register allocation. Since they have no lifetime holes (see Theorem 14) their representation is simple. For every interval, we store the *variable*, the (allocated) location, the spill slot, a *from* and a *to* position, as well as a list of *usages*. Listing 1 in the appendix shows the relevant parts of the class `TraceInterval`. *Register intervals* are used to track *fixed register constraints*, which are imposed by *calling conventions*,[1] instruction set restrictions,[2] or special purpose registers used by the virtual machine.[3] The allocator needs to respect these requirements while it allocates registers for variables. Fixed registers do not adhere to SSA form, since a register might be defined multiple times. This implies that register intervals can have lifetime holes. Therefore, they are represented as a list of *live ranges*, each with a *from* and *to* position. These ranges are usually short, spanning only very few instructions, and are never *live* across a block boundary.

[1] Calling conventions specify, for example, parameter or return registers.
[2] For example, the result of the `idiv` instruction on AMD64 is implicitly stored in `rax` and `rdx` [Intel, 2013b].
[3] On HotSpot, register `rbp` is used as the frame pointer register on AMD64.




(a) LIR before register allocation (b) LIR after register allocation (c) Machine code

On the left, the ADD instruction in LIR with two input operands (v0, v1) and one output operand (v2). To efficiently model the AMD64 two-address add [Intel, 2013b], there is a hint from v2 to v0. If the hint succeeds, only one add instruction is emitted (top right). If the hint is missed, there is an additional *mov* instruction (bottom right).

**Figure 5.2:** Two-address instructions

**Hints** During interval building we record *hints* for intervals, which where proposed by Wimmer and Mössenböck [2005] for global linear scan. A *hint* is a reference to a *variable* or *fixed interval*. It advises the allocator to assign to a variable the same register as in the hint, if the respective register is available. *Register hints* are a light-weight alternative to *coalescing*, i.e., merging two non-overlapping intervals. In Graal, the main source for hints are move instructions. If the hint is adhered, the move instruction can be removed. In addition, hints are important for *two-address* instructions, i.e., instructions where the result is stored back to one of the operands. These instructions violate SSA form since they redefine a value. In LIR, such instructions are represented as *three-address* instructions and emit two machine instructions, a move and the actual operation. Figure 5.2 shows an example. To avoid the move—if possible—a hint is added from the operand to the result variable.

The variables defined via the $\varphi_{in}$ set also get a hint to their respective value in the $\varphi_{out}$ set of the intra-trace predecessor. Note that such a predecessor is always within the trace, due to the fact that a loop header is never a trace head (Theorem 8).

### 5.1.2 Register Allocation on Intervals

The main register allocation procedure iterates all intervals in the order of increasing start positions. In each iteration one interval is processed. We refer to this as the *current interval*. The *from* position of this interval is the *current position*. The allocator organizes the intervals in four sets:

- *Unhandled variable intervals* are intervals which are not yet live, i.e., they start after the current position. This set is sorted by increasing *from* positions.

- *Active variable intervals* are live at the current position and have a register assigned. They are sorted by increasing *to* positions.

- *Active register intervals* are fixed register intervals that have a *range* that covers the current position. They are also sorted by increasing *to* position of their *current* ranges.

- *Inactive register intervals* do not have a range covering the current position. They are sorted by increasing *from* positions of their *next* range.

The allocator continues as long as there are intervals in the *unhandled* set. The interval with the lowest *from* position is removed from the set and is processed. First, the other interval sets are adjusted. Variable intervals with a *to* position smaller than the current position are removed from the *active* set. The register intervals in the *active* (*inactive*) set are removed if the *to* (*from*) position of their current range is smaller than the current position. The removed register intervals might be readded to the *active* or *inactive* set if there is another range with ends after the current position.

Next, the allocator tries to assign a free register to the current interval. To do so, it builds a map from registers to *free-until* positions, i.e., to the position when the registers become available. All entries are initialized with *infinity*. The registers assigned to *variable* and *fixed intervals* in the *active* sets are removed. Finally, for all *inactive register intervals* we record the *from* position of the next range. From this set of feasible registers, the allocator selects the register with the smallest *free-until* position that is still bigger than the current interval's *to* position. If there is such a register the allocation succeeded. However, due to fixed register constrains, there might be no register available for the full interval. In that case, we select the register with the biggest *free-until* position, *split* the current interval at this position and assign the register to the partial interval. The split child is added to the *unhandled* list. In addition, we insert a move from the parent interval to the child interval at the split position. In case the current interval has a hint, we favor the hint register by ignoring the *free-until* position. However, a partially available hint will never overrule a fully available register.

If we fail to find a register that is either fully or partially free, we need to spill. The goal is to select a register which is not accessed until the next usage of the current interval. Again, we collect information about the feasible registers. In case the register is occupied by a variable interval, we record the next usage of this interval, which is where the interval needs to be reloaded. We remove *active fixed intervals* from the list since we cannot spill them. The *from* position of the next range of inactive fixed intervals also contributes to the *next usage* of a register. With this information at hand, we select the register with the highest *next usage* position. If two registers

have the same position, we favor the one where the corresponding variable interval has already been spilled in the past. That way, we can avoid to insert a spill move. If we do not find a register, we bailout and abort compilation.[4] Otherwise, the candidate register is occupied by a variable.

We split the interval which currently occupies the register at the current position, as well as before the next usage. The parent interval has already a register assigned and will not be touched, other than setting the *to* position. The first split child is located on the stack. The last split child is unassigned and inserted into the *unhandled* list. Finally, we set the location of the current interval to the freed register and are finished.

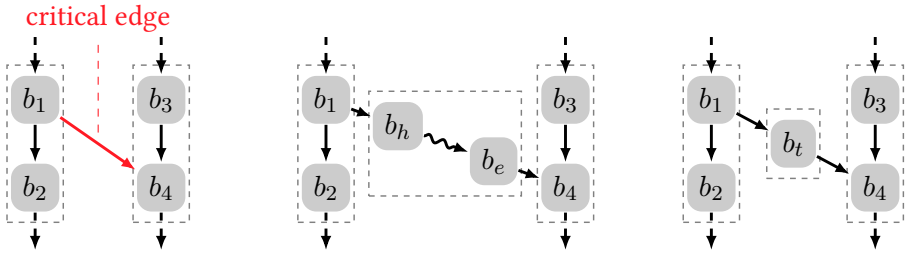
However, an *inactive register interval* might restrict the period for which a register candidate is available. If the start of the next range of the fixed interval is greater than the *to* position of the current interval, we can use the register for the entire interval. Otherwise, the register is only partially available and we perform the same actions as in the *allocate free register* scenario, i.e., split the current interval and add the split child to the *unhandled* list.

**Splitting and Spilling** When an interval needs to be split, the algorithm computes the *latest possible* split position. However, we can split the interval anywhere between the *last usage* and the *latest position*. If both positions are in the same block, no optimization is possible. Otherwise, we split the interval in the block with the lowest execution frequency between both positions. Therefore, the corresponding move instruction is executed less often. Since we need to ensure that moves which are inserted at a particular position are properly ordered, we split the intervals at block boundaries and let the local data-flow resolution phase deal with it. Since we are in SSA form, we only need to spill a value once. The spill position is independent from the split positions, as long as the spill happens between the definition of the value and the first stack interval. Similar to the split position we choose the block with the lowest execution frequency.

### 5.1.3 Local Data-flow Resolution

Due to interval splitting, the location of a value might be different at both ends of an intra-trace edge. Similar to the *global data-flow resolution*, which we described in Section 4.5, we perform a *local* resolution phase. It also takes care of $\varphi$-instructions. The only difference to its global counterpart is that the local resolution operates on intervals instead of on the *global liveness information*. It is worth noting that due to the placement of the resolution moves, the local and the global data-flow resolution phases do not interfere.

[4] Usually, this is an evidence for a compiler bug, since the number of registers required at the current position—that are active fixed register intervals and registers required by the current instruction—is higher than the number of available registers. In production this should never happen.




(a) Critical edge (b) Link trace (c) Trivial link trace

**Figure 5.3:** Link traces

### 5.1.4 Register Assignment on LIR

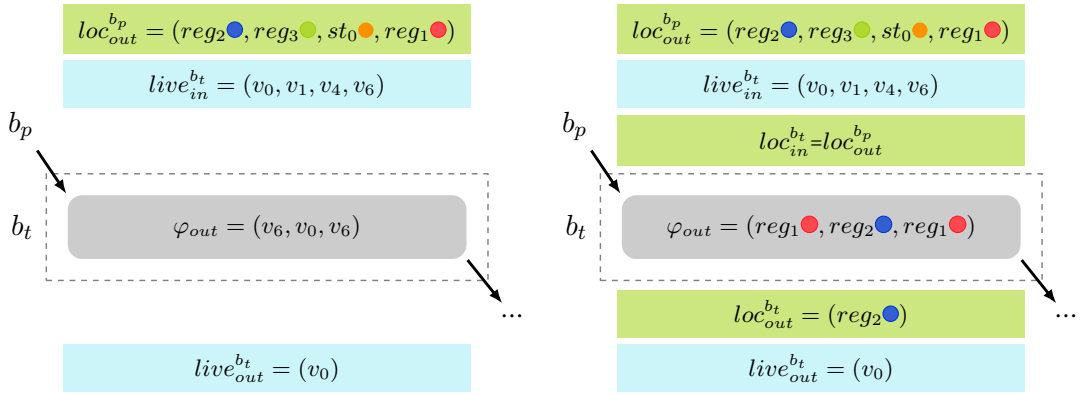
Since the algorithm splits already assigned intervals, the final location of a variable at a given instruction is only known when all intervals are processed. Therefore, we conclude linear scan allocation with an assignment phase, which iterates all instructions and replaces all variables with the location recorded in the corresponding interval.

## 5.2 Trivial Trace Allocator

As already mentioned, control-flow can exit and enter a trace at any block. Let us consider the example in Figure 5.3a. If an edge would lead from inside a trace (i.e., not from the last block) to the middle of another trace, this edge would be *critical*. Therefore, there must be (at least) a trace in between which is entered at the *head* and left via the last block, as shown in Figure 5.3b. Often, this trace consists of a single block. Figure 5.3c depicts an example. We call such traces *trivial traces*.

**Definition 33** (Trivial Trace)**.** A *trivial trace* is a special kind of trace with a single basic block which contains only a single *jump* instruction.

Trivial traces are very common. For the DaCapo benchmark suite, about 40% of the traces are trivial (see Figure 8.2). The unidirectional trace-building example in Figure 4.2c contains one trivial trace (T3). In the bidirectional case (Figure 4.2d) two out of four traces are trivial (T2 and T3).




(a) Before allocation (b) After allocation

Example allocation of a *trivial trace* consisting of block $b_t$. A trivial trace is a trace consisting of a block without any instruction (except a jump). There are several interesting observations to make. First, the live sets are ordered w.r.t. the variable indices. Second, the $live_{out}$ set is a subset of $live_{in}$. Third, the values in $\varphi_{out}$ are ordered w.r.t. to their respective $\varphi_{in}$ and there might be duplicates and variables not in $live_{out}$.

**Figure 5.4:** Example of a trivial trace allocation

Due to their simple structure, trivial traces are significantly easier to allocate than general traces. But why do we even have to allocate them? Like for every other trace, we need to populate the $loc_{in}$ and $loc_{out}$ locations according to their respective $live_{in}$ and $live_{out}$ sets. In addition, there may be outgoing values in $\varphi_{out}$, which need to be replaced with a register or stack slot. However, both requirements are easy to fulfil.[5]

Figure 5.4a shows a typical example of a trivial trace. The variables in the $live_{in}$ and $live_{out}$ sets are sorted by construction. Also, since there cannot be a variable definition in a trivial trace, $live_{out}$ is a subset of $live_{in}$. In which case is $live_{out}$ a proper subset of $live_{in}$? As described in Section 4.3.2, the live sets are shared between a split block and its successors. So a variable $v$ might be *live* in a sibling of a trivial trace block, but in or after the trivial trace. Therefore, $v$ is in $live_{in}$ but not in $live_{out}$ of the trivial trace. The outgoing $\varphi_{out}$ variables can only reference variables in $live_{in}$. However, the *order* in which variables occur in $\varphi_{out}$ is arbitrary. There might even be duplicates.

Algorithm 3 shows the allocator in pseudo code. It assumes that the live sets are ordered according to their variable index.[6] First, on line 2, the $loc_{in}$ is set to the $loc_{out}$ of the predecessor. Next, the algorithm iterates over all incoming variables (line 5). In case the variables occur in the $live_{out}$ set of the trivial trace, the $loc_{out}$ location is set to the current $loc_{in}$ location (loop 6). Afterwards, the allocator resolves $\varphi$ values for the $\varphi_{out}$ list on line 11. Since the $live_{in}$ set is sorted, we can *binary search* for the index of the respective location in the $loc_{in}$ set. Note that we

[5] One might say *trivially*...
[6] In our implementation this is true by construction of the sets by the global liveness analysis (Section 4.3.2).

**Algorithm 3** Trivial Trace Allocation Strategy

```
1:  procedure TrivialTraceAllocator( loc_out^pred, live_in, live_out, φ_out )
2:      loc_in ← loc_out^pred                ▷ Incoming locations are the outgoing of the predecessor
3:      loc_out ← new list of size |live_out|x]   ▷ Reserve space for outgoing locations
4:      idx_out ← 0                          ▷ Incoming and outgoing indices might diverge
5:      for idx_in ∈ {0, ..., |loc_in| − 1} do
6:          if live_out[idx_out] = live_in[idx_in] then   ▷ The incoming variable is also in live_out
7:              loc_out[idx_out] ← loc_in[idx_in]        ▷ Set outgoing location
8:              idx_out ← idx_out + 1                    ▷ Increment outgoing index
9:          end if
10:     end for
11:     for idx_φ ∈ {0, ..., |φ_out| − 1} do
12:         val ← φ_out[idx_φ]                           ▷ Get φ_out value
13:         if val is variable then
14:             idx_in ← binarySearch(val, live_in)      ▷ Find location index for val using binary search
15:             φ_out[idx_φ] ← loc_in[idx_in]            ▷ Update φ_out
16:         end if
17:     end for
18:     return (loc_in, loc_out)
19: end procedure
```

also experimented with a version that used a *variable-to-location* map to avoid repetitive searching. However, empirical experiments suggest that the binary search variant outperforms the map approach by $40\%$. We account this to the reduced number of allocations.

## 5.3 Bottom-Up Allocator

Not all traces of a method are equally important for peak performance. As already said, we do not want to spend too much time on traces that are infrequently executed. Although the linear scan strategy outlined in Section 5.1 exhibits a linear time behavior with respect to the number of instructions [Eisl et al., 2016], it is relatively complex and requires multiple passes over the instructions of a trace. The constant factors are relevant in practice. Therefore, we aim for a fast, general-purpose allocation strategy that sacrifices peak performance for allocation time. The idea is to perform (local) liveness analysis, register selection and LIR modification in a single backwards pass over the instructions. Therefore, we call this strategy *bottom-up allocator*.

### 5.3.1 Tracking Liveness Information

In the bottom-up allocator, liveness information is never maintained for the whole trace but is only known locally for the current instruction. This information is tracked using two data structures. The *register content* map stores the current contents of every register. The entry for

**Algorithm 4** Bottom-up allocator: `allocateTrace`

```
1:  procedure allocateTrace(T)
2:      b_last ← null
3:      variable location, register content ← preinitFromSuccessor(T)
4:      for b ∈ reverse(T) do                          ▷ Iterate blocks in reverse order
5:          if b_last ≠ null then
6:              resolvePhis(b, b_last)                 ▷ Handle φs in Trace, see Algorithm 12
7:          end if
8:          for all inst ∈ reverse(instructions(b)) do  ▷ Iterate instructions in reverse order
9:              allocateInstruction(inst, b)           ▷ See Algorithm 5
10:         end for
11:         b_last ← b
12:     end for
13:     resolveLoopBackEdge(T)                         ▷ Handle loop intra-trace back-edge
14: end procedure
```

a register points to a *variable* if the variable is currently stored in this register. It can also point to a register itself, which indicates that there is a *fixed register constraint*, e.g., due to calling convention requirements. An entry in the register content map might be *empty* in case the register is currently unused. The second data structure is the *variable location* map. It tracks the current location of every variable, which is either a register, a stack slot, or empty if the variable is not live. We also track which registers are used in the current instruction. The memory requirement is therefore linear in the number of registers and the number of variables. Only the size of the second map depends on the compilation unit while the size of the first one is fixed for a given architecture.

Note that the bottom-up approach does not require the SSA-property and can deal with lifetime holes without modification. In fact, it does so for fixed register constraints, which do not adhere to the SSA properties.

### 5.3.2 Register Allocation

Register allocation is done in a single backward pass over the instructions of a trace (see Algorithm 4). If the last block of the trace has a successor that has already been allocated, we use the allocation information from this successor to initialize the *variable location* and *register content* maps.

An instruction is processed in two phases, as outlined in Algorithm 5. We start with the values that are *defined* ($inst.output()$) by the instruction before dealing with the values that are used by it ($inst.input()$). When visiting the values, we first process fixed register usages to mark them as *used* in the *register content* map (see Algorithm 6). Next, we iterate the operands of the instruction. We start with those variables which *must* reside in a register (see Algorithm 7). Such

**Algorithm 5** Bottom-up allocator: `allocateInstruction`

```
1:  procedure allocateInstruction(inst)
2:      if inst.isCall() then
3:          spillCallerSavedRegisters(inst)
4:      end if
5:      allocateOps(inst.output())                    ▷ Allocate output operands
6:      freeRegisters(inst.output())                  ▷ Free register in output operands
7:      allocateOps(inst.input())                     ▷ Allocate input operands
8:  end procedure

9:  procedure allocateOps(ops)
10:     for all op ∈ ops if isFixedRegister(op) do    ▷ Fixed-register operands
11:         allocFixedRegister(op)                    ▷ No need to reassign, see Algorithm 6
12:     end for
13:     for all op ∈ ops if requireRegister(op) do    ▷ Variables which require a register
14:         op ← allocRegister(op)                    ▷ See Algorithm 7
15:     end for
16:     for all op ∈ ops if mayBeOnStack(op) do       ▷ Variables which might be on stack
17:         op ← allocStackOrRegister(op)             ▷ See Algorithm 8
18:     end for
19: end procedure

20: procedure freeRegisters(ops)
21:     for all reg ∈ ops if isRegister(reg) do       ▷ Free registers that have been defined
22:         variable location[register content[reg]] ← null   ▷ Clear variable location
23:         register content[reg] ← null              ▷ Clear register content
24:     end for
25: end procedure
```

**Algorithm 6** Bottom-up allocator: `allocFixedRegister`

```
1:  procedure allocFixedRegister(reg)
2:      evacuateRegisterAndSpill(reg)                 ▷ Spill reg if occupied, see Algorithm 11
3:      setLastRegisterUsage(reg)                     ▷ Mark register as used in the current instruction
4:  end procedure
```

**Algorithm 7** Bottom-up allocator: `allocRegister`

```
1:  function allocRegister(var)
2:      loc ← variable location[var]
3:      if isRegister(loc) then                          ▷ Variable is already in a register
4:          setLastRegisterUsage(loc)                    ▷ Mark register as used in the current instruction
5:          return loc
6:      end if
                                                         ▷ Variable on the stack or not allocated, need new register
7:      reg ←findFreeRegister(var)                       ▷ See Algorithm 9
8:      if no reg avaiable then
9:          reg ←findRegisterToSpill(var)                ▷ See Algorithm 10
10:     end if

11:     variable location[var] ← reg                     ▷ Update variable location
12:     register content[reg] ← var                      ▷ Update register content
13:     setLastRegisterUsage(reg)                        ▷ Mark register as used in the current instruction

14:     if isStack(loc) then                             ▷ Variable was on the stack, need spill move from reg to stack
15:         if var is an input operand then              ▷ Input operand, is live before the instruction
16:             insertMoveBefore(loc, reg)               ▷ Might be destroyed by an output operand
17:         else                                         ▷ Output operand, is live after the instruction
18:             insertMoveAfter(loc, reg)
19:         end if
20:     end if

21:     return reg
22: end function
```

**Algorithm 8** Bottom-up allocator: `allocStackOrRegister`

```
1:  function allocStackOrRegister(var)
2:      loc ← variable location[var]
3:      if loc ≠ null then                               ▷ Variable is already in a stack slot or a register
4:          setLastRegisterUsage(loc)                    ▷ Mark register as used in the current instruction
5:          return loc
6:      end if
                                                         ▷ Variable not allocated, find register or stack slot
7:      loc ←findFreeRegister(var)                       ▷ Try first to get a register, see Algorithm 9
8:      if loc ≠ null then                               ▷ Register available
9:          setLastRegisterUsage(loc)                    ▷ Mark register as used in the current instruction
10:         register content[loc] ← var                  ▷ Update register content
11:     else                                             ▷ No register available, use stack slot
12:         loc ←allocateSpillSlot(var)                  ▷ Get a spill slot for the variable
13:     end if

14:     variable location[var] ← loc                     ▷ Update variable location
15:     return loc
16: end function
```

**Algorithm 9** Bottom-up allocator: `findFreeRegister`

```
1: function findFreeRegister(var)
2:     for all reg ∈ allocatableRegisters(var) do
3:         if register content[reg] = null then
4:             return reg                                   ▷ Free register found
5:         end if
6:     end for
7:     return null                                          ▷ No free register found
8: end function
```

**Algorithm 10** Bottom-up allocator: `findRegisterToSpill`

```
1: function findRegisterToSpill(var)
2:     for all reg ∈ allocatableRegisters(var) do
3:         if ¬isUsedInCurrentInstruction(reg) then   ▷ Register which is not used currently
4:             evacuateRegisterAndSpill(reg)              ▷ Spill reg, see Algorithm 11
5:             return reg
6:         end if
7:     end for
8:     error No register found for spilling
9: end function
```

**Algorithm 11** Bottom-up allocator: `evacuateRegisterAndSpill`

```
1: procedure evacuateRegisterAndSpill(reg)
2:     var ← register content[reg]                          ▷ Get current register content
3:     register content[reg] ← null                         ▷ Clear register content
4:     stack ←allocateSpillSlot(var)                        ▷ Get a spill slot for the variable
5:     variable location[var] ← stack                       ▷ Update variable location
6:     insertMoveAfter(reg, stack)                          ▷ Insert reload from stack to register
7: end procedure
```

**Algorithm 12** Bottom-up allocator: `resolvePhis`

```
1: procedure resolvePhis(b_src, b_dest)
2:     for i ∈ {0, ..., |φ_out^{b_src}|} do
3:         out ← φ_out^{b_src}[i]
4:         in ← φ_in^{b_dest}[i]
5:         loc ← variable location[out]
6:         if loc ≠ null then                    ▷ Outgoing variable already in a location
7:             insertMoveAfter(in, loc)                     ▷ Insert move from out to in
8:         else                          ▷ Outgoing variable not yet assign, reuse incoming
9:             variable location[out] ← in                  ▷ Update predecessor location
10:            if isRegister(in) then
11:                register content[in] ← out               ▷ Update register content
12:            end if
13:        end if
14:    end for
15: end procedure
```

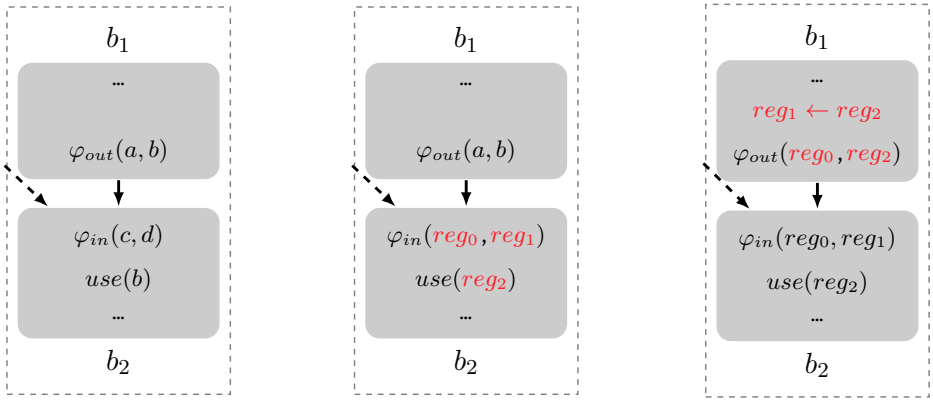
a variable might already be in a register. In this case we only need to replace its occurrence in the instruction with the register and are done. If the location of the variable is not yet defined, i.e., if it is the last usage of the variable, or if it is currently stored on the stack, we need to find a free register for it (see Algorithm 9). To do so, we iterate the list of registers and look up their *register content* entry. If we find a register that is unused, i.e., its entry is empty, we can assign it to the current variable. If there is no free register, we need to free one by spilling one of the registers. This is shown in Algorithm 10. We heuristically choose the first register which is not used in the current instruction and is not a fixed register. We spill the variable in this register and insert a move from the spill slot to the register (i.e., a reload) right *after* the current instruction (Algorithm 11). If the variable that is to be loaded into this register was previously on the stack, we also insert a move from the stack slot to the register *before* the current instruction.

After all *strict register* requirements are fulfilled, we process the variables which might either be in a register or on the stack (see Algorithm 8). If the variable is already in a location, we are done. Otherwise we try to allocate a free register (Algorithm 9). If no register is available, we allocate the variable to a stack slot.

### 5.3.3 $\varphi$-resolution

At block boundaries the allocator needs to take care of $\varphi$-instructions. As explained in Section 2.4.1, $\varphi$-instructions are basically *parallel moves* from the locations in the predecessor block ($\varphi_{out}$) to the locations in the successor block ($\varphi_{in}$). At the beginning of a basic block, all variables in the $\varphi_{in}$ set have already been assigned to a location. Due to the *single definition* property of the SSA form (Property 1), we know that these variables are not live in any predecessor, i.e., they are defined at the beginning of the merge block. Therefore, we can directly reuse their locations for those variables in the $\varphi_{out}$ set which are not yet mapped to a location. This way we can avoid unnecessary move operations. For the variables that are already assigned to a different location, we need to insert moves to satisfy the data-flow requirements. Algorithm 12 shows this resolution step.

Figure 5.5 shows an example for $\varphi$-resolution. Figure 5.5a shows a trace consisting of two blocks $b_1$ and $b_2$. Block $b_2$ is a merge that contains two $\varphi$-variables, $c$ and $d$. In the predecessor $b_1$, these variables are matched to $a$ and $b$, respectively. After allocation of $b_2$ (Figure 5.5b) we allocated $b$ to $reg_2$, $c$ to $reg_0$ and $d$ to $reg_1$. Before we continue with $b_1$, we need to resolve the data flow between $\varphi_{out}$ and $\varphi_{in}$. Namely, we want to map $a$ to $reg_0$ and $b$ to $reg_1$. Since $a$ is not yet assigned to a location, we can simple replace it with $reg_0$. Variable $b$, on the other hand is already stored in $reg_2$. To resolve this data-flow mismatch, we need to insert a move from $reg_2$ to $reg_1$. Figure 5.5c shows the result of the resolution step.




(a) Before $b_2$ (b) After $b_2$ (c) After $b_1$

**Figure 5.5:** $\varphi$-resolution in the Bottom-Up allocator

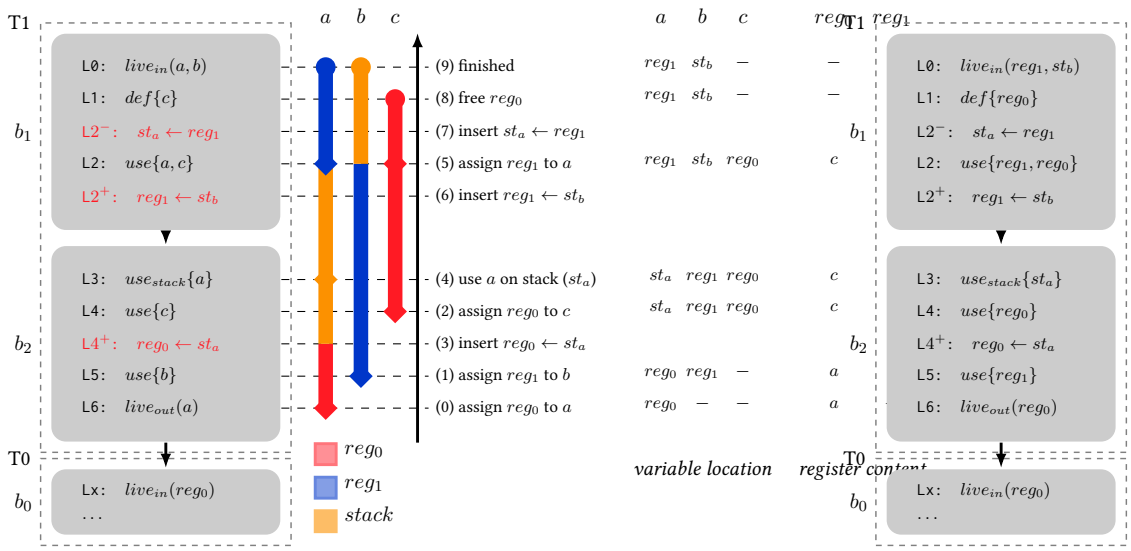
### 5.3.4 Loop Back-Edge

A loop back-edge can only occur at the end of a trace (Corollary 2). Once again, we need a resolution step for this edge to fix the data-flow. This is handled in a similar fashion as done in the *global data-flow resolution*.

### 5.3.5 Example

Figure 5.6 depicts bottom-up allocation of a simple trace T1 with two blocks, $b_1$ and $b_2$. For readability, we omitted the details of the instructions and only show the operand modes $use$, $def$ and $use_{stack}$, where the last one represents a *usage* which can directly access the stack. To the right of the blocks, we visualize the live intervals of the variables. This information is never explicitly stored. Next to the intervals, we describe the *action* that is performed when processing the corresponding instruction. Actions are numbered from (0) to (9) in processing order. On the right-hand side of Figure 5.6, we display the contents of the *variable location* and the *register content* maps *after* the instruction has been processed.

The allocator starts with the outgoing values at line `L6` at the end of block $b_2$. The successor trace T0 has already been allocated, so the algorithm can match the incoming variable location $live_{in}(reg_0)$ of block $b_0$ with the outgoing variable locations $live_{out}(a)$ in $b_2$. This initializes the *variable location* entry of $a$ to $reg_0$ and the *register content* of $reg_0$ to $a$. Also, $a$ is replaced with $reg_0$ in the instruction at `L6` (0). We continue with the instruction in line `L5`. Variable $b$ has no location assigned to it so we query the *register content* map for the next free register which is $reg_1$ (1). The next instruction to be processed is the usage of $c$ in line `L4`. All registers are currently occupied so the allocator arbitrarily selects $reg_0$ for spilling (2). Since the location of $a$ changes from a register to a stack slot, we insert a move from the stack slot $st_a$ to $reg_0$ right




(a) Allocation Example (b) After Allocation

**Figure 5.6:** Bottom-Up allocation example

after the instruction that is currently processed (3) at line L4. We continue at line L3 with the usage of variable $a$, which is currently stored in stack slot $st_a$. Since the instruction can directly address the stack, the allocator simply replaces the variable with $st_a$ (4). Next we process the instruction in line L2. Variable $a$ is currently located in stack slot $st_a$, but the current usage requires a register. Since all registers are occupied, we need to select one for spilling. We cannot spill $reg_0$ because it is the location of $c$, which is used in the current instruction. Therefore, we choose $reg_1$ and assign it to $a$ (5). As $reg_1$ contains the value of variable $b$, we need to insert a move from $st_b$ to $reg_1$ after line L2 (6). Variable $a$ also changed its location from $st_a$ to $reg_1$. To adjust the data-flow, the allocator inserts a move from $reg_1$ to the stack slot $st_a$ before the current instruction on line L2 (7). The allocator advances to line L1 which contains the definition of variable $c$. We mark the register $reg_0$ as free and clear the entry for $c$ in the *variable location* map (8). The last instruction on line L0 contains pseudo usages of variables $a$ and $b$. The operands of the instruction are replaced with the current locations of the variables. Figure 5.6b shows the result after allocating the trace is finished.

### 5.3.6 Ideas that did not Work Out

We kept the bottom-up allocator as simple as possible since its main goal is to be fast. However, we did experiment with modifications to improve the quality of allocation. We want to describe them here and explain why they were removed again.

**Reduce Spill Moves** The placement of spill moves is suboptimal. Due to SSA form, it is sufficient to spill a variable only once since it never changes. Still, the bottom-up allocator will happily update the stack-slot every time a variable's location changes from a register to the stack. We implemented a mode where spilling is delayed and the move is inserted only after the definition of the variable is reached. Although, this reduces the number of moves significantly, it did also increase compile time.

**Better Spilling Heuristics** The bottom-up allocator always spills the first non-blocked register. We experimented with the *furthest-first* strategy [Guo et al., 2003], i.e., we selected the register whose next usage was farthest away. However, it had again a negative influence on compile time, so we abandoned the idea.

**Round Robin Selection** Due to our spilling and selection strategy, registers which come first in the set of available registers, are used more frequently than those at the end of the set. Only recently, Chen et al. [2018] proposed using a *round-robin*-style register set. In theory, this would distribute the register usage better. In addition, the spilling heuristic could easily select a register which has not been allocated recently. In practice, it did not make a difference, neither in terms of peak performance nor in compile time. In favor of simplicity, we kept the original implementation.

**Summing it up** Although, the described modifications did not work out for our use case, they might be worth revisiting if the focus shifts. We also want to remark that our main target was AMD64, which performs hardware optimizations such as *register renaming* [Hennessy and Patterson, 2003]. Simpler architectures might be more sensible to the optimizations above.

# Chapter 6

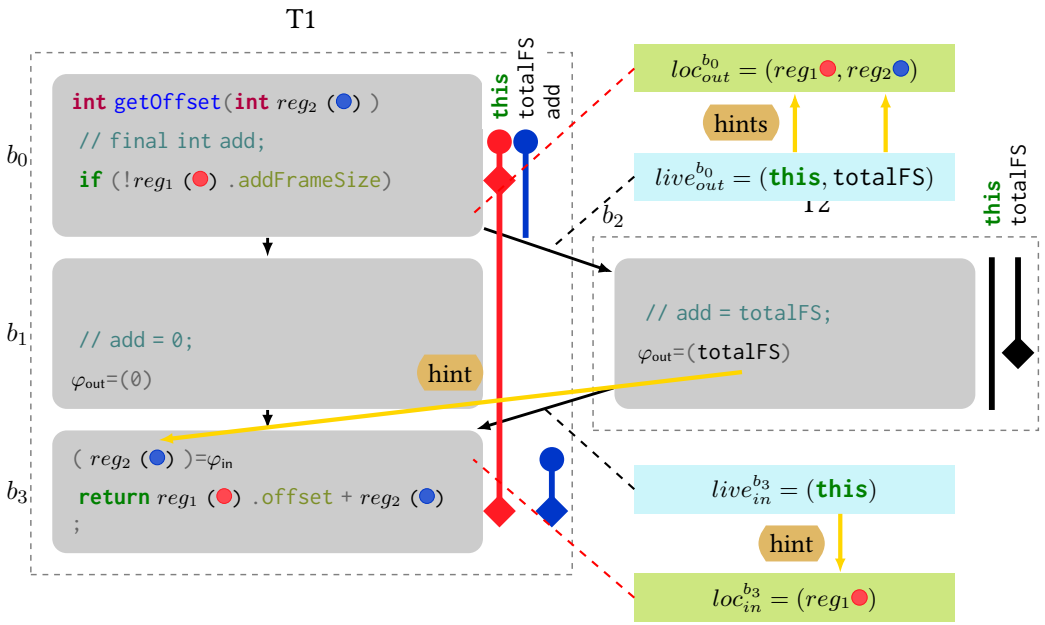
# Inter-trace Optimizations

We have already argued that traces can be processed independently of each other. The data-flow resolution guarantees the correctness of trace register allocation. It does so by inserting moves whenever the location of a variable is different across an inter-trace edge. If we manage to assign the same location, we can avoid the moves and therefore improve the allocation quality. We implemented three optimizations to reduce the number of moves introduced during data-flow resolution and thus to avoid unnecessary spill moves. They all follow the same idea.

All optimizations provide *allocation strategies* with information from already processed traces to *guide* the allocation of an unprocessed trace. Therefore, the order in which we process traces has an influence on the allocation quality. As a basic principle of trace register allocation, we strive for the best allocation of a trace, ignoring the rest of the compilation unit. Using information from other traces violates this principle, since it restricts the freedom of the current trace. To keep the influence on peak performance low, we allocate traces in decreasing order of *importance*. In our case, importance is basically defined by the order the *trace builder* finds traces (see Section 4.1). The only exceptions are *trivial traces*. We allocate them right after their preceding trace.[1] For instance, in the bidirectional trace-builder example in Figure 4.2d, we would allocate T2 and T3 before T1.

Also, the optimizations work on the scope of *inter-trace edges*. This retains the *non-global* character of the approach and keeps the complexity limited. In addition, we communicate all information that is needed for the optimizations via the global liveness information (see Section 4.3.2). Using this canonical representation decouples the optimizations from the details of the allocation strategies. For instance, there is no need to keep the intervals of the linear scan strategy live.

[1] As we have argued in Theorem 9, the trace head has at most one predecessor.




Inter-trace hint (orange) for trace T2 from the allocation result in T1. Note that the $live_{out}$ and $live_{in}$ sets, as well as the $\varphi_{in}$ sets, are considered. Full source code for the `getOffset` method is depicted in Listing 2.

**Figure 6.1:** Inter-trace hints example (`getOffset`)

## 6.1 Inter-trace Hints

*Inter-trace hints* are the most straight forward optimization. We use use $live_{in}/live_{out}$ and $\varphi_{in}/\varphi_{out}$ of an already allocated predecessor trace as a *hint* for the current trace. Figure 6.1 depicts an example. Trace T1 has already been allocated. When allocating T2, the $live^{b_0}_{out}$ advises the strategy to allocate `this` to $reg_1$ and `totalFS` to $reg_2$. Also the $\varphi_{in}$ set in $b_3$ suggests to allocate `totalFS` to $reg_2$. Note, however, that two inter-trace hints may disagree. The final decision which hint is followed—if any—is made by the allocation strategy.

Every strategy has a different way of handling inter-trace hints. In the linear scan strategy (Section 5.1), the inter-trace hints are an addition to the already existing (intra-trace) hints. Only the $loc_{out}$ of the (single) predecessor of the trace head is considered. This offers several advantages. First, if we allocate traces in trace building order, the predecessor has already been allocated. This is due to the *greediness* of the trace building algorithms (Definition 31). Second, as shown in Theorem 13, all variables that are entering a trace are live at the beginning of the trace head. Processing the trace head is therefore sufficient to get hints for *all* variables. We explicitly do not use inter-trace hints for variables defined by $\varphi$s because we favor the hint coming from the predecessor that is part of the current trace.

The trivial trace allocator (Section 5.2) simply forwards the hinting information to make it available to its successor. That is why trivial traces are handled as soon as possible.

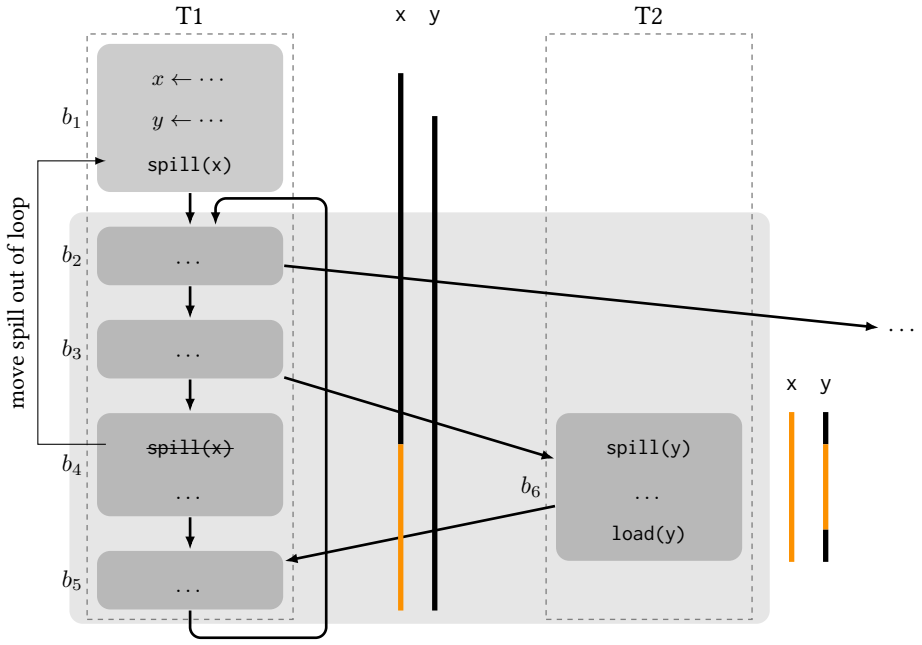
The bottom-up strategy (Section 5.3) uses the $loc_{out}$ and $\varphi_{out}$ information from an already allocated successor of the *last* block in the trace to initialize the current mapping of variables. Note that not all variables must be *live* until the end of a trace. Therefore, we might miss hints for some variables. We experimented with better hinting strategies for the bottom-up allocator, but discarded the idea in favor of faster allocation.

## 6.2 Spill Information Sharing

Since LIR is in SSA form, the value of a variable can be inferred from its single definition point. Due to spilling and spill-position optimization a value might be available in two locations at the same time, i.e., in a register and in a stack-slot. If this is the case at the source of an inter-trace edge, we can exploit it to avoid redundant spill moves. To do so, we inform the allocation strategies that the value is not only available in a register but that there is also a copy on the stack. The linear scan strategy treats intervals with this information as preferred candidates for spilling, since their value is already available in memory so the spill move can be omitted. Similar to inter-trace hints, the trivial trace allocator simply forwards the spill information to its successor. Due to its greedy fashion, the bottom-up strategy does not incorporate enough knowledge to provide spill information. While the allocator could be extended to provide the information, we refrained from doing so.

## 6.3 Known Issue: Spilling in Loop Side-Traces

The non-global scope of trace register allocation is a natural boundary for optimizations. However, in the evaluation of our benchmarks, we did not encounter a situation where this was an issue, except for one benchmark. The issue can occur when spill moves are introduced in a *side-trace* of a loop for variables that were not spilled in the *main-trace*, i.e., in the trace with the loop header. Figure 6.2 shows an example: In trace T1 the allocator is able to move the spill code for $x$ out of the loop (i.e., from $b_4$ to $b_1$). In trace T2, when spilling $y$, we cannot hoist the spill move out of the loop, since the block entering the loop ($b_1$) is not part of T2. This means that we need to execute the spill move inside the loop, every time we enter $b_6$. Doing the spill also in $b_1$ would be preferable, but since register allocation of traces is decoupled, this is not possible. Note that in any case, we cannot remove the load of $y$ at the end of $b_6$.




Spilled ranges are highlighted in orange.

**Figure 6.2:** Spilling in side-traces of a loop

The effect of this issue can be seen in the results of the `sunflow` benchmark in Figure 7.11a. The hot loop of the benchmark contains a `switch` statement where in every branch the register pressure is higher than the number of available registers. The trace allocator creates one long trace containing the loop header and the most likely branch of the switch. In this trace the spills can be moved out of the loop. For the other branches, separate traces are created where the spill moves cannot be hoisted out of the loop. Our investigation showed that in this benchmark the number of executed spill moves is higher with our trace allocator than with the global linear scan allocator.

There are multiple potential solutions to the general problem. We could conservatively insert spill moves for variables that are only used in side traces into the main-trace and try to hoist them out of the loop. Later, we could delete them, if they turn out to be unnecessary. However, this would sacrifice non-globalness of trace register allocation. Another possibility is to apply the *decoupled register allocation* idea, proposed for example by Braun and Hack [2009], Tavares et al. [2011], or Barik et al. [2013]. Their allocator is split into two independent phases. First, spilling decisions are made based on the register demands. The result is a compilation unit for which registers can be allocated without spilling. A similar approach could be applied to trace register allocation. A pre-phase could compute the spill decisions of traces and propagate this information to preceding traces. See Section 10.1.3 for more details on *decoupled register allocation*.

In our specific case, the *stack interval optimization*, which we will describe next, mitigates the above problem. Note, however, it does not solve the problem in general.

## 6.4 Stack Intervals

While further investigating the benchmark that uncovered the *loop side-trace spilling problem*, we noticed that there were many variables that were spilled in the main-trace and not used in the side-trace. Although those variables were first-class spill candidates due to *spill information sharing*, the linear scan strategy first tries to allocate a register for them. To some extent this seems reasonable. The variable is *live* because there is a path to a usage. If the register pressure is low, storing the variable in a register is favorable for its future usage. However, we have to move it from the stack slot to the register. If this happens in a hot loop it has a significant performance overhead. Based on this observation, we decided not to allocate registers for stack intervals, i.e., intervals that are in a stack slot at the beginning of the trace and have no usage in the current trace. It turned out that this heuristic mitigates the issues we were seeing in our early evaluation and no other benchmark is negatively affected by it.

Note that this optimization was explicitly added to the linear scan strategy. After interval building, we remove intervals that are allocated on the stack and have no usages in the current trace from the *unhandled* list and take the stack slot as their location. The bottom-up strategy allocates registers on demand, so it will never try to allocate a register for a variable without a usage, assuming it is in a stack slot at the end of the trace. Also for the trivial trace allocator, this is a non-issue.

# Chapter 7

# Evaluation

In this chapter, we want to verify that our implementation of trace register allocation solves the problems as well as reaches the goals we described in the introduction (Section 1.4 and Section 1.5). To validate our approach, we want to answer the following questions positively.

Can the *trace register allocation* approach ...

**RQ1** ...reach the same *code quality* as a *state-of-the-art* global allocator? [Eisl et al., 2016]

**RQ2** ...be *as fast as* a global allocator for the same quality of allocation? [Eisl et al., 2017]

**RQ3** ...enable *fine-grained* trade-offs between *compile time* and *allocation quality*? [Eisl et al., 2017]

**RQ4** ...reduce the *compilation latency*, i.e., the duration until the result of a compilation is available? [Eisl et al., 2018]

In this chapter we present results and show that the trace register allocation approach can positively answer **RQ1** and **RQ2**. To give more insights on code quality, we also give a detailed evaluation of the inter-trace optimizations, which were presented in Chapter 6.

In the following two chapters on register allocation policies (Chapter 8) and parallel trace register allocation (Chapter 9), we present how we approached **RQ3** and **RQ4**.

All experiments were performed on GraalVM on top of HotSpot (Section 3.2.1). We start with a discussion of the benchmark suites. Afterwards, we compare our trace register allocation approach with the global linear scan approach in terms of allocation quality and compile time.

## 7.1 Benchmarks

To ensure that our approach is applicable to a broad area of workloads we used a variety of common Java benchmark suites. In the following, we provide details about their structure and how we collected the results.

### 7.1.1 SPECjvm2008

The SPECjvm2008[1] benchmark suite consists of a collection of real-world Java applications. Its goal is to benchmark core functionality of a Java Runtime Environment, including the processor and memory subsystems but excluding disk I/O and inter-machine networking. The result metric is *throughput-based*, i.e., an operation is executed repeatedly for a fixed amount of time. The number of performed operations during this time is the score. Therefore, higher numbers reflect a better performance of the system. The harness first warms up the benchmark for 120 seconds to trigger JIT compilation. The warmup is followed by a 240 second interval for the actual measurement. Usually, all benchmarks of SPECjvm2008 are executed in the same JVM instance. This means that the order in which the benchmarks are executed is important since profiling data is shared for all benchmarks. However, different applications exhibit different profiling patterns, so this leads to what we call *profile pollution*, i.e., that the profile does not represent the current—and expected future—behavior of the application. Therefore, we start a new JVM instance for every benchmark. We believe that this is a more realistic methodology.

The SPECjvm2008 suite also incorporates a *startup* mode. Its goal is—according to the vendor—to benchmark the "user experience of Java." In this mode, the benchmark workloads are only executed once without any warmup. The duration of this run is used to calculate a throughput value. For most benchmarks, this interval is too short to even trigger JIT compilation, so the significance of this mode for evaluating our approach is very limited. We therefore do not present *startup* numbers.

### 7.1.2 SPECjbb2015

SPECjbb2015[2] is a server benchmark suite for Java. In contrast to the SPECjvm2008, it does not consist of multiple independent benchmarks but executes a single business application. The benchmark models the components of the IT infrastructure of a sales company. The different

[1] https://www.spec.org/jvm2008/
[2] https://www.spec.org/jbb2015/

components (controller, backend, transactors) can either be executed on a single JVM or on multiple instances to test outbound communication performance [SPEC, 2017]. Since we are only interested in compiler performance, we are only evaluating the single-JVM mode. The harness provides two metrics: the `critical` score, which relates to response time, and the `max` value, which measures throughput.

### 7.1.3 DaCapo

The DaCapo benchmark suite was proposed by Blackburn et al. [2006] to tackle certain shortcomings of the SPEC suites such as more complex code patterns, a more diverse object behavior, and more demanding memory requirements.[3] We used the 9.12 version of DaCapo for our experiments. Similar to SPECjvm2008, DaCapo consists of a set of real-world applications. The benchmarks are iteration-based, meaning that they run the same workload for a predefined number of times in order to warm up the virtual machine. We chose this number high enough to make sure that all important methods are compiled. Since the work performed in one iteration varies considerably from benchmark to benchmark, the iteration numbers range from $5$ to $120$. For getting more stable results, the final performance result for a benchmark is the *average* of the time for the last $n$ iteration, where $n$ is a number between $5$ and $10$, depending on the total number of iterations. Since the result is a *time* value, lower numbers mean better performance (in contrast to the SPEC suites). As for SPECjvm2008, every benchmark is executed in a *fresh* virtual machine.

Note that, due to Java 8 compatibility issues, we excluded the `eclipse`, `tomcat`,[4] `tradebeans`, and `tradesoap` benchmarks. To minimize the effect of disk I/O, we executed the benchmarks in a fresh directory on a ram disk. For some benchmarks, for instance `lusearch`, `luindex`, `h2`, or `batik`, this is necessary to get stable results.

### 7.1.4 Scala-DaCapo

There is a multitude of programming languages that are compiled to Java Bytecode and executed on JVMs. Examples include Clojure, Groovy, JRuby, Jython, and Scala. However, with the exception of the `jython` benchmark in DaCapo, those languages are not represented by the benchmark suites described above. To cope with this, at least with respect to Scala, Sewe et al. [2011] proposed the Scala-DaCapo benchmark suite. They show that Scala workloads differ significantly

[3] Note that they compared against older versions of the SPEC benchmark suites, so some points of criticism might no longer be valid. Also, we are using a newer version of DaCapo than the one analyzed in the paper by Blackburn et al. [2006].

[4] `https://bugs.openjdk.java.net/browse/JDK-8155588`

from Java, for example in term of *call site polymorphism* or the number of *boxed values*. This poses different challenges to the JVM and especially to the JIT compiler, as for example shown by Stadler et al. [2013] in the case of Graal. Scala-DaCapo uses the same benchmarking harness as DaCapo, which is why we often present the results together.

## 7.2 Configurations

To answer **RQ1** and **RQ2**, we compared our approach against the global linear scan allocation (GlobalLSRA) that is the default in Graal. We used the TraceLSRA configuration, i.e., only using the linear scan and the trivial trace strategy, since this is the configuration that achieves the best allocation quality. In order to positively answer **RQ1** and **RQ2**, the TraceLSRA configuration needs to perform at least as good as GlobalLSRA. For some benchmarks we also included the BottomUp configuration, where we only used the bottom-up and the trivial trace strategy.

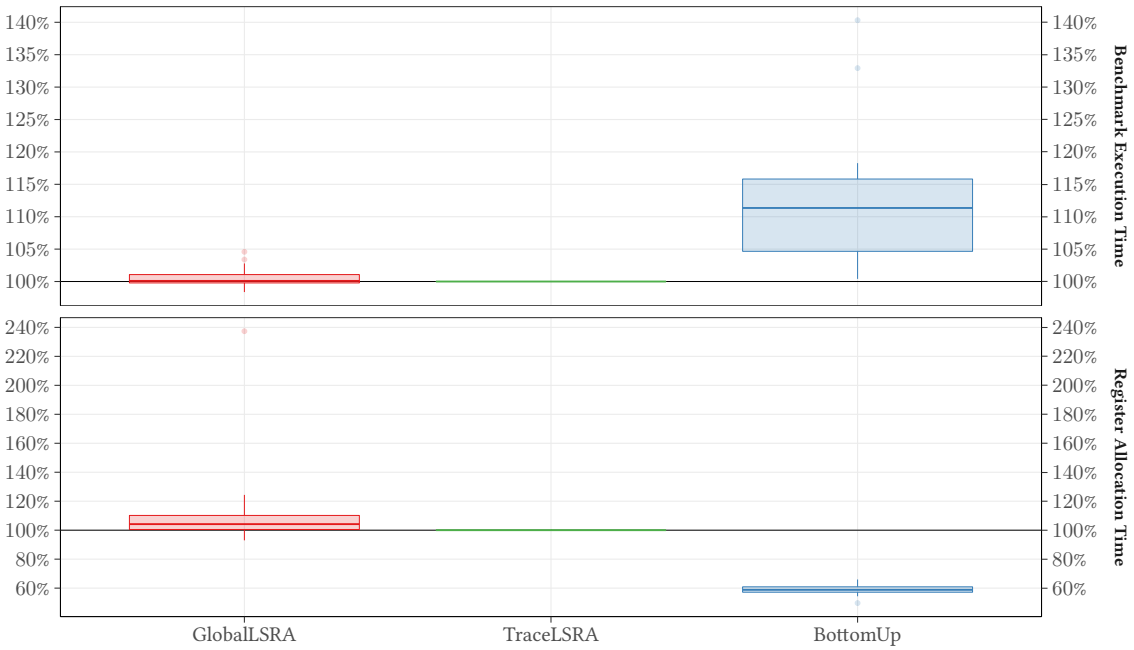
## 7.3 Peak Performance/Allocation Quality

Let us first focus on **RQ1**, i.e., whether the trace register allocator can compete with a state-of-the-art global register allocator in terms of allocation quality.

### 7.3.1 DaCapo and Scala-DaCapo

**AMD64** The top half of Figure 7.1 depicts the composite performance results for DaCapo and Scala-DaCapo on AMD64. The experiments were performed on our X3-2 cluster. The details of the machine are given in Section C.1 in the appendix. For every experiment, we randomly selected a node from the cluster to execute a benchmark suite (DaCapo or Scala-DaCapo) with a single configuration. For every benchmark, we started a new Java VM with an initial and maximum heap size of 8GB. To avoid distortion due to node switching, we fixed the CPU and the memory of the JVM to a single NUMA node using the `hwloc-bind`[5] utility. Figure 7.1 shows the composite results for the benchmark suites, i.e., every observation in the box plot [Tukey, 1977] (e.g., a dot) represents the mean result of a single benchmark (e.g., the mean of all `lusearch` runs with the TraceLSRA configuration). A *per-benchmark* evaluation is given in Figure 7.2, where an observation represents a single benchmark run. The results are all relative to TraceLSRA. We used the `git`[6] revision `b398d21c7c21` of Graal for this evaluation.[7]

[5] hwloc-bind(1) — Linux man page: `https://linux.die.net/man/1/hwloc-bind`
[6] git(1) — Linux man page: `https://linux.die.net/man/1/git`
[7] `https://github.com/zapster/graal`



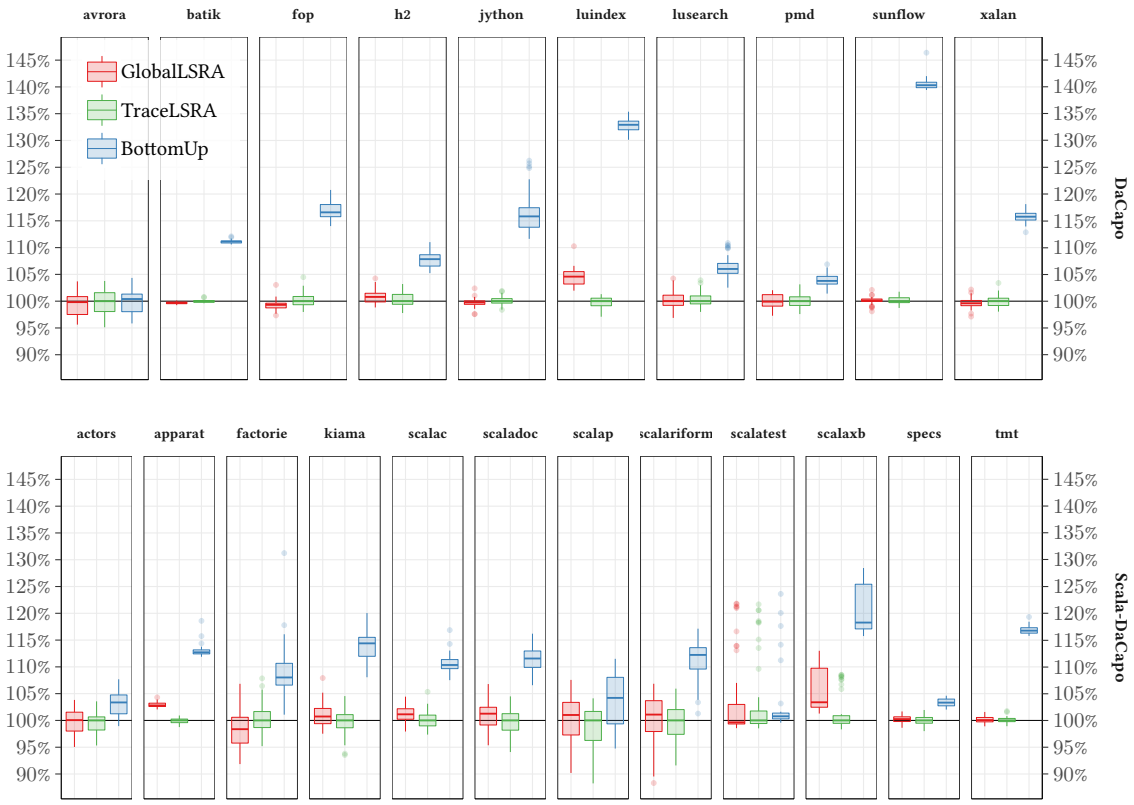

Benchmark execution time (allocation quality) and register allocation time. Values relative to TraceLSRA median. Every observation in the box plot, e.g., a dot, is the median result of a benchmark, e.g., `lusearch`. For every benchmark we collected at least 38 results. The experiments were conducted with the `git` revision `b398d21c7c21` of Graal. Lower is better ↓.

**Figure 7.1:** Composite results for DaCapo and Scala-DaCapo on AMD64

The figures suggest that there is no significant performance difference between the GlobalLSRA and the TraceLSRA configuration. GlobalLSRA performs worst on the `luindex` benchmark from DaCapo, where it is 5% slower than TraceLSRA on average. On the other hand, GlobalLSRA is 2% faster on the `factorie` benchmark from Scala-DaCapo.

The BottomUp configuration shows a significant difference to TraceLSRA. The median slowdown is 11%. The worst case is the `sunflow` benchmark from DaCapo, where the gap is as big as 40%. Some benchmarks are insensitive to register allocation. For example, the `avrora` benchmark from DaCapo shows hardly any difference to between the configurations.

**SPARC** Register allocation is highly influenced by the underlying processor. Also the operating system plays a role since it affects the calling conventions and other register constraints. To avoid optimizing towards a single processor/operating system pair (e.g., AMD64 and Linux) we also verified our approach on a SPARC processor running a Solaris operating system. The details of this setup are presented in Appendix C.3. Since SPARC is a RISC (reduced instruction set computer [Hennessy and Patterson, 2003]), it imposes different challenges on a register allocator than an AMD64, which is a CISC (complex instruction set computer [Hennessy and Patterson, 2003]). RISC architectures are usually load/store architectures. They first *load* operands from memory

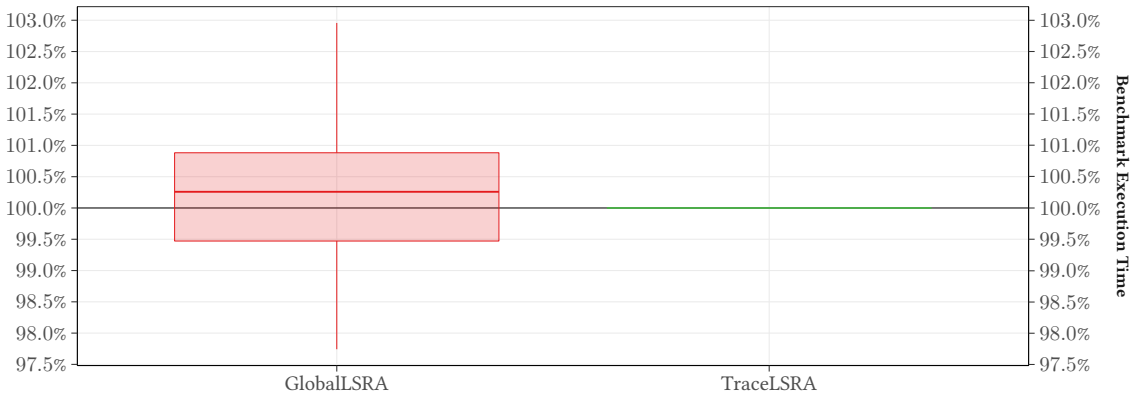



Same data as in Figure 7.1. Values relative to TraceLSRA median. Lower is better ↓.

**Figure 7.2:** Peak performance of individual benchmarks from (Scala-)DaCapo on AMD64

to a register, perform an operation on them, and then *store* the results back. CISC, in contrast, allow direct addressing of memory in many operations. Consequently, RISCs usually have more registers than CISCs, which makes spilling less likely. On the other hand, on a CISC, reloading a spilled value is often not necessary, because instructions can directly use it from memory.

Figure 7.3 shows the results for DaCapo and Scala-DaCapo on SPARC. The conclusion is the same as on AMD64. Trace register allocation achieves the same performance as GlobalLSRA. It is worth noting, that the linear scan implementation (and also our trace-base linear scan implementation) was tuned for AMD64 [Wimmer and Mössenböck, 2005]. We think that an investigation on whether RISC architectures offer further opportunities for trace register allocation is interesting future work.




Values relative to TraceLSRA median. Same methodology as in Figure 7.1. For every benchmark we collected at least 15 results. The experiments were conducted with the `git` revision `3e8cb34059d2` of Graal. Lower is better ↓.

**Figure 7.3:** Composite peak performance results for DaCapo and Scala-DaCapo on SPARC

### 7.3.2 SPECjvm2008

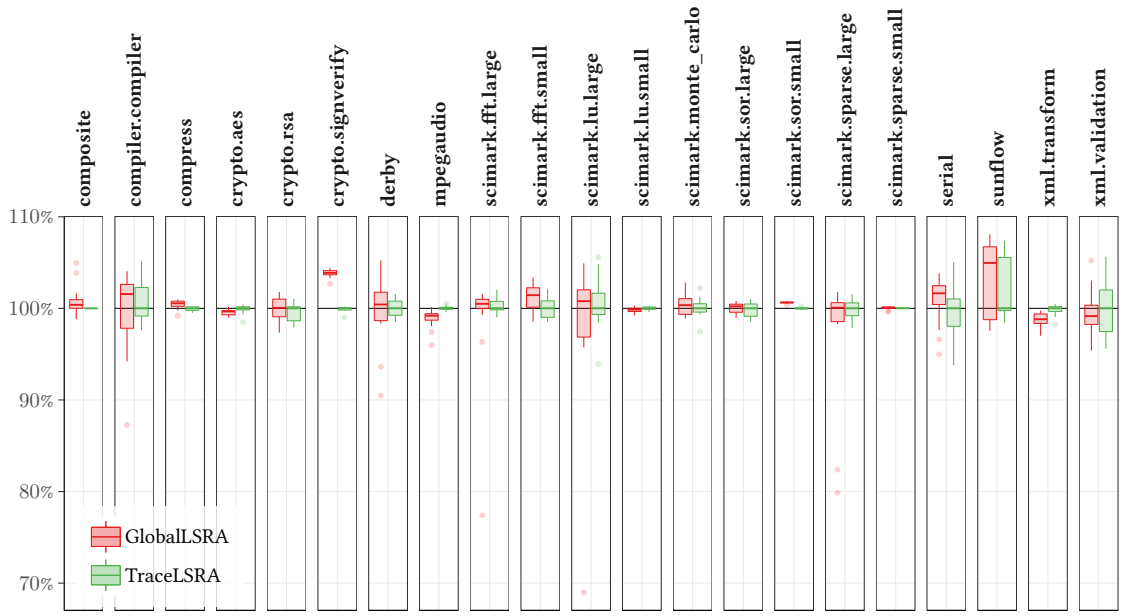
The AMD64 results for SPECjvm2008 are summarized in Figure 7.4. The experiments were performed on our X5-2 cluster (see Appendix C.2). Again, the results for GlobalLSRA are in the same range as for the TraceLSRA configuration. They span from 1.2% slower to 5% faster compared to TraceLSRA. One interesting observation is that the difference on the `crypto.signverify` is significant in favor of GlobalLSRA. It seems that the trace register allocator always chooses a less optimal allocation than the global linear scan approach.

### 7.3.3 SPECjbb2015

The results for SPECjbb2015 in Figure 7.5 show no significant difference between GlobalLSRA and TraceLSRA (again on X5-2). In our experiments, it appeared as if GlobalLSRA performs better on the `critical` metric (responsiveness) and TraceLSRA on `max` (throughput). However, the results vary too much to draw a definite conclusion.

### 7.3.4 Answering RQ1

The benchmark results let us conclude that **RQ1** can be answered positively, i.e., that trace register allocation can achieve the same allocation quality as a state-of-the-art global allocator.



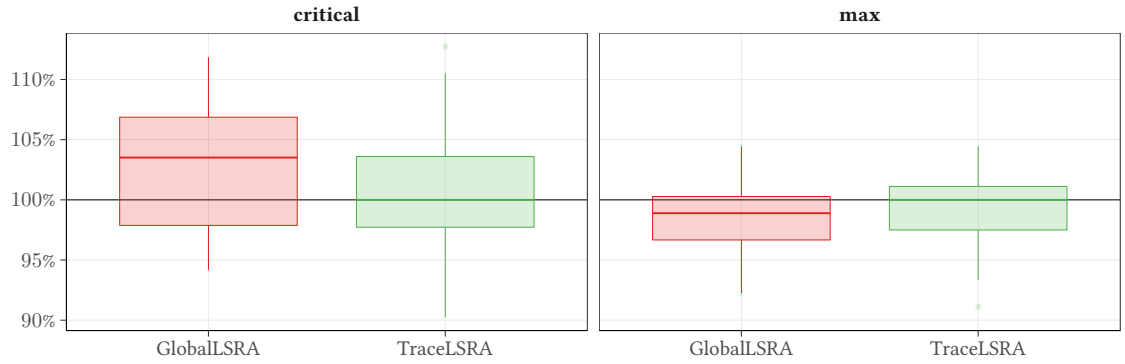

Composite result on the very left. Values relative to TraceLSRA median. For every benchmark we collected at least 20 results. The experiments were conducted with the `git` revision `5e7a1574d37b` of Graal. Higher is better ↑

**Figure 7.4:** Peak performance results for SPECjvm2008 on AMD64

## 7.4 Compile Time

Defining a meaningful compile time metric is inherently more difficult for a dynamic compilation system than for a static compiler. On the one hand, the compilation and the execution of every benchmark are intertwined. Compile time is an integral part of the run time. On the other hand, experiments are harder to reproduce, since the executed machine code can be different for every run after recompilation and depends on non-deterministic factors such as the order in which compilation happens.

The meta-circular aspect of the GraalVM adds another layer of challenges to the problem. Since the compiler itself is subject to compilation, as it is implemented in Java, changes in the compiler influence not only the generated machine code, but also the time it takes to translate the compiler itself (compiler compile time). To minimize this effect, Graal avoids self-compilation, i.e., all methods in the Java packages `jdk.vm.ci` and `org.graalvm.compiler` are compiled by the HotSpot client compiler and not by the Graal compiler. This on the other hand means that Graal code is less optimized which again affects compile time.




We collected at least 20 results for every configuration. The experiments were conducted with the `git` revision `5e7a1574d37b` of Graal. Higher is better ↑.

**Figure 7.5:** Peak performance results for SPECjbb2015 on AMD64

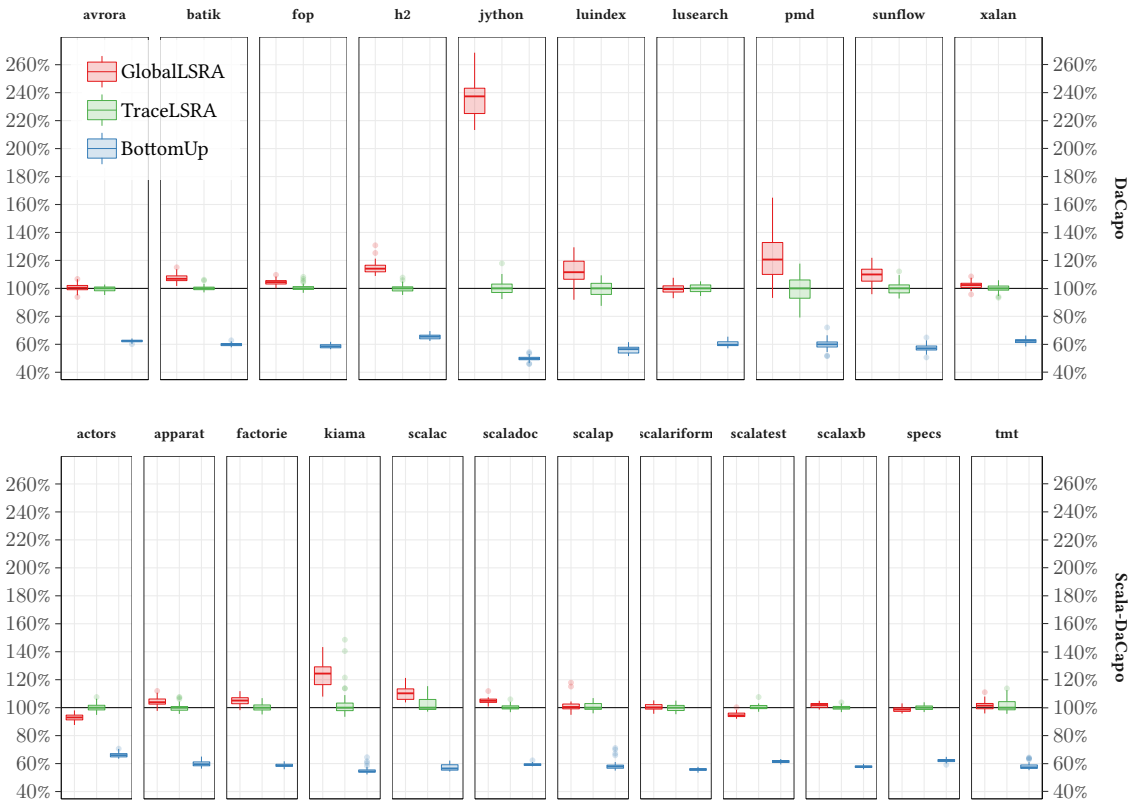
We need to accept that compile time results in our system are influenced by the surrounding context and that not all differences are necessarily caused by a change in the register allocator. Not all effects that we see in the evaluation of a specific implementation must also happen in a different context. *Project Metropolis* [OpenJDK, 2018] could mitigate these shortcomings. Its goal is to *ahead-of-time compile* the Graal compiler into a shared library so that it is not compiled during application run time. Unfortunately, at the time of writing, the project was just started. Repeating the compile time experiments in this deployment is highly interesting future work.

For our compile time metric we measure the *CPU time* spend on register allocation. We also refer to it as *register allocation time*. *CPU time* is the time the compiler thread did actual work and was not waiting, for example, because of *preemption*. Ignoring wait time is fair, since compilation of a method is single threaded,[8] so there is no waiting for resources. In the case of trace register allocation, register allocation time includes all phases of the framework: trace building, global liveness analysis, global data-flow analysis and the actual processing of the traces.

We only present compile time results for DaCapo and Scala-DaCapo since, due to the harness, the results of (most) DaCapo-style benchmarks are more stable than the results of the SPEC benchmarks. Also, we present only results for AMD64. We did experiments on SPARC, but the number of machines is limited (Appendix C.3). Anyhow, since SPARC offers more registers, the pressure on the register allocator is lower than on AMD64.

The lower half of Figure 7.1 depicts the composite register allocation time results for DaCapo and Scala-DaCapo on AMD64. Figure 7.6 shows the results for the individual benchmarks. In summary (Figure 7.1), GlobalLSRA and TraceLSRA operate in the same range, although our implementation seems to be faster by a median difference of 4%. Figure 7.6 indeed shows that

[8] Except when it is not, see Chapter 9.

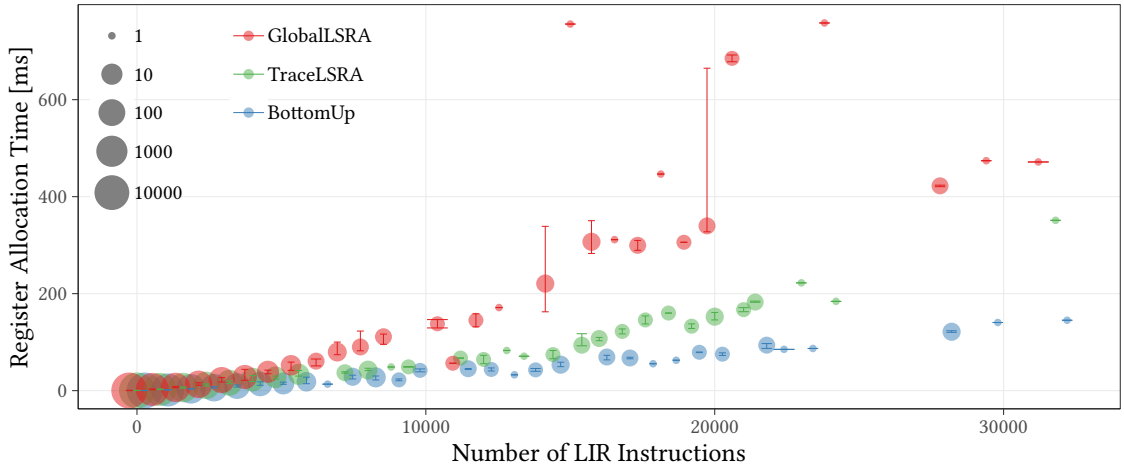



Same data as in Figure 7.1. Values relative to TraceLSRA median. Lower is better ↓

**Figure 7.6:** Register allocation time of individual benchmarks from (Scala-)DaCapo

trace register allocation outperforms GlobalLSRA on a number of benchmarks. Most notable is `jython`, where GlobalLSRA takes more than twice as long as TraceLSRA. The `jython` benchmark produces Java bytecodes dynamically and the generated methods tend to become significantly larger than typical Java or Scala methods. As we will show in Section 7.4.1, trace register allocation performs especially well on large methods. It is only fair to note that there are also cases where GlobalLSRA is faster than TraceLSRA. On the `actors` benchmark from Scala-DaCapo, for example, GlobalLSRA is 7% faster.

The BottomUp configuration is significantly faster than TraceLSRA in all cases. The median speedup is 41%. Again, `jython` stands out. On this benchmark, the BottomUp configuration is 50% faster than TraceLSRA.



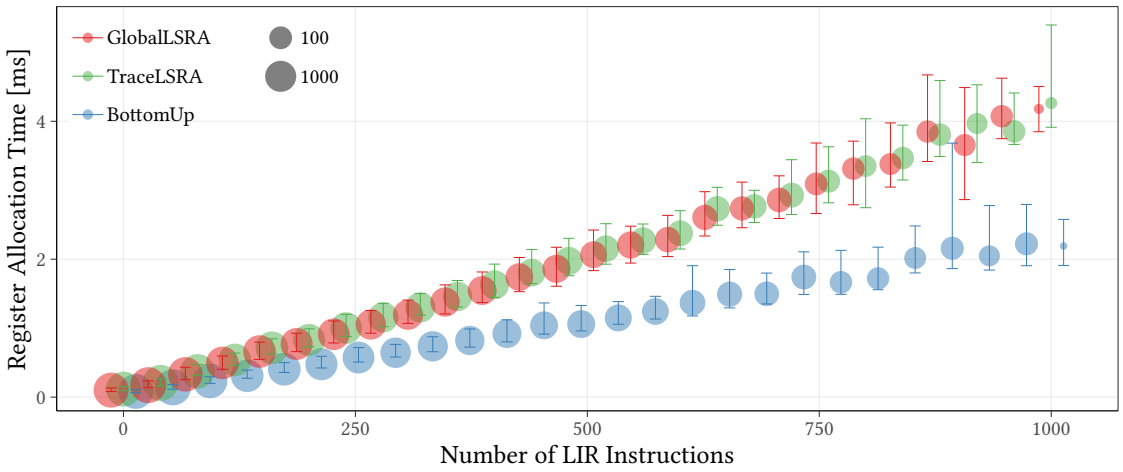

Register allocation time in milliseconds vs. number of LIR instructions for DaCapo and Scala-DaCapo on AMD64. Each dots denotes a bucket of 800 instruction each. The $y$-value is the median register allocation time. The size of the dot indicates the number of methods in the bucket. The error bars represent the lower and upper quartiles. Lower is better ↓.

**Figure 7.7:** Register allocation time vs. LIR instructions for DaCapo and Scala-DaCapo

### 7.4.1 Compile Time per Method

To gain more insight on the compile-time behavior of the various register allocation approaches, we investigated the register allocation time on a *per-method* basis. Figure 7.7 shows the relationship between of the number of LIR instructions of a method and its register allocation time. Note that the figure represents the results of all DaCapo and Scala-DaCapo benchmarks. Plotting all 79155 compilations is not feasible, so we group the method sizes into buckets of 800 instructions and print a circle for the median register allocation time of this group. The size of the circles indicates the number of methods in a bucket (logarithmic scale). The plot suggests that TraceLSRA performs particularly well for large method, compared to GlobalLSRA. As expected, BottomUp is faster than the other configurations.

Since the method sizes are not evenly distributed, we show the results for small methods in Figure 7.8. In this case, the TraceLSRA has no advantage over the GlobalLSRA configuration. The BottomUp allocator, however, continues to outperform the other approaches in terms of register allocation time.




Register allocation time vs. number of LIR instructions for methods with less than 1000 LIR instructions. Same methodology as in Figure 7.7. Bucket width is 40. Lower is better ↓.

**Figure 7.8:** Register allocation time vs. LIR instructions for (Scala-)DaCapo (small methods)

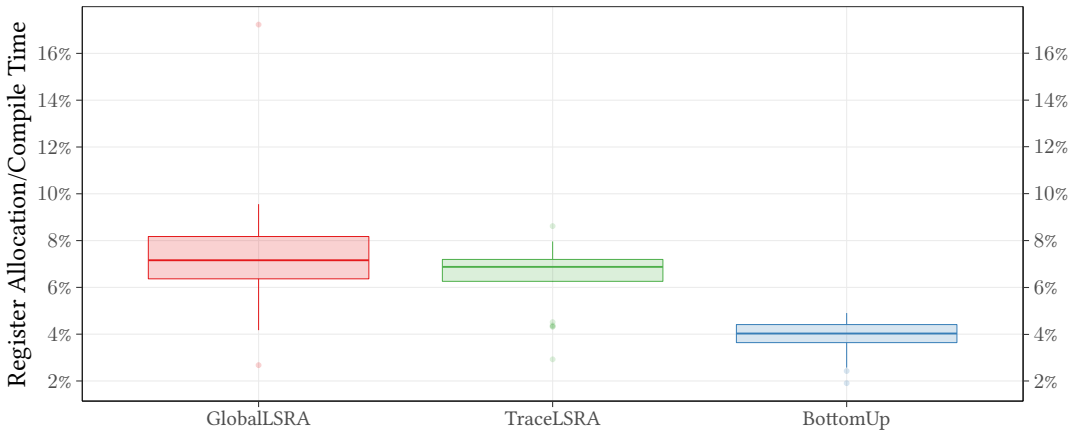
### 7.4.2 Overall Compile Time

The compile time results that we presented so far only show register allocation time. The ultimate goal, however, is to reduce the *overall compile time*, i.e., the time required for all phases, not only for the register allocator.

The Graal compiler is currently tuned for *peak performance*. The majority of the compile time is spent in the front end on code optimizations. Figure 7.9 shows how much of the overall compile time can be accounted to register allocation. With TraceLSRA, 7% of the time is spend on register allocation, while in the BottomUp configuration this number goes down to 4%. Repeating this experiments with a Graal configuration that is tuned towards compile time rather than towards peak performance, would show more potential for saving compile time by optimizing the register allocation.

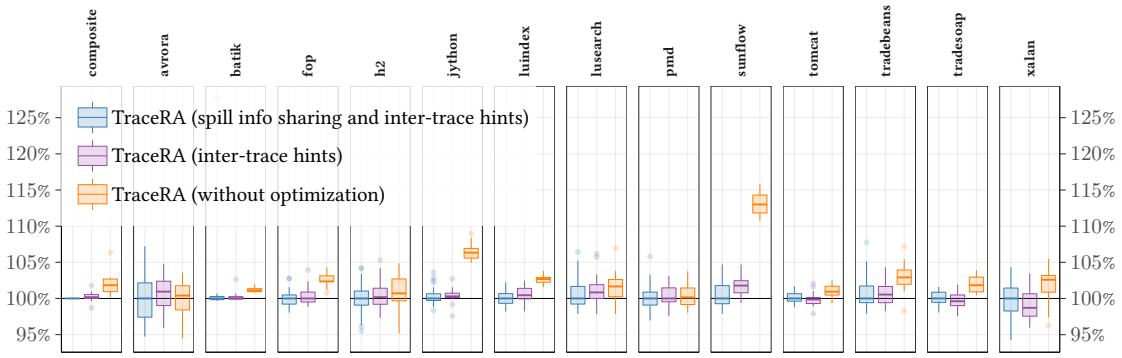
### 7.4.3 Answering RQ2

As for peak performance, the trace register allocation approach can compete with state-of-the art global register allocators. Thus we can affirm **RQ2**.




Overall compile time on AMD64. Same methodology as in Figure 7.1. Lower is better ↓.

**Figure 7.9:** Share of register allocation time in the overall compile time for (Scala-)DaCapo

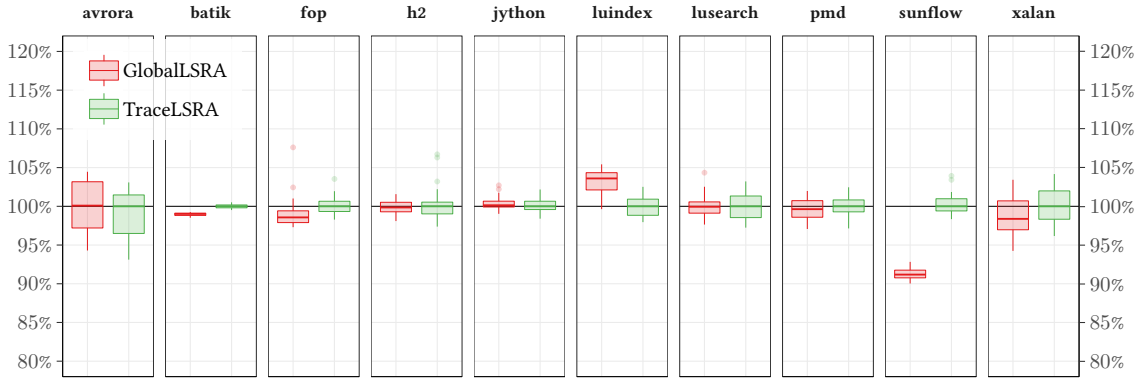



Performance impact of inter-trace hints and spill information sharing on DaCapo on AMD64. Numbers relative to the performance of TraceLSRA with inter-trace hints and spill information sharing. Lower is better ↓.

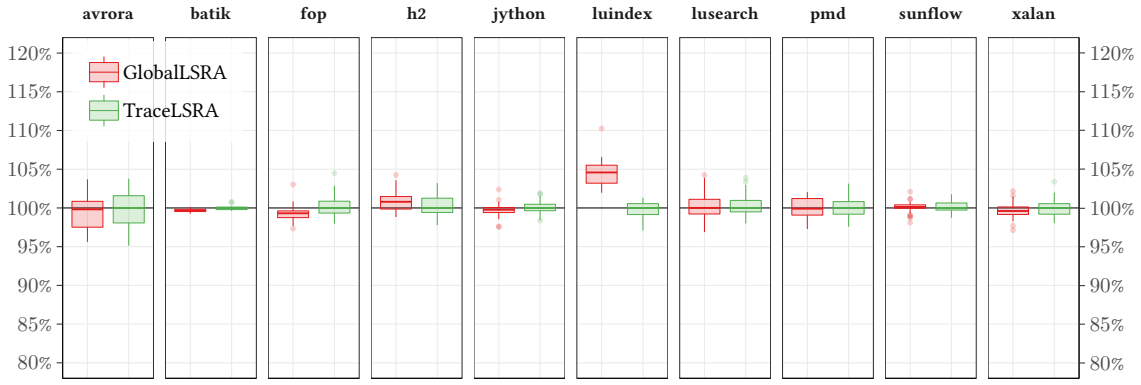
**Figure 7.10:** Influence of inter-trace hints and spill information sharing on peak performance

## 7.5 Inter-trace Optimizations

Since the time we did our original experiments on *inter-trace optimizations* as described in Chapter 6 [Eisl et al., 2016], the trace register allocation implementation was further improved. Most optimizations where algorithmic improvements, i.e., doing the same work in less time. However, results for the *stack interval* optimization described in Section 6.4 where not yet published. Yet, we decided to use the original measurements [Eisl et al., 2016] in order to be consistent to this paper and because the results did not change much. Also, some optimizations are now mandatory and can no longer be deactivated easily. The new changes do not invalidate the arguments presented in the above-mentioned chapters but only the relation to global linear scan. Therefore, the numbers presented in this chapter are taken from [Eisl et al., 2016]. We used the `git` revision `a563a1d51507` of Graal for our evaluation.




(a) Without stack interval optimization




(b) With stack interval optimization

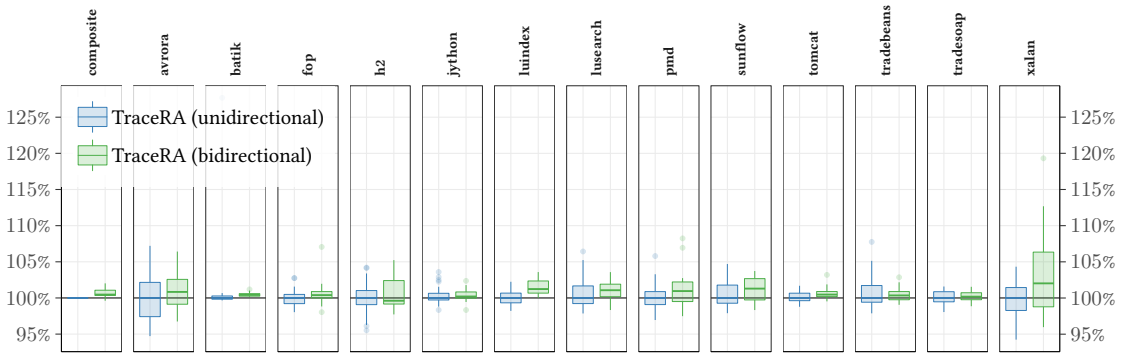
The influence of *stack interval optimization* on DaCapo. Same methodology as in Figure 7.2. Note the impact on the `sunflow` benchmark. Lower is better ↓.

**Figure 7.11:** Peak performance impact of the stack interval optimization on DaCapo

Figure 7.10 shows our evaluation of the inter-trace optimizations of DaCapo on AMD64.[9] The figure clearly shows that *inter-trace hints* are the most influential optimization. The most significant change is on `sunflow` with about 10% improvement when enabled. The core of the `sunflow` benchmark is a hot loop with multiple equally hot branches (see also Section 6.3). Since every branch will be allocated in a different trace, sharing allocation information is utterly important. Also `jython` is highly affected by inter-trace hints: Turning them off, results in a 5% degradation.

The influence of *spill information sharing* is less significant. The `luindex` and the `sunflow` benchmarks seem to be the most influenced. However, spill information sharing is important for expanding the applicability of the *stack intervals* optimization.

[9] Note that this evaluation was performed on an older version of the HotSpot VM where `tradesoap` and `tradebeans` still worked.




Results for AMD64. Same methodology as in Figure 7.2. Lower is better ↓.

**Figure 7.12:** Unidirectional vs. bidirectional trace builder w.r.t. peak performance of DaCapo

The *stack interval optimization* was added after the other inter-trace optimizations. Figure 7.11 shows its impact on the peak performance of DaCapo. In the version without the optimization (Figure 7.11a) there is an outlier in terms of peak performance, namely `sunflow` where GlobalLSRA is about 9% faster than TraceLSRA. The benchmark in running into the loop spilling problem that was discussed in Section 6.3. In Figure 7.11a, which includes the stack intervals optimization, this outlier is gone. The other benchmarks (including Scala-DaCapo) are not significantly affected. However, note that the issue is not directly solved by the optimization but only mitigated.

## 7.6 Trace Builder Evaluation

Figure 7.12 compares the *unidirectional* and the *bidirectional* trace building algorithms. The results were again conducted for our first evaluation [Eisl et al., 2016]. The numbers suggest that they both perform about equally well. We decided to use the *unidirectional* trace builder by default, since it is the simpler algorithm and it follows our *focus on the common case* philosophy more closely.

# Chapter 8

# Trace Register Allocation Policies

In Chapter 5 we proposed three allocation strategies: The linear scan strategy (Section 5.1) to produce good code, the bottom-up strategy (Section 5.3) for fast allocation, and the trivial trace strategy (Section 5.2) for special traces. In this chapter we focus on the question *when* to use *which* strategy. The goal is to gain a fine-grained control over the trade-off between compile-time and peak-performance. Doing so allows us to answer **RQ3** (Chapter 7).

Intuitively, we want to use linear scan for the *important* traces and bottom-up for the others and thus, reduce allocation time. Of course, *trivial traces* should be processed by the trivial trace allocator unconditionally, since it is *fast* and *optimal*. To apply this in practice, we need to define our notion of importance. We evaluated a case study of 8 decision heuristics, so-called *allocation policies* [Eisl et al., 2017].

First, we identified *properties*, which allow us to characterize a trace. Based on these properties, we developed policies to select either the linear scan, the bottom-up, or the trivial allocator. The list of properties and policies is non-exhaustive.

## 8.1 Properties

Our allocation policies are based on properties of basic blocks, traces, the complete compilation unit, or a combination of them.

### 8.1.1 Block Properties

A trace consists of a sequence of basic blocks. For every block $b$ we know its *relative execution frequency*, which we denote as $freq(b)$. It is a *real number* estimating how frequently this block is executed per invocation of the enclosing method. A value of $0.5$ means that the block is executed in $50\%$ of all invocations. For blocks inside of loops this value can be larger than $1$. For example, if a loop header is entered with a probability of $1$, a frequency of $10$ indicates a loop iteration count of $10$. Note that these numbers are relative to the invocation count of the enclosing method. Therefore, the frequency of the method entry block is always $1$. We cannot infer absolute execution counts from these numbers. The block frequencies are calculated from branch profiles collected by the virtual machine in previous executions of the compilation unit.

Another block metric is the *loop nesting level*, or $loopDepth(b)$. It indicates on which loop nesting level this block occurs. However, this metric can be misleading since not all branches inside a loop are equally likely. It should be used as a structural indicator only.

Due to the *Global Liveness Analysis*, described in Section 4.3, we can also consider the $live_{in}$ and $live_{out}$ sets as block properties, i.e., the variables that are live at the beginning and the end of the block. More live variables increase the likelihood of spilling.

### 8.1.2 Trace Properties

The properties of the blocks in a trace can be aggregated to define properties for the trace. For example, the *frequency* of a trace can be defined as the *maximum* frequency of the blocks in the trace.

Another important property of a trace is its *triviality*, i.e., the fact that a trace consists of a single block containing just a jump instruction. It determines whether or not the framework can use the trivial trace allocator.

We also consider the *trace building order*, denoted by $id(trace)$, as a trace property. The trace building algorithms, as described in Section 4.1, construct important traces first. That means that a trace with a lower number is generally more performance-critical than one with a higher number.

### 8.1.3 Compilation Unit Properties

For compilation units we can apply the same aggregation techniques as for traces. We use compilation unit properties to set trace properties into relation. For example, the maximum block frequency of a trace vs. the maximum block frequency of the whole compilation unit. We also exploit structural properties of a compilation unit to switch between different sub-policies. For instance, if a method contains a loop we might want to choose a different decision model than for methods without loops.

### 8.1.4 Aggregation of Properties

As outlined above, we aggregate the block properties to calculate new metrics for traces of the compilation unit. We consider different aggregation functions including *maximum*, *minimum*, *sum*, *average*, and *count*.

## 8.2 Policies

We developed a set of 8 *allocation policies*, based on the identified properties. A policy is a decision function that selects an allocation strategy for a given trace.

For trivial traces, we always use the trivial trace allocator. For non-trivial traces, we have to decide whether to use the trace-based linear scan or the bottom-up approach. We describe this decision as a *hotness condition*. If the condition is *true* the trace is considered important, i.e., we use the linear scan approach for it.

In the remainder of this section, $trace$ refers to the trace for which we want to choose a strategy. We use the term $method$ to describe the set of all blocks of the method (compilation unit).

**TraceLSRA** This policy uses the linear scan strategy for all traces that are not *trivial*.

**BottomUp** The BottomUp policy uses the bottom-up strategy for all traces that are not *trivial*.[1]

**Ratio** The Ratio policy uses linear scan for a fixed fraction $p$ of the traces.

$$id(trace) \leq |traces| \times p$$

Since traces are processed in trace-building order (i.e., in the order of their importance), a fraction of $p = 0.5$ means that the first $50\%$ of the created traces (i.e., those with an $id$ less or equal to $|traces| \times 0.5$) is allocated with linear scan (or the trivial allocator).

**Budget** The Budget policy is a budget-based approach. The idea is to allocate traces with the linear scan strategy in trace-building order until we run out of budget.

$$\left( \sum_{\substack{t \in traces \\ id(t) < id(trace)}} \sum_{b \in t} freq(b) \right) < \left( \sum_{b \in method} freq(b) \right) \times p$$

The cost function is the sum of the block frequencies of all traces that have already been allocated. The budget is a fraction of the total sum of the block frequencies in the compilation unit.

**Loop** The Loop policy uses the linear scan strategy for all traces that contain at least one block that is in a loop.

$$HasLoop(trace) \vee \neg HasLoop(method)$$

where $HasLoop(blocks)$ is defined as:

$$\exists\, b \in blocks \text{ where } loopDepth(b) > 0$$

[1] Due to implementation reasons, there is one exception to this rule, namely traces with edges to *compiled exception handlers*. These edges require slightly different handling. Graal assumes that the *framestate* at the instruction that causes the exception, e.g., a *call*, is the same as at the beginning of the exception handler. In other words, we are not allowed to insert moves between the *throwing instruction* and the end of the block. The linear scan implementation in Graal guarantees this by design. The bottom-up allocator, however, does not. It could be easily implemented in the bottom-up allocator, but it would make the algorithm more complicated. Since exceptions in Graal are usually handled via deoptimization, this case is uncommon. To keep the implementation simple, we decided to ignore this special case and fall back to the linear scan strategy if it occurs. The entry for the BottomUp policy in Figure 8.2 shows that the fraction of *linear-scan-compiled* traces is indeed marginal ($\sim$0.3% as depicted in Table 8.1 in the evaluation).

The idea is that we consider loops to be performance-critical, so we want to find a good allocation for them. In addition to that, linear scan is used if the whole compilation unit does not contain a loop at all. The rationale behind this is that the virtual machine compiles only methods which either exceed a certain invocation threshold or a loop back edge threshold. If a method without a loop is queued for compilation, the runtime did so due to the invocation count only. This means that the method was called often enough to be considered important.

**LoopBudget** This policy combines the Loop policy with the Budget policy. Instead of using linear scan for all compilation units without loops, we apply the MaxFreq condition.

$$HasLoop(trace) \vee \big(\neg HasLoop(method) \wedge \textsc{Budget}(trace)\big)$$

The resulting policy can decrease compile time compared to the Loop policy since fewer traces are allocated with linear scan. Nevertheless, loop traces are still prioritized.

**MaxFreq** The MaxFreq policy considers a trace important if the maximum execution frequency of the blocks in the trace is greater than a fraction $p$ of the maximum frequency of the blocks in the compilation unit.

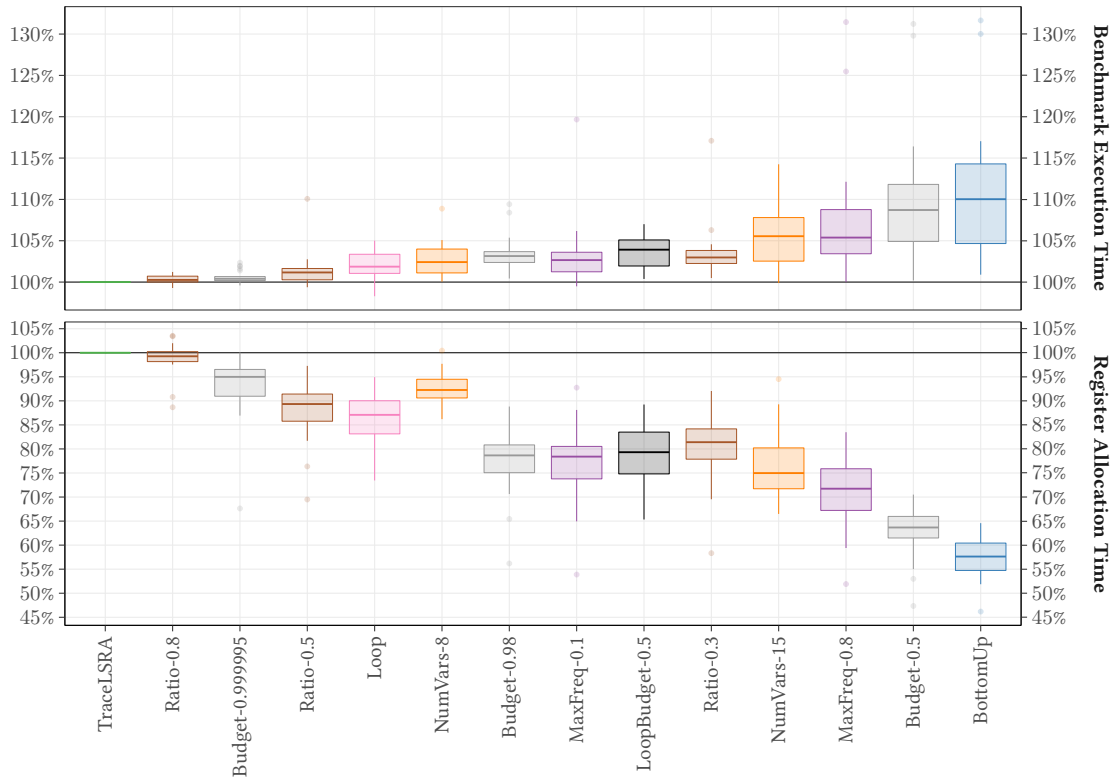
$$\max_{b_t \in trace} freq(b_t) > \max_{b_m \in method} freq(b_m) \times p$$

Only traces with high-frequency blocks are allocated with the linear scan strategy since these traces are most critical for performance. For example, if $p = 0.8$, a trace is compiled with the linear scan allocator if its frequency is larger than $0.8\times$ the frequency of the most frequent block of the method.

**NumVars** The NumVars policy uses linear scan for all traces where the maximum number of live variables at block boundaries exceeds a certain threshold $p$.

$$\max_{b \in trace} \Big( \max \big( |live_{in}(b)|, |live_{out}(b)| \big) \Big) > p$$

The idea is that traces with a higher number of live variables are more likely to require spilling. The spilling mechanism in the linear scan strategy leads to better code than the spilling mechanism in the bottom-up allocator. On the other hand, if no spilling is needed the bottom-up allocator produces code of similar quality as the linear scan allocator but in shorter time.




Evaluation of register allocation policies for DaCapo and Scala-DaCapo on AMD64. Values relative to TraceLSRA median. Same methodology as in Figure 7.2. Lower is better ↓.

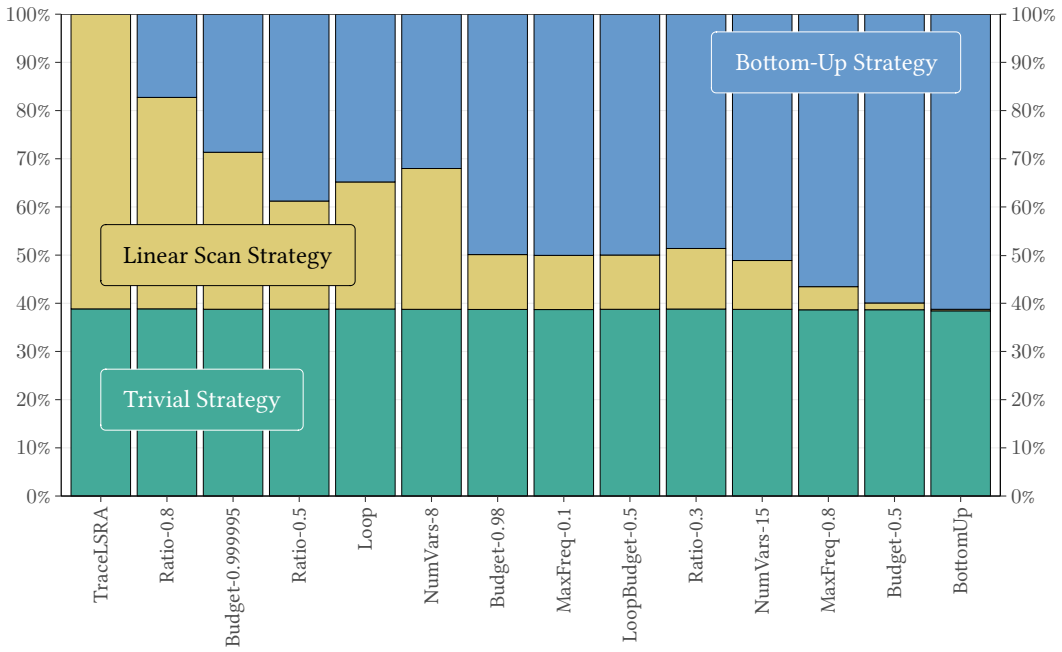
**Figure 8.1:** Peak performance and register allocation time of various allocation policies

## 8.3 Evaluation

The goal of this evaluation is to answer **RQ3** and to support our claim that selective trace-based register allocation is an appropriate approach for controlling the trade-off between compile time and peak performance on a fine-grained level. To this end, we study the impact of the 8 allocation policies discussed in the last section. For policies with parameters we compare multiple values to further highlight the flexibility of our approach. In total, we selected 14 configurations as case study to supports our claim. Our experiments were performed using `git` revision `f5cad2eda111`.

We performed the experiments on the X3-2 cluster (see Appendix C.1) with at least 30 repetitions. The baseline is again the trace-based linear scan allocator, denoted by TraceLSRA. We show the numbers relative to the baseline of a given benchmark.

Figure 8.1 shows the total peak performance and the register allocation time for various allocation policies relative to the trace-based linear scan. For measuring the register allocation time, we included all compilations of the benchmarks, including those in warm-up iterations, since the




The distribution is calculated per benchmark. The figure shows the mean over all benchmarks.

**Figure 8.2:** Distribution of the allocation strategy per policy

peak performance of the last iteration depends on all these compilations. Figure 8.2 depicts the distribution of the allocation strategies for the various policies. Not surprisingly, the numbers suggest that there is a correlation between the percentage of linear-scan-compiled traces and the register allocation time in Figure 8.1. Unless otherwise noted, the numbers mentioned in this section represent the geometric mean of the averaged benchmark results relative to TraceLSRA. All results are summarized in Table 8.1.

**TraceLSRA** TraceLSRA is the policy that performs best with respect to peak performance. It is the upper bound in terms of register allocation time but also produces the best code. In this baseline configuration, linear scan is used for 61% of the traces. The other traces are *trivial* and are therefore allocated by the trivial trace allocator.

**BottomUp** The BottomUp policy, on the other hand, is the lower bound with respect to allocation time. It requires only about 57% of the time used by TraceLSRA. In terms of peak performance, this policy is the slowest with an average performance decrease of about 11%. For the `sunflow` benchmark from the DaCapo suite, however, the performance penalty is 30%.

**Ratio** In our experiments, we evaluated the Ratio policy with the parameters 0.8, 0.5, and 0.3. Although the number of linear-scan-allocated traces decreased by 17% for $p = 0.8$, this is hardly noticeable in the allocation time and the peak performance. Setting $p = 0.5$

decreases the time for allocation to 87%. The performance slowdown is about 1% relative to TraceLSRA. With $p = 0.3$ allocation time is further reduced to 80% with a performance degradation of 4%. In this configuration, only 13% of the traces are allocated with the linear scan strategy. The results suggest that the Ratio policy allows a fine-grained tuning of compile time vs. peak performance.

**Budget** The Budget policy exhibits a non-linear behavior with respect to the parameter $p$. For $p = 0.99995$, we see a performance degradation of only 1% while the register allocation time goes down to 93%. Only 33% of the traces use the linear scan strategy. Setting $p = 0.98$ reduces the allocation time to 77% with a performance decrease of 3%. For $p = 0.5$, allocation time drops to 62%. The linear scan strategy is used for only 1% of the traces. In this configuration, basically, only the first trace of a method is considered important. The performance decrease is 10%, which is almost at the level of the BottomUp policy (11%).

**Loop** The Loop policy triggers for 26% of the traces, which is slightly less than half of the non-trivial traces (61%). Performance-wise this policy is about 2% slower than TraceLSRA. On the other hand, it requires only 86% of the time for register allocation.

**LoopBudget** The LoopBudget policy ($p = 0.5$) combines the advantages of Loop, i.e. good and stable peak performance, with the fast allocation time of the Budget policy. About 11% of the traces use the linear scan strategy. The allocation time therefore drops to 79% compared to TraceLSRA. With respect to peak performance this policy is 3% slower.

**MaxFreq** We evaluated the MaxFreq policy with $p = 0.1$ and $p = 0.8$. Compared to the TraceLSRA policy, MaxFreq with $p = 0.1$ is about 3% slower regarding peak performance. Again, `sunflow` exhibits the worst behavior with a performance decrease of 20%. Allocation time, on the other hand, is only about 77% of the time used by TraceLSRA. With $p = 0.8$ the allocation time drops to 71%. However, the impact on peak performance is significant. On average, the generated code is 7% slower than with TraceLSRA (max. 31%).

**NumVars** The evaluation of the NumVars policy shows that 32% of the traces have at most 8 live variables at their block boundaries (and are not *trivial*). Allocating these traces with the bottom-up strategy reduces the allocation time to 92%. Performance decreases by 3%. Extending the scope to 15 variables increases the fraction of bottom-up-allocated traces to 51% and reduces performance by 5% compared to TraceLSRA. However, the register allocation time goes down to 76%.

### 8.3.1 Discussion

Some of the evaluated policies are more appealing than others. The Ratio policy convinces due to its simplicity and scalability. There is no need to calculate metrics and the parameter correlates with the register allocation time saving. The Budget policy seems to offer the best trade-off, i.e., most compile time savings for the lowest performance penalty. However, it requires summing up the frequencies of all blocks. Loop is interesting because it solely depends on the structure of the trace, not on *profiled* frequencies. Therefore, this policy is an option even if profiling is not available or inaccurate. The NumVars policy is disappointing, especially for lower numbers where it achieves little compile time improvement for relatively high performance degradation.

### 8.3.2 Answering RQ3

Our trace-based register allocation approach offers the flexibility to switch between allocation algorithms within the same compilation unit. This gives us fine-grained control over the trade-off between compile time vs. peak performance, which is not supported in other register allocation approaches. Most policies can be parameterized, which allows adjusting the trade-off between compile time and peak performance on a fine-grained level. Our results confirm that our trace register allocation policy framework offers unique flexibility not seen in other approaches. Thus, trace policies answer **RQ3** positively.

| | Register Allocation Time (Δ%) | | | | | | Peak Performance (Δ%) | | | | | | Strategy (%) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Policy | mean | median | min | min | max | max | mean | median | min | min | max | max | LS | BU | TT |
| TraceLSRA | 0.0 | 0.0 | 0.0 | avrora | 0.0 | avrora | 0.0 | 0.0 | 0.0 | avrora | 0.0 | avrora | 61.2 | | 38.8 |
| Ratio-0.8 | −1.3 | −0.6 | −10.7 | jython | 3.0 | avrora | 0.7 | 0.4 | −0.8 | kiama | 6.5 | apparat | 43.9 | 17.3 | 38.8 |
| Budget-0.999995 | −7.5 | −4.9 | −33.4 | jython | −0.1 | avrora | 0.7 | 0.5 | −0.2 | sunflow | 2.0 | scalaxb | 32.6 | 28.7 | 38.7 |
| Ratio-0.5 | −12.6 | −11.0 | −30.6 | jython | −2.8 | avrora | 1.5 | 1.1 | 0.1 | scalatest | 9.2 | sunflow | 22.4 | 38.8 | 38.8 |
| Loop | −14.0 | −13.1 | −27.4 | jython | −4.8 | avrora | 2.2 | 1.8 | 0.3 | scalaxb | 5.5 | xalan | 26.4 | 34.8 | 38.7 |
| NumVars-8 | −7.9 | −8.3 | −14.2 | scalatest | −2.3 | luindex | 2.6 | 2.2 | 0.2 | sunflow | 8.7 | apparat | 29.2 | 32.0 | 38.7 |
| Budget-0.98 | −23.2 | −21.5 | −44.9 | jython | −11.3 | avrora | 3.3 | 2.9 | 0.6 | scalatest | 9.0 | luindex | 11.4 | 49.9 | 38.7 |
| MaxFreq-0.1 | −23.5 | −21.9 | −47.0 | jython | −7.7 | avrora | 3.4 | 2.7 | 0.1 | avrora | 19.6 | sunflow | 11.3 | 50.0 | 38.7 |
| LoopBudget-0.5 | −21.0 | −20.5 | −34.9 | scalac | −10.9 | lusearch | 3.5 | 3.8 | 0.2 | scalatest | 6.7 | apparat | 11.2 | 50.0 | 38.7 |
| Ratio-0.3 | −20.3 | −19.2 | −42.4 | jython | −8.1 | avrora | 3.6 | 3.0 | 0.7 | scalatest | 17.2 | sunflow | 12.6 | 48.6 | 38.8 |
| NumVars-15 | −23.9 | −25.1 | −34.0 | scalac | −6.4 | sunflow | 5.4 | 5.1 | 0.6 | avrora | 13.3 | scalaxb | 10.2 | 51.2 | 38.7 |
| MaxFreq-0.8 | −29.2 | −28.5 | −49.1 | jython | −16.5 | avrora | 7.2 | 5.5 | 0.6 | avrora | 30.5 | sunflow | 4.8 | 56.6 | 38.6 |
| Budget-0.5 | −37.5 | −37.0 | −52.9 | jython | −29.5 | avrora | 9.7 | 9.2 | 0.5 | scalatest | 30.6 | sunflow | 1.4 | 60.0 | 38.6 |
| BottomUp | −43.0 | −42.3 | −54.6 | jython | −36.0 | factorie | 10.6 | 9.6 | 0.8 | scalatest | 31.7 | luindex | 0.3 | 61.2 | 38.4 |

For every configuration, we show the (geometric) *mean*, the *median*, the *min* and *max* values of the benchmark results for both the register allocation time as well as the peak performance (lower is better). For *min* and *max* we also show the corresponding benchmark. The given numbers are the differences relative to TraceLSRA in %. The last three columns depict the distribution between the allocation strategies, *Linear Scan Allocator* (LS), *Bottom-Up Allocator* and *Trivial Trace Allocator* (TT).

Table 8.1: Experimental results for trace register allocation policies

# Chapter 9

# Parallel Trace Register Allocation

In this chapter, we focus on **RQ4**, i.e., whether trace register allocation can reduce compilation latency. *Compilation latency* is the duration required to compile a given method. If compilation happens on the main thread, latency has a significant impact on response time, since the execution of the application is delayed. But also for systems with one or more background compilation threads [Krintz et al., 2001], short latencies mean that the compiled code is available earlier and can therefore be executed earlier.

We exploit the flexibility of trace register allocation to reduce compilation latency without impacting peak performance. The idea is to use multiple threads to allocate traces concurrently. The trace register allocation framework allows traces to be allocated in arbitrary order. However, as discussed in Chapter 6, traces that are processed later can profit from decisions in already allocated traces. Optimizations based on this principle can improve the peak performance for some DaCapo benchmarks by up to 10%, as shown in Figure 7.10. We want to ensure that concurrent allocation does not influence the allocation quality. Therefore, we define dependencies in a way that all the information available in the serial mode is also available in the parallel mode.

We prototyped parallel trace register allocation and summarized our findings in a *work-in-progress* paper, which is currently under review [Eisl et al., 2018]. However, the current implementation is only a *proof-of-concept* and is not used by default.

## 9.1 Concurrency Potential

Before investing time on an implementation, we wanted to gain more insight into whether there is enough potential for parallelization. Therefore, we simulated the speedups when using 2, 4 and 8 threads for register allocation of the DaCapo and Scala-DaCapo benchmarks.

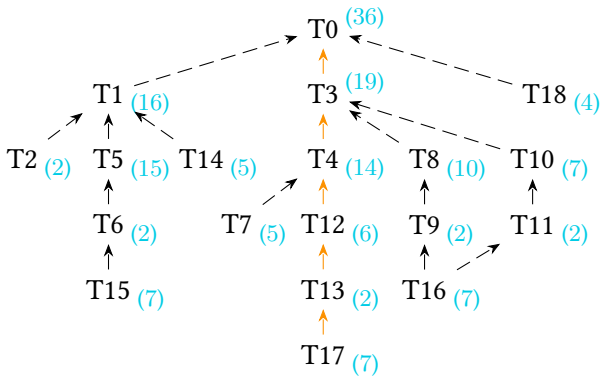
Not all traces require the same time for allocation. We use the *number of instructions* as a compile time estimator since it correlates with the time required for register allocation, as depicted in Figure 7.7. The lower bound for compile time is the length of the *critical path* in the dependency graph. Our experiments show that the (geometric) mean of the critical path length is $51\%$ ($min = 44\%$, $max = 57\%$) of the number of instructions in the compilation unit. That means that ideally, with an infinite number of threads and ignoring all overheads, the register allocation step could be done in about half the time.

To simulate the concurrency potential with a given number of threads, we need to find a schedule that satisfies the dependencies. Finding an optimal schedule with minimal duration is NP-hard [Pinedo, 2016]. Therefore, we apply a simple heuristic for finding a schedule. Whenever a thread is idle, we assign it to the *longest* trace in terms of instructions, whose predecessor traces have already been processed. For $2$ threads, the simulated register allocation time goes down by $68\%$ ($64\%$, $71\%$). This is already an interesting result, since adding only a single thread to the system can potentially improve register allocation time by about $30\%$. Another noteworthy metric is the *utilization* of the threads, that is the ratio between *work* and *idle* time. For $2$ threads this ratio is $74\%$ ($70\%$, $78\%$), which means that allocation threads are idle only one fourth of their run time. If $4$ threads are used, the simulated allocation time is $56\%$ ($50\%$, $61\%$) of the single-threaded case. However, the utilization also decreases to $45\%$ ($41\%$, $50\%$). The threads are idle more than half of the time. With $8$ allocation threads the allocation time is $52\%$ ($45\%$, $58\%$), which is almost the best that can be achieved given a mean critical path length of $51\%$ of all instruction. As expected, the thread utilization further decreases to $24\%$ ($22\%$, $28\%$).

### 9.1.1 Example

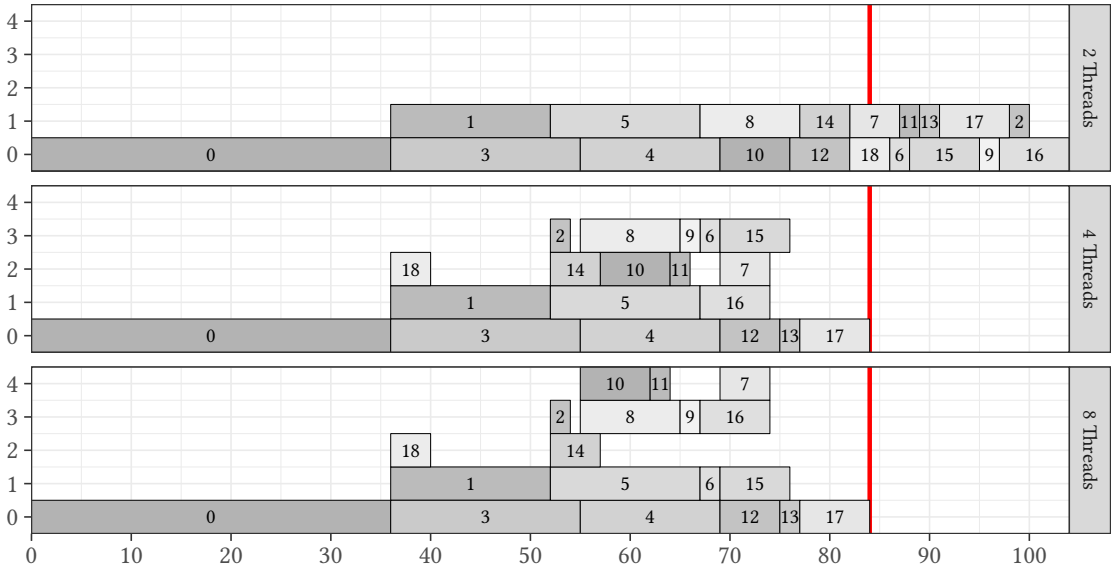
Let's illustrate the simulation with an example. We chose the method `PrintStream.write()`[1] from the Java standard library, since it is small enough to be understandable, yet long enough to exhibit scheduling potential. After high-level optimizations (e.g., inlining) there are $19$ traces. Their dependencies are depicted in Figure 9.1. The values in parentheses are the trace lengths in instructions. The compilation unit consists of $168$ instructions in total, which is also the *length* of the single threaded schedule. Figure 9.2 shows the calculated schedule as a *Gantt chart* [Gantt, 1913] for $2$, $4$ and $8$ threads (top, middle, bottom, respectively). Each rectangle represents a trace with its number. The horizontal axis depicts the length in terms of *instructions*. The length of the schedule with $2$ threads is $104$ instructions with a thread utilization of $81\%$. With $4$ threads the utilization decreases to $50\%$. However, the length of $84$ is already optimal, i.e., the *critical path* length. Therefore, increasing the number of threads to $8$ cannot yield any improvements and utilization drops to $25\%$. Figure 9.2 also shows that although $8$ threads are available, only $5$ are used.

[1] Full descriptor: `java.io.PrintStream.write(byte[], int, int)void`




Dependency graph for the method `PrintStream.write()`. The values in parentheses are the length of the traces, i.e., the number of instructions. The critical path (T0–T17, orange edges) is 84 instructions long.

**Figure 9.1:** Trace dependency graph for `PrintStream.write()`




Gantt chart [Gantt, 1913] for 2, 4 and 8 threads. The horizontal axis denotes the number of instructions. The compilation unit consists of 168 instructions in total. The critical path (red vertical line) is 84 instructions long.

**Figure 9.2:** Trace allocation scheduling for `PrintStream.write()`

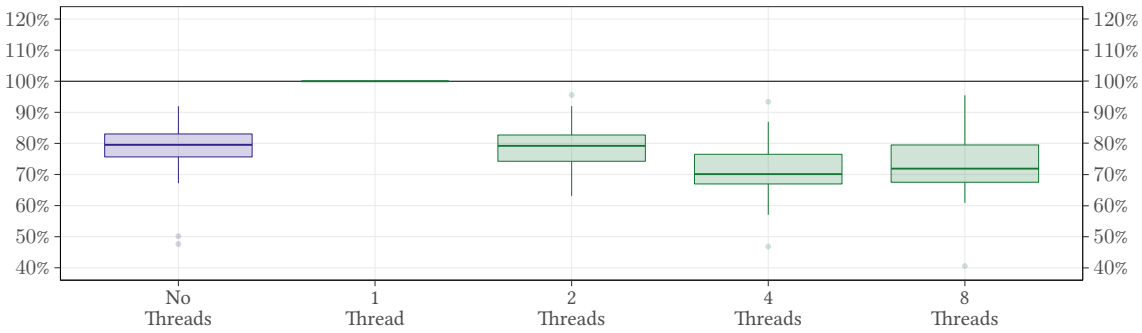
## 9.2 Evaluation

To verify that the theoretic results presented in Section 9.1 also apply in practice, we prototyped a parallel version of our trace register allocator. More specifically, we wanted to ensure that parallel register allocation (1) can improve register allocation latency, and (2) does not impair allocation quality. We used a standard Java *thread pool*[2] with a fixed pool size for the allocation threads. Idle threads are kept alive to avoid the overhead for starting a new thread. Once a trace is allocated, its successors are added to the work queue, if all traces on which they depend have already been processed. We use a *priority queue*,[3] where traces are ordered by decreasing instruction count, so that longer traces are allocated first.

From the register allocation point of view, synchronization is only needed for the queue and for tracking dependencies on finished traces. Accessing and modifying traces is safe by design of the trace register allocator if the dependencies are respected. However, Graal assumes that every method is compiled by just a single thread and therefore uses unsynchronized data structures. For example, the map from the *register class* to the set of *available registers*, which is lazily initialized. For our concurrent allocation approach, this is a problem. We worked around this issue, for example, by pre-populating cached maps, duplicating data-structures or simply recalculating results. These workarounds cause allocation time overheads which we are willing to accept for our prototype. We are confident that most of them could be mitigated by a more advanced implementation.

By default, GraalVM uses multiple compiler threads to concurrently compile different methods. Compilation happens in the background, that means the application continues to execute while a method is compiled. The compiler threads compete with the application threads. Adding threads for register allocation makes the situation even more challenging. To keep this interference low, we suppressed the parallel compilation of different methods and rather used the threads for the parallel allocation of registers. To measure register allocation time, we take a timestamp[4] before and after register allocation and report the difference, i.e., the *duration*. In other words, the *duration* is the time elapsed from the beginning of register allocation until it is finished and the result is available. Of course, the numbers are influenced by the scheduling of threads by the virtual machine. For example, if the VM decides to preempt the register allocation threads in favor of an application thread, the *duration* increases although the compiler or allocator did not perform more work. More precisely, the *duration* does not represent *CPU time*. However, the metric of *duration* is what we are interested in, since the goal is not to reduce the *work* that is done by the allocator but to have the result available *earlier*.

[2] See `ThreadPoolExecutor`:
`https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html`
[3] See `PriorityBlockingQueue`:
`https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/PriorityBlockingQueue.html`
[4] See `System.nanoTime()`: `https://docs.oracle.com/javase/8/docs/api/java/lang/System.html`



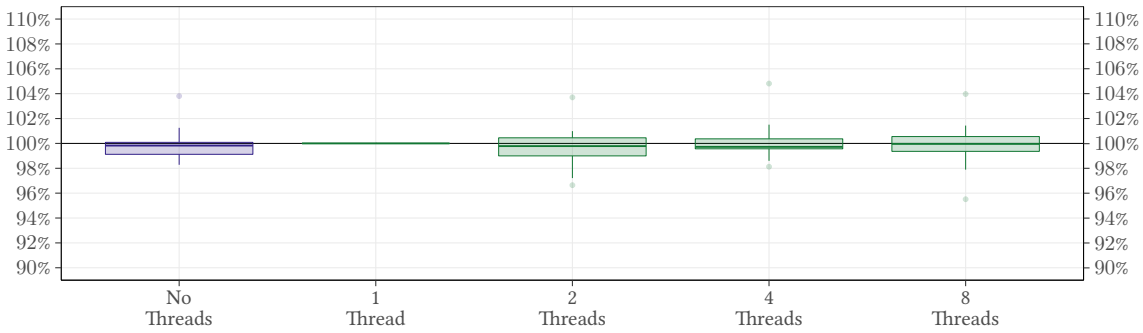

DaCapo and Scala-DaCapo on AMD64. Values are relative to 1 allocation thread. Lower is better ↓.

**Figure 9.3:** Concurrent register allocation time

We evaluated our approach using (Scala-)DaCapo on the X3-2 cluster (see Appendix C.1). For each configuration, we collected at least 10 results. The experimental results are summarized in Figure 9.3. Note that the reported values are the durations of register allocation, including all necessary phases. This includes *trace building*, *global liveness analysis* and *global data-flow resolution*. Note that the potential analysis in the previous section only reasons about the register allocation allocation itself, not about the global phases. All numbers are relative to the 1 Thread configuration, which uses the priority queue and the other synchronization mechanisms, but the thread pool only consists of a single thread. To see the overhead imposed by our prototype, we also compare it against the No Threads configuration, i.e., the trace register allocation where all work is done by the single compiler thread. The mean overhead of 23% might seem high at first sight. However, as already mentioned before, our implementation is an initial proof-of-concept prototype. We are confident that this gap can be reduced by a more advanced implementation.

To answer the question whether we can reduce allocation latency, we evaluated the prototype with 2, 4, and 8 allocation threads. Using 2 threads instead of one decreases the latency by 21% (4%, 37%). Using 4 threads instead of one decreases the latency by 30% (7%, 53%). As we already expected from the potential analysis, using 8 threads does not give any further advantages. In fact, the result is slightly worse than with 4 threads. This is due to the additional synchronization effort and the fact that we are close to the number of hardware threads.

The second assumption we wanted to verify in this evaluation is that parallel register allocation does not affect allocation quality. To do so, we report steady-state performance in Figure 9.4. The results suggest that parallel register allocation has no negative impact on allocation quality.




DaCapo and Scala-DaCapo on AMD64. Lower is better ↓.

**Figure 9.4:** Benchmark execution time for concurrent register allocation

### 9.2.1 Answering RQ4

Although our prototype is in an early stage, the results show that parallel register allocation is practical and can improve compilation latency without impairing the allocation quality. **RQ4** can thus be answered positively.

## 9.3 Future Directions

For our parallel trace register allocation experiment, we applied a strict dependency model to guarantee the same results in serial and parallel mode. *All* preceding traces with higher importance must have been allocated before processing the succeeding trace. This could be relaxed by processing a trace on a new thread already when its most important predecessor trace has been processed, which would open up more potential for parallelization (although some inter-trace data flow might become suboptimal). We used the number of instructions as the priority function for selecting the next trace to be processed. An alternative would be to use the number of successors in the dependency graph in order to enable more concurrency opportunities.

# Chapter 10

# Related Work

Trace register allocation relates to a variety of research areas in the field of compilers and virtual machines. The diverse field of *register allocation* is of course highly relevant for our work. Thus, we first look at common register allocation approaches and discuss their complexity and optimality results. Optimizations that work on non-global code units, such as traces or regions, are often motivated by similar considerations as in our approach. We also look at *trace scheduling*, since this class of optimizations did not only influence the name of our approach, but also uses the very same unit of scope. Another related area is *trace compilation*, which follows a very different approach for compiling code than our method-based compilation does. However, there are areas where trace compilation and trace register allocation can learn from each other. *Liveness* is the core information of register allocation. Our global liveness information relates to other representations of liveness used for register allocation and compilation in general. Our main focus is on the improvement of compile-time characteristics by either trading-off peak performance, or doing work in parallel. In the last section of this chapter we discuss other trade-off techniques used in virtual machines.

## 10.1 Register Allocation

In our introduction in Chapter 1, we have already discussed two well established register allocation approaches, *graph coloring* and *linear scan*. Register allocation is a very active area of research in compiler construction for at least six decades. There are literally hundreds of papers on register allocation proposing dozens of different approaches. In this section we focus on further work that either directly relates to trace register allocation, provides pointers for future improvements, or is for other reasons interesting in this context.

### 10.1.1 Local Register Allocation

*Local register allocation*, where registers are allocated just for a single basic block, is particularly interesting for trace register allocation. Like a single basic block, traces consist of straight-line code, which simplifies the allocation problem. Approaches that were proposed for local register allocation can also be used for traces. Other results, for example, regarding complexity or optimality, might apply as well, although the prerequisites must be considered carefully. If reasoning, for example, builds upon branch-free code, then it does not hold for traces because there are control-flow joins and merges.

Approaches similar to our bottom-up allocation strategy have been described previously for *local register allocation*, which is not surprising given its simple and straight-forward idea. Cooper and Torczon discuss a *bottom-up allocator* in their book *Engineering a compiler* [2011, Chapter 13]. However, there are a few minor differences in our implementation. We initialize our variable/location map to match the successor trace to avoid data-flow mismatches, which usually is not done in local register allocators. Another difference is how we select spill candidates. The bottom-up allocator described by Cooper and Torczon spills the register with the longest distance to the next usage (*furthest-first*). While this improves the allocation quality, it also requires more work to maintain this information. We experimented with similar heuristics, but they all have a significant negative impact on allocation time. Since fast allocation time is the main goal of our bottom up approach, we refrained from such optimizations.

Farach-Colton and Liberatore [2000] showed that local register allocation is NP-hard with respect to minimal spill loads and stores. Since local register allocation can be trivially *reduced* [Cook, 1971] to trace register allocation,[1] the hardness result also applies to allocating a trace. However, other results by Farach-Colton and Liberatore do not. The 2-approximation algorithm, for example, assumes that the cost of a spill is the same for all program points. For traces this is not true because of splits and joins. Whether the algorithm can be extended to traces is left for future research.

The *furthest-first* heuristic is often used for selecting a spill candidate in local register allocators, but also in global approaches such as linear scan. Facing the problem that no register is available at a certain instruction, the *furthest-first* strategy selects the register whose next usage is the furthest in the future. The heuristic is often contributed to Belady [1966] and Horwitz et al. [1966], but apparently it dates back to FORTRAN compilers from the 1950s [Farach-Colton and Liberatore, 2000]. It is optimal for basic blocks if a variable is never written, e.g., there is no need to move its value from a register to memory [Farach-Colton and Liberatore, 2000]. We have already seen that minimizing the spill costs is NP-hard.

[1] Every block is considered a trace.

Although the *furthest-first* strategy is not optimal for realistic cost models, the heuristic has successfully been applied in practice. Guo et al. [2003] evaluated the *furthest-first* strategy on large basic blocks (300–5000 instructions). The blocks are the result of unrolling loop bodies of number crunching libraries such as *Fast-Fourier-Transformation* or *Matrix Multiplication*. On the MIPS architecture their compiler using a furthest-first strategy outperformed GCC and the MIPSpro compiler by up to 33%. Especially in cases where the register pressure is high, the heuristic seems to work well.[2] A variant of this heuristic is also used in our linear scan strategy.

### 10.1.2 Non-Global Register Allocation

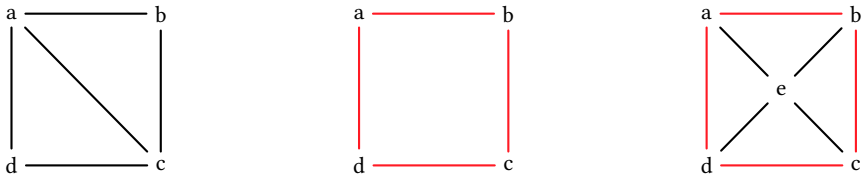
In contrast to global approaches such as graph coloring or linear scan, there are register allocators that operate on subparts of the method and merge the intermediate results for a final solution, just like our trace register allocation approach. However, most of them do not strictly separate the allocation of the subparts, or they force a particular allocation order, for example, from inner loops to outer loops.

Callahan and Koblenz [1991] proposed *hierarchical graph coloring* as a technique to minimize the number of dynamically executed spills by moving them to infrequently executed code parts. They also divide the control-flow graph into what they call *tiles* and (partially) solve the register allocation problem for each of them. Tiles are sets of basic blocks. Two tiles are either non-overlapping or one is a proper subset of the other. Callahan and Koblenz use loop-nesting as well as branches for their tile hierarchy. The difference to our approach is that tiles are not independent but organized in a hierarchical tree. Allocation starts at the innermost tile (the leaf of the tree). Once all children are processed, the parent tile uses the information of its children to continue. It is also worth noting that the algorithm does not assign registers in the first pass but uses pseudo registers to record the requirements of a tile. In a second top-down pass, physical register are assigned and spill code is inserted if needed. Callahan and Koblenz use graph coloring for allocating a tile. Also, the interference graph is used to communicate allocation decisions from a sub-tile to its parent. This rules out the use of algorithms for tiles that do not work on interference graphs, for example linear scan. The hierarchical graph coloring allocator features *preferencing* as an alternative to coalescing. It supports *local preferencing*, which is similar to the hints we use in our linear scan strategy, as well as *inter-tile preferencing*, which relates to our inter-trace hints (Section 6.1). It is noteworthy that sub-tiles do not deal with variables which are *live* but not used in the tile. This is somehow related to our handling of *stack intervals* (Section 6.4). However, our approach basically works in top-down direction. Therefore, we still maintain stack intervals to keep them in registers if possible, or inform successor traces that these intervals are already spilled.

[2] However, no details are given about the allocation algorithms used in the reference compilers, nor whether they apply global or local approaches. This makes is difficult to assess the results in more detail.

The motivation for *register allocation based on graph-fusion* by Lueh et al. is similar to Callahan and Koblenz. Its goal is to minimize the number of dynamically executed spill instructions. Their allocator works on regions of the control-flow graph, where a region can be a basic block, a loop-nest, a superblock [Hwu et al., 1993], or some other combination of basic blocks. They build the interference graph for each region and ensure that it is colorable by potentially spilling live ranges, as it is done in the global graph coloring approach. However, registers are not yet assigned. Connected regions are then *fused* to build up the graph for larger parts of the compilation unit. The key property of *fusing* is that it ensures that the graph stays colorable. If fused regions would yield an uncolorable result, live ranges are split at the edge that connects the two regions and *shuffle code* is inserted. This way, the interference graph for the whole compilation unit is incrementally constructed. Due to the invariant that the graph is always colorable, the *register assignment phase* can simply perform a Chaitin-style *simplify and coloring*. No more spill decisions need to be made. Although, the *fusion-based allocator* starts with local spilling decisions, the decisions are getting more global incrementally. Also, the actual register allocation is delayed until a global view on the problem is available. In contrast to that, we solve the problem locally and independently for each trace. In addition, the graph fusion approach uses the interference graph as the single model for liveness. The complex *fusion* operation and the global *assignment phase* limit the potential for doing work concurrently. In contrast to that, in our trace register allocation approach the tasks are independent and are only connected via $live_{in}/live_{out}$ set at trace boundaries. Once a trace has been processed, its result is final and will not be changed apart from inserting resolution code.

Koes and Goldstein [2006b] proposed a *global progressive register allocator* using *multi-commodity network flow* (MCNF). Their flow formulation explicitly models spill cost, register hints, live range splitting and constant rematerialization. They first create a network for the local problem (i.e. for a basic block) and then extend it to the global case. To find an initial solution, Koes and Goldstein first use a heuristic approach (*second-chance binpacking* [Traub et al., 1998] or a simple graph coloring) which yields a valid, but potentially suboptimal solution. Then their progressive solver iteratively improves this result towards the optimal solution with respect to the overall costs. The progressive model allows them to control the compile-time vs. peak-performance trade-offs which relates to our *allocation policies* (Chapter 8). Note, however, that their decision whether to continue optimizing or stop is global, i.e., on the whole method, while we distinguish between important and unimportant traces. Later, Koes and Goldstein [2006a] proposed an extension to their idea where they used traces instead of basic blocks as the scope for the local problem, which further improved the quality and the speed of their approach. We find the idea of *progressively* improving the allocation result very appealing. Applying their approach (or earlier work on a local progressive allocator [Koes and Goldstein, 2005]) to traces could perfectly fit into our *policy model* and poses an interesting future research direction.




(a) chordal (b) non-chordal (c) non-chordal

A graph is *chordal* if every cycle with four or more edges has an edge which is not part of the cycle but which connects two vertices on the cycle. Cycles of length four without a *chord* are highlighted in red. Example taken from Pereira and Palsberg [2005].

**Figure 10.1:** Examples for *chordal* and *non-chordal* graphs

### 10.1.3 Decoupled Register Allocation

*Decoupled register allocation* is an interesting class of allocation approaches that got a lot of attention over the last years. Basically, the idea is to split the *spilling* and the actual *register allocation* into two separate phases. Since the first phase guarantees that the register allocation will succeed, the second phase is very simple. Actually, the *graph fusion* approach by Lueh et al. [1997] already follows a similar principle. However, most recent decoupled approaches exploit graph-theoretic properties of *chordal graphs*.

**Definition 34** (Chordal Graph)**.** A graph is *chordal* if every cycle with four or more edges has a *chord*. A chord is an edge which is not part of the cycle but which connects two vertices on the cycle.

Figure 10.1 depicts examples for *chordal* and *non-chordal* graphs. In 2005, several research groups have independently proven [Brisk et al., 2005; Hack, 2005; Bouchez et al., 2005] that interference graphs for programs in SSA form are *chordal*.[3] Hack et al. [2006] have motivated that chordal graphs have two properties that are utterly relevant for register allocation:

1. Their *chromatic number*, i.e., the smallest number of colors needed to color a graph, is equal to the size of the largest *clique* (see below).

2. They can be optimally colored in quadratic time with respect to the number of nodes.

*Interference graph cliques* are the set of variables that are live at the same time, i.e., the *register pressure*. Consequently, if we can reduce the register pressure at every program point to be at most the number of available registers, coloring will succeed. Optimal coloring means that we can color a graph with as few colors as possible. In register allocation, however, we are only

[3] Those results eventually ended up in the PhD theses "Advances in Static Single Assignment Form and Register Allocation" by Brisk [2006], "Register Allocation for Programs in SSA Form" by Hack [2007], and "A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases" by Bouchez [2009].

interested in *any* coloring which does not exceed the number of registers. We can find such a coloring in linear time, for example, by iterating the lifetime intervals in a linear scan fashion, as suggested by Rong [2009]. Since this renders allocation a non-issue, it allows us to concentrate solely on spilling. Braun and Hack [2009] discussed a spilling technique based on the *furthest-first* heuristic. In its traditional form, all variables that are not used in the block itself but are live afterwards have the same distance ($\infty$). Braun and Hack [2009] extended this to control-flow graphs. During the liveness analysis they collected the use-distance at basic block boundaries. The minimum of the distances of the successors plus the distance to the block end is then used as the *furthest-first* estimator. Another issue with decoupled spilling is that reloading a variable from the stack destroys SSA form since it redefines a variable. Braun and Hack [2009] solved this by introducing a new variable and placing $\varphi$s in successor blocks to merge the original and the reloaded version.[4] See Braun et al. [2013] for more details on SSA form reconstruction.

Decoupled register allocation is interesting for trace register allocation for multiple reasons. First, the interference graph of variables in a trace is an *interval graph* (Corollary 3), which is—in the case of SSA form—a subclass of chordal graphs (see also Rong [2009]). Therefore, we can decouple spilling and allocating registers for a single trace. However, we could extend this to a set of traces, for example, the most important ones. This would allow us to systematically solve the issue presented in Section 6.3. The question remains, how to deal with the loss of the *single definition* property of SSA form due to the reloads from the stack? The good news is that these violations are local to a trace. The overall framework does not care as long as the $loc_{in}/loc_{out}$ sets are preserved.[5] We only need to ensure that our allocation strategies can deal with *pre-spilled* variables. As already argued in Section 5.3, the bottom-up strategy does not strictly require SSA form. We only need to ensure that whenever a variable is *defined* by a reload instruction, the *variable location* is updated to the stack slot, since we need this information for updating the $loc_{in}/loc_{out}$ sets at trace boundaries. Also for the linear scan strategy (Section 5.1), dealing with this special form of redefinition is possible. In the interval building step we create a new lifetime interval for every definition of a variable. Whether there are multiple intervals for the same variable does not matter since all the required moves have already been added. Since the linear scan does no longer introduce new spills, this cannot interfere with the linear scan allocation procedure.

Rong [2009] made some interesting observations regarding variable liveness and chordal interference graphs. First, he reformulated a theorem by Gavril [1974] saying that if and only if the interference graph is chordal, the lifetime of the variables span a subtree in a tree of basic blocks. Based on this formulation he proposed the class of *tree register allocation*. Moreover, he argues that this model subsumes previous approaches such as linear scan or local register allocation. Rong evaluated different tree structures such as a single basic block (local register allocation),

[4] However, the new $\varphi$ defines yet another variable!
[5] In fact, register allocation in a trace always destroys SSA form!

the dominator tree, extended basic blocks, and the *maximal spanning tree*. Clearly, traces fit perfectly into this model, so trace register allocation belongs to the same class. For every tree, Rong performs a decoupled spilling and allocation phase. A data-flow resolution phase fixes mismatches on control-flow edges that are not part of a tree.

As Brisk and Sarrafzadeh [2007] showed, interference graphs which are *interval graphs* (Definition 32) have interesting properties for register allocation. One property is that the $k$*-colorable subgraph problem* can be solved in polynomial time [Yannakakis and Gavril, 1987]. That means that given an (interval) interference graph, we can get the *largest* subgraph (with respect to nodes) which is $k$-colorable. The subgraph represents the variables that are kept in registers, the others are spilled.[6] Since interference graphs of traces are interval graphs (Corollary 3), this also holds for trace register allocation. Note, however, that this is orthogonal to minimizing spill costs which is NP-hard, as we have argued above.

The $k$*-colorable subgraph problem* can be solved in polynomial time for chordal graphs if $k$ is fixed [Yannakakis and Gavril, 1987]. In register allocation, $k$ is the number of registers, which is usually a constant for a given architecture. However, it is polynomial in the order of $n^k$ which is for realistic cases still unpractical [Brisk and Sarrafzadeh, 2007].

### 10.1.4 Mathematical Programming Register Allocation Approaches

We have already seen several ways to model register allocation. However, most of them only cover parts of the overall problem. Graph coloring nicely describes interference but fails when spilling is involved. Linear scan precisely models live ranges and allows simple splitting, but information about control flow is mostly lost. The *furthest-first* spilling heuristic optimally handles the selection of a register to be spilled, but does not model the cost of spill code. Also, architectural irregularities like *register aliasing* [Lee et al., 2007] or fixed register constraints are often ignored or handled heuristically. Unifying all requirements into a single model is appealing if an optimal or near-optimal solution is the goal.

We have already discussed Koes and Goldstein's MCNF model for register allocation. There are a wide variety of register allocation approaches that apply different mathematical programming models for solving the problem.

Goodwin and Wilken [1996] described an optimal register allocator as a 0-1 integer programming problem. It incorporates coalescing, splitting, rematerialization, fixed registers, and spill code placement. The input program is modeled as a function of boolean decision variables, each

[6] Note, however, that this ignores the fact that not all usages of a variable can directly access the stack. An intermediate register is needed. In practice, scratch registers could solve this issue.

of which represents a certain allocation action, for example, inserting a spill move. Afterwards, their allocator uses the CPLEX solver[7] to find a solution. Once there is one, the actions associated with variables that are assigned to 1 are performed. Since 0-1 integer programming is NP-complete [Karp, 1972], the solver has exponential worst-case run time, so they use time limits in their experiments. Since the high compile time overhead is not always practical nor needed, they proposed a *hybrid allocation approach*. Based on profiling, they distinguish between *critical* and *non-critical* regions in a function. Critical regions are allocated optimally, while others use a standard graph coloring approach. Resolution code is inserted at region boundaries. This drastically reduces allocation time while the result is still near-optimal. The *hybrid* approach is similar to our *policies*, which we use to switch between the linear scan and the bottom-up allocator. As already remarked, adding an *optimal* allocation strategy would further enrich our system.

Scholz and Eckstein [2002] formulated register allocation as a *partitioned boolean quadratic optimization problem* (PBQP). They specifically focused on irregularities of non-RISC architectures, as for example used in *digital signal processing* (DSP). One of the main issues with comparable approaches, for example, Goodwin and Wilken [1996], is the exponential solving time. To overcome this issue, they developed a heuristic PBQP solver that exhibits nearly linear run time. The allocator with the heuristic solver outperforms graph coloring in all but one of their (limited set of) benchmarks (with a speedup of 1.9–10%) while showing a similar allocation time behavior. The speedup of the optimal solution is in the range of 1.3 to 13.6%.

Later, Hames and Scholz [2006] presented a new branch-and-bound solver for PBQP to circumvent poor performance of the initial heuristic on SPEC2000 benchmarks on Intel's IA-32 architecture. With this new solver, they were able to be within 3% of the optimal solution regarding spill costs. Graph coloring, which is about 4× faster, the spill costs were never worse than 7% above the optimum. However, overall results suggest that both, graph coloring and branch-and-bound-based PBQP, are very close to the optimal solution and that there is only little room for improvement (at least for IA-32).

Ebner et al. [2009] combined the *decoupled register allocation* idea with a mathematical model. Since the actual allocation is a non-issue, their formulation deals with spill code placement. For every variable, their allocator produces a network where the nodes represent positions at which the variable is *live*, and the weighted edges denote the cost of *reloading* a variable between these positions.[8] The networks for the individual variables are combined to correctly represent the register demand by every program point. By searching for the *minimum-cut* in the combined network, they get a solution to the reload placement problem. The comparison with the linear scan implementation that was used by LLVM at that time [Lattner and Adve, 2004] showed that their approach speeds up the execution of the compiled code for a selected set of benchmarks from

[7] `https://www.ibm.com/products/ilog-cplex-optimization-studio`

[8] In addition, a node representing the *spill move* is added. An interesting extension would be to support with multiple spill moves.

the MiBench suite by up to 30% on a *Very Long Instruction Word* (VLIW) architecture. Especially when register pressure is high, they achieve significant improvements. For the SPECint 2000 benchmark on an ARM Thumb their speedup is 6% on average. In addition, they present a *Lagrangian relaxation* algorithm that finds a valid solution fast and progressively improves it. This allows them to trade-off compile time for code quality.

Lozano et al. [2012] modeled register allocation using constraint programming [Mayoh et al., 2013]. One interesting property of their approach is the use of *Linear SSA* (LSSA). This is similar to our global liveness information. Every block explicitly denotes $live_{in}$ and $live_{out}$ variables. They model every block independently. The blocks are only connected via their $live_{out}/live_{in}$ relations. There are analogies to the way we allocate traces independently. We can think of two potential crossover areas of their approach and ours. First, the constraints model could be applied to traces instead of basic blocks to increase the scope for optimization. This is especially interesting since their approach also integrates instruction scheduling, which could benefit from long straight-line code. Second, we could integrate their local constraints model into our framework to solve individual traces and combine this with our policies and existing strategies.

### 10.1.5 Register Allocation in Virtual Machines

Many of the register allocation approaches we already described, were developed for—or evaluated on—virtual machines. There are, however, approaches that are interesting not only because of their allocation algorithm, but because they explicitly integrate with a virtual machine.

Cavazos et al. [2006] proposed a *hybrid optimization* mechanism to switch between a graph coloring and a linear scan allocator in the Jikes RVM. They use an *offline* machine learning algorithm to find a *decision heuristic*. Their induced heuristic reduces the *total time* (compile time plus benchmark execution time) by 9% on average over graph coloring for a selected set of benchmarks from the SPECjvm98 suite. To classify a method, they use *properties* which are similar to those we are using in our policies in Chapter 8. However, we can change the allocation algorithm for each trace even within a method. This allows a more fine-grained control over the compile-time vs. peak-performance trade-off. Nevertheless, deducing parameters for our policies using machine learning techniques seems like an interesting experiment to us.

Also related to our proposed bottom-up allocator is the work by Yang et al. [1999]. They described LaTTe, a compile-only Java VM that focuses on compilation speed, including a fast, non-local register allocator. Register allocation is performed on tree regions, which are trees of basic blocks with a single entry and potentially multiple exits. The allocator does a backward pass to collect register preferences based on the requirements at the exits of the allocation region. After

collecting the references, a forward pass performs the actual register allocation. Their spilling technique is similar to the approach used by our bottom-up allocation strategy. However, we perform allocation on traces instead of trees and require only a single pass over the instructions.

The (*baseline*) compiler of the CACAO VM uses a simple yet efficient register allocation scheme [Krall, 1998]. It is tailored for allocation of an intermediate representation that is based on Java bytecodes, as well as for RISC architectures like the Alpha processor, where there are many registers available. The first observation Krall [1998] made is that the Java-to-Bytecode compiler `javac` already performs *pre-coloring*. Non-interfering local variables are assigned to the same slot. Second, the live range of stack variables is implicitly encoded in the *push* and *pop* operations of the values.[9] Therefore, no complex lifetime analysis is required, which enables a simple first-come-first-serve allocation scheme. Another observation is that the expression stack at basic block boundaries is mostly empty (93%) or very small ($< 7$). This allows efficient interfacing between basic blocks. The CACAO allocator first allocates these interface variables. Next, temporal variables used only in a basic block are processed. Finally, local variables are assigned on a global scope. Therefore, local variables are most likely to be spilled. While this approach performs well if the register set is large and the method is small, it tends to be suboptimal for architectures with only few registers, and if method size increases, for example, due to inlining, as discussed by Steiner et al. [2007] and Steiner [2007].

## 10.2 Non-global Code Units

As we showed in this thesis, the global scope of a method is not always the adequate unit for optimizations. In literature, alternative approaches were proposed.

Fisher [1981] introduced traces as compilation units for instruction scheduling on VLIW architectures. The goal is to reorder instructions so that instruction level parallelism (ILP) can be increased. The Bulldog compiler by Ellis [1985] and later the Multiflow compiler by Lowney et al. [1993] are popular implementations of this approach. Their definition of a trace is similar to our use of the term. However, their traces do not span across loop headers. Although the main focus of the approach is on scheduling, they treat registers as a scheduling resource in the process [Lowney et al., 1993]. Freudenberger and Ruttenberg [1992] presented a detailed description of the interaction of those two optimizations. Whenever a trace is processed, they record the register assignment decisions at trace exits and entries in a so-called *value-location-map* (VLM). This is basically the same as our $loc_{in}/loc_{out}$ sets. Subsequent traces need to adhere to the VLMs of neighboring traces, i.e., the freedom of later traces is incrementally reduced. This relates to

[9] This also includes implicit pushing and popping, for example, by arithmetic instructions. Also, stack manipulation instructions like `dup` and `swap` need special care.

our *inter-trace hinting* (Chapter 6). However, since we use a dedicated data-flow resolution to fix mismatches, an allocation strategy is free to ignore hints. In Multiflow, values that are live throughout a trace but never referenced are not immediately assigned to a register, but the decision is *delayed* until they are referenced in a later trace. There are similarities to *stack intervals* in our approach.

For most of this thesis, we assumed methods to be the input to the compiler. However, a programmer usually (hopefully) partitions the code into methods based on software engineering principles, which do not take into account what is important to be optimized by the compiler. Outsourcing code into subroutines is mostly due to modularity or maintainability concerns, not due to performance considerations. Method-based compilers "undo" this restructuring by aggressively inlining methods. However, this can lead to big compilation units which increase the pressure on the optimizations.

As an alternative, Hank et al. [1995] suggested *region-based compilation* which allows for aggressive optimization while keeping the compilation unit size small. First, they perform aggressive inlining to enlarge the scope. Afterwards, the basic blocks are partitioned into smaller regions which are then compiled independently. Their region building algorithm starts similar to our *bidirectional trace builder*. They use profiling data to select a *start* block and then append successors before going upwards. They always add the most frequently executed successor or predecessor. In contrast to our approach, they use a threshold value to limit the size of the region. After there is an initial trace, the algorithm continues to append successors to the blocks as long as their execution frequency exceeds the threshold.

Hank et al.'s analysis revealed, that programs exhibit different performance-critical interactions between functions. Some benchmarks spend over 90% of the time in regions of one or two functions. Other benchmarks, however, spend 70% of their time in regions consisting of blocks from 10 or more functions. Their region-based approach is able to cope with all of them in a uniform manner.

## 10.3 Trace Compilation

Hank et al. already pointed out that methods are not always the right unit of compilation. Nevertheless, they still perform inlining before determining their units of operation. However, in some cases this detour is not necessary or not even possible, for example, in the case of dynamic machine code optimization. Although *call* and *return* instructions can still give a hint about the original structure, determining which code segment belongs to which method is impractical. Therefore, other approaches are required.

Dynamo by Bala et al. [2000] was the first widely known system using an approach called *trace compilation* for dynamic translation of compiled binaries. Their system executes machine code in an interpreter. For every backward branch, they increment a counter associated with the branch target. Once this counter exceeds a certain limit, their interpreter enters a *recording mode*, i.e., it appends every executed instruction to a *trace buffer*. Calls and returns are handled just as if they would be jumps, i.e., functions are opaque to the trace. Recording continues as long as no backward branch is taken.[10] Once trace recording has finished, the machine code is transformed into an IR and is optimized. Since traces are straight-line code, the optimizing compiler is simple and efficient. The optimized code is installed into a so-called *fragment cache*. Whenever the interpreter would branch to the location that triggered trace recording, the compiled fragment is invoked instead of interpreting the original machine instructions. Branches that exit the compiled code return to the interpreter, potentially through compensation code to fix the execution state. To avoid switching to the interpreter, traces can be linked together.

Gal et al. [2006] showed that this approach perfectly fits the needs of resource-constrained devices. Their light-weight implementation of a trace-based JIT compiler for a JVM reached a performance that is comparable with full-fledged production quality systems. The traces are recorded on bytecode level using an interpreter. Gal et al. use an SSA-based representation for their compiler. Because only the relevant basic blocks are considered, bytecode parsing overhead is avoided for uncommon parts. Similar to Dynamo, the HotPathVM performs *trace merging* where different traces are stitched together. When compiling the child trace, they initialize the state of the variables to the mapping present at the end of the parent trace. The idea is similar to our *inter-trace hints*. For register allocation inside a trace, they use a bottom-up approach which also emits machine code. As we do in our allocation strategies, $\varphi_{in}$ and $\varphi_{out}$ operands are coalesced, if possible.

The trace-based approach is also suitable for dynamically typed languages as shown by Gal et al. [2009]. The empirical study by Garret et al. [1994] suggests that dynamic types in the hot regions of a program are relatively stable. Based on this assumption, Gal et al. generate type-specialized native code for hot traces. They implemented this idea in TraceMonkey, a tracing JavaScript virtual machine that was used in Mozilla's Firefox browser.

Häubl and Mössenböck [2011] modified the HotSpot client compiler in order to perform trace-based compilation. A difference to other tracing approaches is that when a call is encountered, a new trace is started and linked to the current one. This keeps the exact call information, which is important for a system like the HotSpot VM, that is tuned for method based compilation. Their approach provided speedups of the compiled code compared to the method-based client compiler, although less code was compiled. They use the linear scan implementation for register allocation that is used by default in the client compiler [Wimmer and Mössenböck, 2005].

[10] Or branches to already compiled segments, as we will see shortly.

Bolz et al. [2009] used trace compilation for obtaining an efficient JIT compiler for bytecode interpreters that were implemented in the PyPy language implementation framework. They applied the tracing compiler not to the user program but to the bytecode interpreter that was written by the language implementer. This approach is also referred to as *meta-tracing*. The meta-level imposes some challenges compared to traditional trace compilation. Usually, tracing assumes that the path through a hot loop is the same in most iterations. For bytecode interpreters, the inner loop is the dispatch loop, where each iteration represents one interpreted bytecode. Since usually a different bytecode is executed in every iteration, the original assumption does not hold. To solve this problem, Bolz et al. proposed *hints* in the interpreter code to inform the trace recording mechanism about program counter information in the language interpreter. This information, in addition to the interpreter code, is used to refine trace-start and trace-end conditions. In other words, a trace is an *unrolled* version of the dispatch loop that represents a loop in the *user program*. The advantage of the meta-tracing approach is similar to Truffle (see Chapter 3). A language implementer only needs to implement an annotated bytecode interpreter and the meta-trace framework enables JIT compilation automatically.

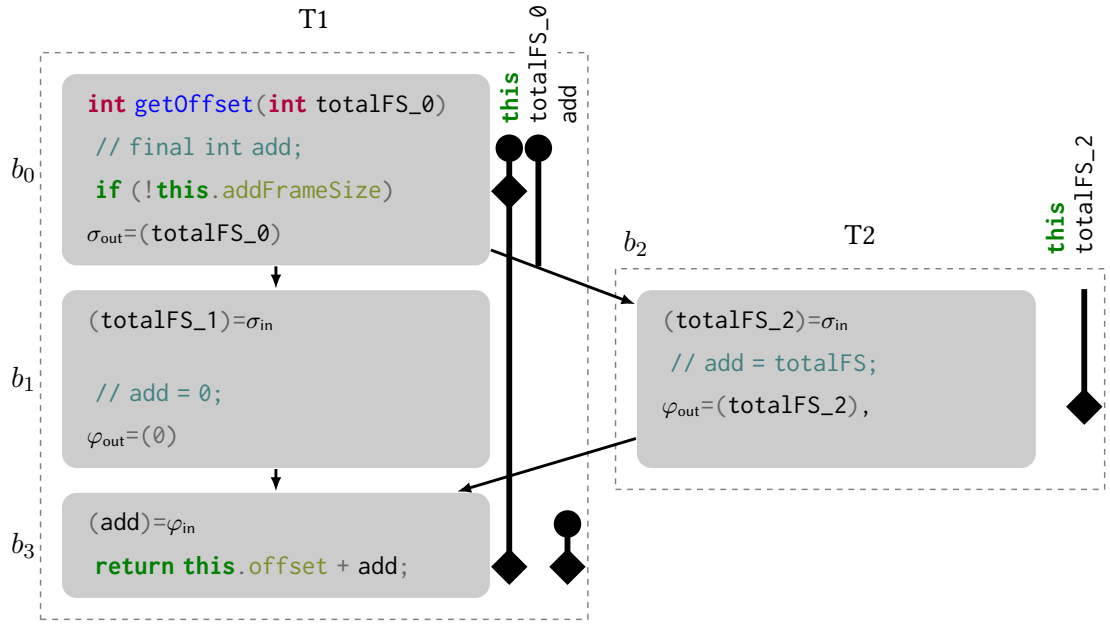
There are some similarities between trace compilation and the traces we use for register allocation. Both work on straight-line code, which simplifies optimizations and analysis, for example, when calculating lifetimes. One difference, however, is that in general, traces have no side entries, i.e., a trace is always entered through its head.[11] In the context of trace compilation, code pieces (e.g. a basic-block or an instruction) can usually occur in multiple traces, i.e., there is some kind of code duplication. This is not the case in our implementation. Although register allocation is not the focus of trace compilation, allocation strategies used in this area can also be applied to trace register allocation and *vice versa*.

## 10.4 Liveness and Intermediate Representations

A proper and easy-to-use notion of liveness is utterly important for register allocation. In our approach, we use what we call *global liveness information* to decouple trace-local liveness from liveness in other traces. Several related approaches have been proposed in literature.

Pereira and Palsberg's paper on *register allocation via puzzle-solving* [Pereira and Palsberg, 2008] sparked our interest in *Static Single Information* (SSI) form. SSI form, which was proposed by Ananian [1999] and later refined by others [Singer, 2006; Boissinot et al., 2012], is an extension to the SSA form. In addition to the $\varphi$ functions at control-flow joins, SSI also has $\sigma$-functions[12] at control-flow splits. Basically, in SSI form, every *usage* of a variable *post-dominates* its definition.

[11] Although there are exceptions like in the approach by Häubl and Mössenböck [2011].
[12] Sometimes called $\pi$-functions.




The example from Figure 4.5 in SSI form. Note the $\sigma_{out}$ in $b_0$, which renames `totalFS` in the successor branches $b_1$ and $b_2$. While SSI form is sufficient to keep `totalFS` *live* until the end of $b_2$, the liveness of **`this`** in $b_2$ is not represented properly, due to *single-entry-single-exit region bypassing* [Singer, 2006]. See Figure 4.6 for our representation, which captures this information.

**Figure 10.2:** Example for a program in SSI form

See Figure 10.2 for an example. SSI form has several interesting properties. Most important for register allocation, the interference graph is an *interval graph* (Definition 32) [Boissinot et al., 2012]. As we have already mentioned above, interval graphs allow solving the *k-colorable subgraph* problem in polynomial time.

Pereira and Palsberg [2008] exploited a further extension to SSI form, namely *elementary programs*, in their register allocator based on *puzzle-solving*. In addition to $\varphi$- and $\sigma$-functions, they insert parallel copies of all live variables in-between two instructions of a basic block to rename variables. That way not only definitions and usages are unique, but also liveness of a variable at every program point.

Although we borrowed ideas from the SSI representation and elementary programs, our IR does not adhere to their definitions. First, SSI form might not provide all the liveness information that we need. A $\sigma$-function (and the corresponding $\varphi$-function) might be omitted for variables that are defined before a split, but only used after the corresponding control-flow join. Singer [2006] calls this property *single-entry-single-exit region bypassing*. See the variable **`this`** in Figure 10.2. This defeats the purpose of our liveness representation, since we need exact liveness information in every trace. Also, we avoid excessive renaming of variables for the sake of fewer instruction

redefinitions. Still, our global liveness model (Section 4.3.2) provides properties to make register allocation simpler, including interval interference graphs (Corollary 3) and the independent allocation of traces.

Lozano et al. [2012] introduced *Linear Static Single Assignment* (LSSA) form as an intermediate representation for their *constraint-based register allocation and instruction selection* approach. The *in- and out-delimiters* in LSSA form are related to our representation. In our approach, however, we distinguish between $live_{in}/live_{out}$ and $\varphi_{in}/\varphi_{out}$ sets. While the latter are a proper part of the IR, global liveness information is stored off-site.

## 10.5 Compile-time Trade-offs and Concurrent Compilation

The trade-off between time spent for executing application code and time spent in the runtime is an important design parameter for a virtual machine. For many virtual machines, such as the early Java VMs, portability was the main focus. The deployed code, e.g. Java bytecode, is executed by an interpreter. Although optimization techniques were proposed, for example by [Ertl and Gregg, 2003], the execution speed is limited. Many modern virtual machines use dynamic compilation to produce efficient native machine code. However, for such systems the time constraints for the compiler are very strict.

Some virtual machines, e.g., the CACAO VM [Krall, 1998] or the Jalapeño VM [Arnold et al., 2000], follow a compile-only approach. This means that there is no interpreter but a *baseline* compiler. Since every method needs to be compiled before it is executed, this compiler must be fast. The CACAO baseline compiler therefore performs optimizations only on a local scope. Systems such as the Jalapeño VM [Arnold et al., 2000] or the HotSpot VM [Paleczny et al., 2001], introduce adaptive compilation, i.e., dynamic compilation and optimization of the most relevant parts based on the current execution profile. They use multiple optimization stages that are invoked for performance-critical parts only. While Jalapeño uses a baseline compiler, HotSpot VM features an interpreter for the initial execution. Methods are usually selected for optimization based on profiling information, for instance invocation and loop counters, or stack sampling. Although, these systems can select thresholds to control the compile time, they can do so only on a per-method basis. Our trace register allocation policy approach is orthogonal to that. For a compilation that is considered *hot* by the virtual machine, we can make a fine-grained compile time vs. peak performance decision.

Just-in-time compilers often apply concurrency techniques to improve compilation performance. The HotSpot VM, for example, executes the compilers in background threads [Kotzmann et al., 2008]. So does V8, Googles JavaScript engine [McIlroy, 2018]. The advantage of this approach is

that the main thread can continue executing the program, for example, in an interpreter, while a method is compiled. HotSpot also uses multiple compiler threads [Oracle Corporation, 2015]. The synchronization overhead is low. Compilation threads need to synchronize at the beginning when taking a method from the compilation queue and at the end when installing the code. The compiler itself can be single-threaded. Also, adding more compiler threads scales well, as long as there are enough methods in the queue. Background compilation on multiple threads improves throughput, i.e., the number of units compiled in a given time frame. However, it cannot improve compilation latency, i.e., the time that is required to compile a given method. This is in contrast to our parallel register allocation approach where multiple threads can reduce the time required for compiling a single method. However, the scaling potential is bound by the number and the dependencies of traces.

# Chapter 11

# Conclusion and Future Work

Register allocation is a task that almost every compiler writer has to deal with. Most register allocation approaches fall into one of two categories, *local register allocation* or *global register allocation*. Local register allocators solve the problem for a single basic block. The absence of control flow guarantees *liveness* properties, most importantly the absence of lifetime holes, which renders the problem easy to tackle. However, the narrow scope limits the opportunities for optimizations. Especially spill decisions and spill placement suffer from this constraint. Global register allocators, on the other hand, deal with the whole method at once. Their omniscient view allows them to optimize decisions in a way that improves the quality of the allocation. This comes at a cost. First, these optimizations require a lot of contextual information which needs to be computed and/or stored. Second, a global allocator has to deal with control flow, which in turn complicates the maintenance of the *liveness*, which is the central piece of information the allocator deals with. Consequently, global register allocators are slower in terms of compilation time and more complex in terms of implementation, compared to a local approach. Usually, a compiler creator needs to choose. Either use a fast and simple-to-implement local register allocator and accept suboptimal allocation quality, or implement a complex and slower global allocator to achieve better allocation results. Existing approaches did not offer a middle ground.

We wanted to gain the advantages of both worlds. The simplicity of a local allocator and the allocation quality of a global approach. In this thesis, we presented the *trace register allocation framework*, a novel, flexible, non-global and extensible register allocation approach, that combines the advantages of local and global allocators. The approach works in two phases. A *global* pre- and post-processing phase that deals with control flow and liveness, and the actual allocation phase. Instead of processing a whole method at once, the framework divides the control-flow graph into traces, i.e., lists of sequentially executed basic blocks. The allocator then handles one trace at a time. From the allocator's perspective, a trace is no different than a basic block. There is no need to deal with control flow and the liveness behaves nicely. Therefore, we can apply simple algorithms used for local register allocation. This is possible due to the information the framework collects in the pre-processing phase. Register allocation can be done independently

for each trace. This flexibility offers novel opportunities. First, we can choose different allocation algorithms for each trace. This allows us to invest more time on the important traces of the methods and reduce the time spent on insignificant parts. We can make this decision based on the structure of the trace, its expected execution frequency, or due to the compile-time budget. Second, since traces can be processed independently, they can be allocated in parallel by multiple threads and thus, compilation latency can be reduced without a negative impact on allocation quality.

Although the simplicity and flexibility of our trace register allocation approach is compelling, we wanted to convince ourselves that the idea can be successfully applied in practice. To quantify *success*, we posed four research questions: **RQ1**: Can trace register allocation produce the same code quality as state-of-the-art global allocators, despite the limited scope of traces? **RQ2**: Can the result be found in the same amount of time? Once these two questions were answered, we could focus on exploiting the unique flexibility of trace register allocation. Thus we stated **RQ3**: Can different allocation strategies help us to better control the trade-off between compile time and peak performance? Finally, we posed **RQ4**: Can the compilation latency, i.e., the duration required to compile a given method, be reduced by allocating traces concurrently by multiple threads without impacting allocation quality?

All four research questions could be answered positively. We implemented trace register allocation in GraalVM, a production-quality Java Virtual Machine, and compared it to the existing register allocator of Graal, a global linear scan implementation. In Section 7.3 we showed that for common benchmarks there is virtually no difference in terms of allocation quality (**RQ1**). This result is remarkable, because it suggests that a global view on the problem is not necessary to achieve good performance. We also showed that trace register allocation is at least as fast as global linear scan, sometimes even slightly faster (**RQ2**, Section 7.4). Especially for large compilation units, traces offer a compile time advantage over the existing allocator. Section 8.3 demonstrates how the careful selection of trace register allocation strategies allows us to gradually reduce register allocation time from 0 to 40%, depending on how much peak performance we are willing to sacrifice (ranging from 0–12%, on average) (**RQ3**). Finally, in Section 9.2, we showed that register allocation can be parallelized without a negative effect on peak performance. With our early prototype we were able to reduce register allocation time by 30% when using 4 threads compared to a single allocation thread (**RQ4**).

In our research, we focused primarily on achieving better compile-time behavior than existing global approaches. In terms of allocation quality, we were satisfied with reaching the same results as the global linear scan allocator. We think that exploring further improvements of the allocation quality a highly interesting research direction. From our perspective, this poses the following challenges. First, the question is whether there is even potential for improvement. We believe that this question should be answered before investing resources on this topic. Ideally, the

experimentation platform, be it Graal or any other system, should be equipped with an allocator that finds an optimal or near-optimal allocation. Only if this uncovers missed opportunities that really make a difference, working towards improving the allocation quality of the trace register allocator makes sense. The absence of this upper bound on allocation quality is one of the reasons why we focused on compile time first. This brings us to the second challenge. One of the key principles of trace register allocation is to reduce the allocation scope and to divide the problem into smaller sub-problems that are easier to solve than the whole problem. While this simplifies the problem we need to solve, it also reduces the amount of information that is available. In this respect, a global allocator has an immanent advantage over the trace-based approach (whether this advantage is exploited or not). We think that one of most the promising ideas to improve allocation quality of a trace register allocator is to apply methods, that are too costly for the global scope, to a single trace, or a set of important traces. We envision strategies that compute the optimal allocation for a few traces, while the rest of the method uses the heuristic methods that we described here.

The idea of trace-based processing does not end with register allocation. In fact, similar ideas were already applied to instruction scheduling decades ago. We are confident that other optimizations can also profit from the approaches we developed for trace register allocation. We believe that our results lay the foundation for future research in the area of trace-based optimizations, and in particular to trace register allocation. We are convinced that the flexibility of our approach can push the boundaries of current register allocation and optimization techniques and can have an impact on both, research and production compilers.

# Appendix A

# Additional Sources

**Listing 1** Fields of the `TraceInterval` Class

```
class TraceInterval {

    /** The variable for this interval prior.  */
    final Variable operand;

    /** The start of the range, inclusive. */
    int intFrom = Integer.MAX_VALUE;

    /** The end of the range, exclusive. */
    int intTo = Integer.MAX_VALUE;

    /** List of (use-positions, register-priorities) pairs, sorted by use-positions. */
    int[] usePosListArray;

    /** The register or spill slot assigned to this interval. */
    AllocatableValue location;

    /** The stack slot to which all splits of this interval are spilled if necessary. */
    AllocatableValue spillSlot;

    /* ... */
}
```

**Listing 2** Java Source of `StackSlot.getOffset()`

```
class StackSlot {

    int offset;
    boolean addFrameSize;

    int getOffset(int totalFrameSize) {
        final int add;
        if (!this.addFrameSize) {
            add = 0;
        } else {
            add = totalFrameSize;
        }
        return this.offset + add;
    }

    /* ... */
}
```

# Appendix B

# Graal Backends

The basic *parameters* for register allocation in GraalVM depends on three factors: the architecture (processor), the virtual machine, and the operating system. We evaluated Graal on top of the HotSpot VM, with AMD64 on Linux and SPARC on Solaris.

## B.1 AMD64 on HotSpot

For AMD64 on HotSpot, the configuration is done by the `AMD64HotSpotRegisterConfig` class, which is part of JVMCI [2014]. There are 14 general purpose registers, namely `rax`, `rcx`, `rdx`, `rbx`, `rbp`, `rsi`, `rdi`, and `r8–r14`. The register `rsp` is used as the *frame pointer*, `r15` is the *thread register* which contains a pointer to the current thread storage (e.g., to access `ThreadLoacals`). If *compressed object pointers* [Oracle Corporation, 2018] are enabled, `r12` is used as the *heap base register*. HotSpot uses 16 floating point registers, `xmm0–xmm15`. If the processor supports AVX512, there are 16 additional registers `xmm16–xmm31`.

The calling convention for Java methods uses 6 general purpose registers (`rsi`, `rdx`, `rcx`, `r8`, `r9`, `rdi`) for integers and reference types, and 8 floating point registers (`xmm0–xmm7`). Additional parameters are passed via the stack. The `rbp` register is an implicit parameter, which contains the frame pointer of the calling method. However, it can be reassigned, just like other parameter registers.

Integer values are returned via `rax`, floating point values via `xmm0`. All allocatable registers are *caller-saved*, i.e., they need to be spilled before every call.

## B.2 SPARC on HotSpot

The register set on SPARC is slightly more complex. There are 23 general purpose register, `g1`, `g4` and `g5`, `o0–o5`, `l0–l7`, and `i0–i5`. The frame pointer is in `o6`, `g0` is always zero, and other registers are system-reserved. The register `g6` is the optional heap base register. SPARC has separate registers for *single precision* (`float`, `f8–31`) and *double precision* (`double`, register pairs `d32_d33–d62_d63`) floating point values.

Integer parameters are passed via `i0–i5` (callee side), single precision floats via `f0–f7`, double precision floating point values via `d0_d1–d6_d7`. The return register is either `i0`, `f0`, or `d0_d1`.

The SPARC architecture features a *register window*. At every procedure call the registers `i0–i7` and `l0–l7` are automatically stored in a predefined location on the stack. In addition, the registers `o0–o7` of the caller are mapped to the `i0–i7` of the callee (that includes the parameter registers). The compiler does not need to spill those registers across a call.

# Appendix C

# Hardware Environment

The experiments on AMD64 were performed on two kinds of machines, the Sun Server X3-2 and Sun Server X5-2. The machines were running an Oracle Linux Server 6.8 operating system with Linux Kernel version 4.1.12. For the experiments we disabled all frequency scaling modes (e.g. scaling governors or Intel Turbo Boost).

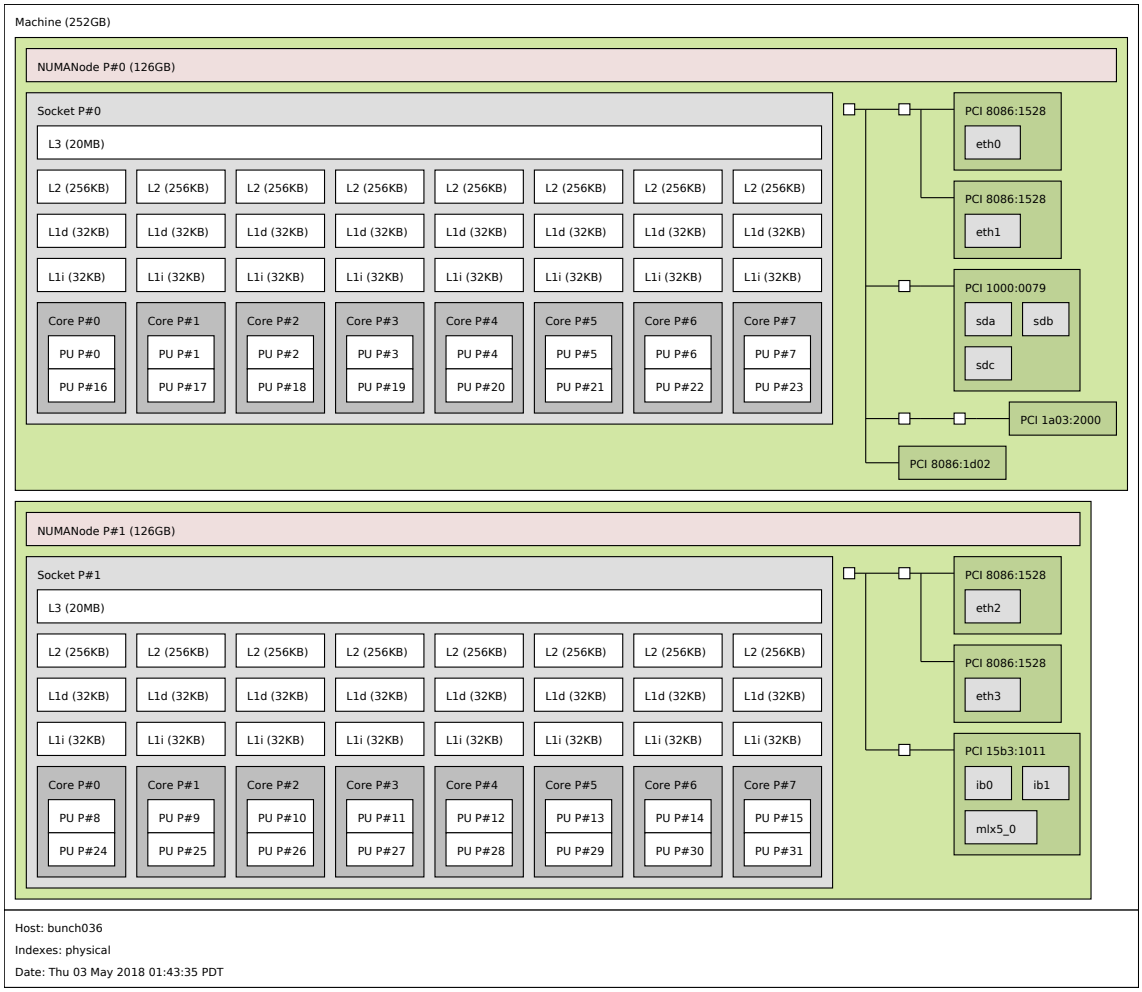
## C.1 Sun Server X3-2

The X3-2 cluster consists of 64 identical Sun Server X3-2 machines,[1] equipped with two Intel "Sandy Bridge" Xeon E5-2660 at 2.20GHz with 8 real cores per processor, and 256GB of DDR3-1600 memory. Each core has 2 threads. The cache sizes are 32KB for L1, 265KB for L2 and 20MB L3. Figure C.1 shows the hardware topology of the machine.

## C.2 Sun Server X5-2

The X5-2 cluster consists of 36 identical Sun Server X5-2 machines,[2] equipped with two Intel "Haswell" Xeon E5-2699 v3 at 2.30GHz with 18 real cores per processor, and 384GB of main memory. Each core features 2 threads. The cache sizes are 32KB for L1, 265KB for L2 and 45MB L3. Figure C.2 shows the hardware topology of the machine.

[1] Sun Server X3-2: `http://www.oracle.com/goto/x3-2/docs`
[2] Sun Server X5-2: `http://www.oracle.com/goto/x5-2/docs`




Topology of the X3-2 system collected with the `lstopo`[a] utility.

[a] lstopo(1) — Linux man page: `https://linux.die.net/man/1/lstopo`

**Figure C.1:** Sun Server X3-2 topology


Topology of the X5-2 system collected with `lstopo`.

**Figure C.2:** Sun Server X5-2 topology

## C.3 SPARC T7-2 Server

For our SPARC/Solaris experiments we used a SPARC T7-2 Server[3] running the Solaris 11.3 operating system.[4] It is equipped with two SPARC M7 processors, which are SPARC V9 architectures [Weaver and Germond, 1994]. A single M7 is running at 4.13 GHz and features 32 cores with 8 threads each. Consequently, the T7-2 machine has 64 cores and 512 threads, respectively. Each core has 16KB L1 cache, 256KB of L2 per core pair, and 64MB per chip. Our system features 2×500GB of main memory, one for each processor. We use *Solaris Kernel Zones*[5] to divide the T7-2 into 8 isolated zones. Each zone can use 4 dedicated cores and has access to 96GB of memory. The remaining cores and memory are left for the root zone.

[3]SPARC T7-2 Server: `http://www.oracle.com/goto/t7-2/docs`
[4]Solaris 11.3 `https://www.oracle.com/solaris/oracle-solaris-11-3.html`
[5]Solaris Kernel Zones: `http://docs.oracle.com/cd/E36784_01/pdf/E36848.pdf`

# List of Figures

# List of Algorithms



# List of Tables

# Index

# Publications

Eisl, Josef (2015). "Trace Register Allocation". In: *SPLASH Companion 2015*. ACM. DOI: 10.1145/2814189.2814199. *(SPLASH 2015 Doctoral Symposium Vision Paper)*

Eisl, Josef (2018a). "Divide and Allocate: The Trace Register Allocation Framework". In: *CGO'18*. ACM. PREPRINT: http://ssw.jku.at/General/Staff/Eisl/papers/CGO_SRC2018.pdf *(Peer Reviewed Extended Abstract, Winner CGO ACM Student Research Competition Graduate Category)*

Eisl, Josef (2018b). "Divide and Allocate: The Trace Register Allocation Framework (Extended Grand Finals Version)". In: *SRC'18*. ACM. URL: https://src.acm.org/binaries/content/assets/src/2018/josef-eisl.pdf. *(Submitted to the ACM Student Research Competition Grand Finals 2018)*

Eisl, Josef, Matthias Grimmer, Doug Simon, Thomas Würthinger, and Hanspeter Mössenböck (2016). "Trace-based Register Allocation in a JIT Compiler". In: *PPPJ'16*. ACM. DOI: 10.1145/2972206.2972211.

Eisl, Josef, David Leopoldseder, and Hanspeter Mössenböck (2018). "Parallel Trace Register Allocation". In: *ManLang 2018*. ACM. DOI: 10.1145/3237009.3237010.

Eisl, Josef, Stefan Marr, Thomas Würthinger, and Hanspeter Mössenböck (2017). "Trace Register Allocation Policies: Compile-time vs. Performance Trade-offs". In: *ManLang 2017*. ACM. DOI: 10.1145/3132190.3132209.

Leopoldseder, David, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck (2018). "Dominance-based Duplication Simulation (DBDS) – Code Duplication to Enable Compiler Optimizations". In: *CGO'18*. ACM. DOI: 10.1145/3168811. *(best paper finalist)*

# Bibliography

Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley. ISBN: 9780201000290.

Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc. ISBN: 0321486811. URL: http://dragonbook.stanford.edu/.

Ananian, C. Scott (1999). "The Static Single Information Form". MA thesis. Princeton University. URL: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-801.pdf.

Appel, Andrew W. (1998). *SSA is Functional Programming*.

Appel, Andrew W. and Jens Palsberg (2003). *Modern Compiler Implementation in Java*. 2nd. Cambridge University Press. ISBN: 052182060X.

Arnold, Matthew, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney (2000). "Adaptive Optimization in the Jalapeño JVM". In: *OOPSLA '00*. ACM. DOI: 10.1145/353171.353175.

Bala, Vasanth, Evelyn Duesterwald, and Sanjeev Banerjia (2000). "Dynamo: A Transparent Dynamic Optimization System". In: *PLDI'00*. ACM. DOI: 10.1145/349299.349303.

Barik, Rajkishore, Jisheng Zhao, and Vivek Sarkar (2013). "A Decoupled non-SSA Global Register Allocation Using Bipartite Liveness Graphs". In: *TACO'13*. DOI: 10.1145/2544101.

Barrett, Edd, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt (2017). "Virtual Machine Warmup Blows Hot and Cold". In: *Proc. ACM Program. Lang.* DOI: 10.1145/3133876.

Belady, L. A. (1966). "A study of replacement algorithms for a virtual-storage computer". In: *IBM Systems Journal*. DOI: 10.1147/sj.52.0078.

Bergner, Peter, Peter Dahl, David Engebretsen, and Matthew O'Keefe (1997). "Spill Code Minimization via Interference Region Spilling". In: *PLDI'97*. ACM. DOI: 10.1145/258915.258941.

Blackburn, S. M., R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann (2006). "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA'06*. ACM Press. DOI: 10.1145/1167473.1167488.

Boissinot, Benoit, Philip Brisk, Alain Darte, and Fabrice Rastello (2012). "SSI Properties Revisited". In: *ACM Trans. Embed. Comput. Syst.* DOI: 10.1145/2180887.2180898.

Bolz, Carl Friedrich, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo (2009). "Tracing the meta-level". In: *ICOOOLPS'09*. DOI: 10.1145/1565824.1565827.

Bouchez, Florent (2009). "A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases". Theses. Ecole normale supérieure de lyon - ENS LYON. URL: https://tel.archives-ouvertes.fr/tel-00403504.

Bouchez, Florent, Alain Darte, Christophe Guillon, and Fabrice Rastello (2005). *Register allocation and spill complexity under SSA*. Tech. rep. École Normale Supérieure de Lyon.

Brandis, Marc M. and Hanspeter Mössenböck (1994). "Single-pass Generation of Static Single-assignment Form for Structured Languages". In: *TOPLAS'94*. DOI: 10.1145/197320.197331.

Braun, Matthias, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau (2013). "Simple and Efficient Construction of Static Single Assignment Form". In: *CC 2013*. Springer-Verlag. DOI: 10.1007/978-3-642-37051-9_6.

Braun, Matthias and Sebastian Hack (2009). "Register Spilling and Live-Range Splitting for SSA-Form Programs". In: *CC 2009*. Springer-Verlag. DOI: 10.1007/978-3-642-00722-4_13.

Briggs, Preston (1992). "Register Allocation via Graph Coloring". PhD thesis. Rice University.

Briggs, Preston, Keith D. Cooper, and Linda Torczon (1994). "Improvements to graph coloring register allocation". In: *TOPLAS'94*. DOI: 10.1145/177492.177575.

Brisk, Philip (2006). "Advances in Static Single Assignment Form and Register Allocation". PhD thesis.

Brisk, Philip, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh (2005). "Polynomial time graph coloring register allocation". In:

Brisk, Philip and Majid Sarrafzadeh (2007). "Interference graphs for procedures in static single information form are interval graphs". In: *SCOPES'07*. ACM. DOI: 10.1145/1269843.1269858.

Buchwald, Sebastian, Manuel Mohr, and Ignaz Rutter (2015). "Optimal Shuffle Code with Permutation Instructions". In: Springer International Publishing. DOI: 10.1007/978-3-319-21840-3_44.

Callahan, David and Brian Koblenz (1991). "Register Allocation via Hierarchical Graph Coloring". In: *SIGPLAN Not.* DOI: 10.1145/113446.113462.

Cavazos, John, J. Eliot B. Moss, and Michael F. P. O'Boyle (2006). "Hybrid Optimizations: Which Optimization Algorithm to Use?" In: *CC 2000*. Springer Berlin Heidelberg. DOI: 10.1007/11688839_12.

Chaitin, Gregory J. (1982). "Register Allocation & Spilling via Graph Coloring". In: *SIGPLAN Not.* DOI: 10.1145/872726.806984.

Chaitin, Gregory J., Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein (1981). "Register Allocation via Coloring". In: *Computer languages*. DOI: 10.1016/0096-0551(81)90048-5.

Chen, Wei-Yu, Guei-Yuan Lueh, Pratik Ashar, Kaiyu Chen, and Buqi Cheng (2018). "Register Allocation for Intel Processor Graphics". In: *CGO'18*. ACM. DOI: 10.1145/3168806.

Chow, Fred C. and John L. Hennessy (1990). "The priority-based coloring approach to register allocation". In: *ACM Trans. Program. Lang. Syst.* DOI: 10.1145/88616.88621.

Clang (2018). *clang: a C language family frontend for LLVM*. URL: https://clang.llvm.org/ (visited on 01/12/2018).

Click, Cliff and Michael Paleczny (1995). "A simple graph-based intermediate representation". In: *IR'95*. ACM. DOI: 10.1145/202529.202534.

Click, Clifford Noel (1995). "Combining Analyses, Combining Optimizations". PhD thesis. Rice University.

Cook, Stephen A. (1971). "The complexity of theorem-proving procedures". In: *STOC'71*. ACM. DOI: 10.1145/800157.805047.

Cooper, Keith and Linda Torczon (2011). *Engineering a compiler*. 2nd ed. Elsevier. ISBN: 9780120884780.

Cytron, Ron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck (1991). "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *TOPLAS'91*. DOI: 10.1145/115372.115320.

Deutsch, L. Peter and Allan M. Schiffman (1984). "Efficient Implementation of the Smalltalk-80 System". In: *POPL '84*. ACM. DOI: 10.1145/800017.800542.

Duboscq, Gilles (2016). "Combining Speculative Optimizations with Flexible Scheduling of Side-effects". PhD thesis. Johannes Kepler University Linz. URL: http://resolver.obvsg.at/urn:nbn:at:at-ubl:1-9708.

Duboscq, Gilles, Thomas Würthinger, and Hanspeter Mössenböck (2014). "Speculation without regret". In: *PPPJ'14*. DOI: 10.1145/2647508.2647521.

Duboscq, Gilles, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck (2013). "An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler". In: *VMIL'13*. DOI: 10.1145/2542142.2542143.

Ebner, Dietmar, Bernhard Scholz, and Andreas Krall (2009). "Progressive Spill Code Placement". In: *CASES '09*. ACM. DOI: 10.1145/1629395.1629408.

Eisl, Josef (2015). "Trace Register Allocation". In: *SPLASH Companion 2015*. ACM. DOI: 10.1145/2814189.2814199. *(SPLASH 2015 Doctoral Symposium Vision Paper)*

Eisl, Josef (2018a). "Divide and Allocate: The Trace Register Allocation Framework". In: *CGO'18*. ACM. PREPRINT: http://ssw.jku.at/General/Staff/Eisl/papers/CGO_SRC2018.pdf *(Peer Reviewed Extended Abstract, Winner CGO ACM Student Research Competition Graduate Category)*

Eisl, Josef (2018b). "Divide and Allocate: The Trace Register Allocation Framework (Extended Grand Finals Version)". In: *SRC'18*. ACM. URL: https://src.acm.org/binaries/content/assets/src/2018/josef-eisl.pdf. *(Submitted to the ACM Student Research Competition Grand Finals 2018)*

Eisl, Josef, Matthias Grimmer, Doug Simon, Thomas Würthinger, and Hanspeter Mössenböck (2016). "Trace-based Register Allocation in a JIT Compiler". In: *PPPJ'16*. ACM. DOI: 10.1145/2972206.2972211.

Eisl, Josef, David Leopoldseder, and Hanspeter Mössenböck (2018). "Parallel Trace Register Allocation". In: *ManLang 2018*. ACM. DOI: 10.1145/3237009.3237010.

Eisl, Josef, Stefan Marr, Thomas Würthinger, and Hanspeter Mössenböck (2017). "Trace Register Allocation Policies: Compile-time vs. Performance Trade-offs". In: *ManLang 2017*. ACM. DOI: 10.1145/3132190.3132209.

Ellis, John R. (1985). "Bulldog: A Compiler for VLIW Architectures". PhD thesis. Yale University.

Ertl, M. Anton and David Gregg (2003). "The Structure and Performance of Efficient Interpreters". In: *The Journal of Instruction-Level Parallelism*. URL: http://www.complang.tuwien.ac.at/papers/ertl%26gregg03jilp.ps.gz.

Farach-Colton, Martin and Vincenzo Liberatore (2000). "On Local Register Allocation". In: *Journal of Algorithms*. DOI: 10.1006/jagm.2000.1095.

Fisher, Joseph Allen (1981). "Trace Scheduling: A Technique for Global Microcode Compaction". In: *Computers, IEEE Transactions on Computers*. DOI: 10.1109/TC.1981.1675827.

Flajolet, Philippe and Robert Sedgewick (2009). *Analytic Combinatorics*. Cambridge University Press.

Freudenberger, Stefan M. and John C. Ruttenberg (1992). "Phase Ordering of Register Allocation and Instruction Scheduling". In: *Workshops in Computing*. Springer London. DOI: 10.1007/978-1-4471-3501-2_9.

Gal, Andreas, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz (2009). "Trace-based Just-in-Time Type Specialization for Dynamic Languages". In: *PLDI'09*. ACM. DOI: 10.1145/1542476.1542528.

Gal, Andreas, Christian W. Probst, and Michael Franz (2006). "HotpathVM: An Effective JIT Compiler for Resource-constrained Devices". In: *VEE'06*. ACM. DOI: 10.1145/1134760.1134780.

Gantt, Henry Laurence (1913). *Work, Wages, and Profits*. Second Edition. The Engineering Magazine Co.

Garret, Charles D, Jeffrey Dean, David Grove, and Craig Chambers (1994). *Measurement and Application of Dynamic Receiver Class Distributions*. University of Washington.

Gavril, Fănică (1974). "The intersection graphs of subtrees in trees are exactly the chordal graphs". In: *Journal of Combinatorial Theory, Series B*. DOI: 10.1016/0095-8956(74)90094-X.

GCC (2017). *Integrated Register Allocator in GCC*. URL: https://github.com/gcc-mirror/gcc/blob/216fc1bb7d9184/gcc/ira.c.

George, Lal and Andrew W. Appel (1996). "Iterated register coalescing". In: *TOPLAS'96*. DOI: 10.1145/229542.229546.

Goodwin, David W. and Kent D. Wilken (1996). "Optimal and Near-optimal Global Register Allocation Using 0–1 Integer Programming". In: *Software: Practice and Experience*. DOI: 10.1002/(SICI)1097-024X(199608)26:8<929::AID-SPE40>3.0.CO;2-T.

Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley (2015). *The Java Language Specification: Java SE 8 Edition*. URL: https://docs.oracle.com/javase/specs/jls/se8/html/.

Guo, Jia, Maria Jesus Garzaran, and David Padua (2003). "The Power of Belady's Algorithm in Register Allocation for Long Basic Blocks". In: *LCPC'03*. DOI: 10.1007/978-3-540-24644-2_24.

Hack, Sebastian (2005). *Interference Graphs of Programs in SSA-form*. Tech. rep. Universität Karlsruhe. URL: http://compilers.cs.uni-saarland.de/papers/ifg_ssa.pdf.

Hack, Sebastian (2007). "Register Allocation for Programs in SSA Form". PhD thesis. Universität Karlsruhe. ISBN: 978-3-86644-180-4. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/6532.

Hack, Sebastian, Daniel Grund, and Gerhard Goos (2006). "Register Allocation for Programs in SSA-Form". In: *CC'06*. Springer Berlin Heidelberg. DOI: 10.1007/11688839_20.

Hames, Lang and Bernhard Scholz (2006). "Nearly Optimal Register Allocation with PBQP". In: Springer Berlin Heidelberg. DOI: 10.1007/11860990_21.

Hank, R.E., W.W. Hwu, and B.R. Rau (1995). "Region-based compilation: an introduction and motivation". In: *MICRO'95*. IEEE Comput. Soc. Press. DOI: 10.1109/micro.1995.476823.

Häubl, Christian and Hanspeter Mössenböck (2011). "Trace-based compilation for the Java HotSpot virtual machine". In: *PPPJ'11*. DOI: 10.1145/2093157.2093176.

Hennessy, John L. and David A. Patterson (2003). *Computer Architecture: A Quantitative Approach*. 3rd ed. Morgan Kaufmann Publishers Inc. ISBN: 1558607242.

Hölzle, Urs, Craig Chambers, and David Ungar (1992). "Debugging Optimized Code with Dynamic Deoptimization". In: *SIGPLAN Not.* DOI: 10.1145/143103.143114.

Horwitz, L. P., R. M. Karp, R. E. Miller, and S. Winograd (1966). "Index Register Allocation". In: *J. ACM*. DOI: 10.1145/321312.321317.

Hwu, Wen -Mei W., Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, and et al. (1993). "The superblock: An effective technique for VLIW and superscalar compilation". In: *The Journal of Supercomputing*. DOI: 10.1007/bf01205185.

Intel (2013a). *Intel® 64 and IA-32 Architectures Software Developer's Manual — Volume 1 Basic Architecture*. URL: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

Intel (2013b). *Intel® 64 and IA-32 Architectures Software Developer's Manual — Volume 2 Instruction Set Reference*. URL: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

JVMCI (2014). *JEP 243: Java-Level JVM Compiler Interface*. URL: http://openjdk.java.net/jeps/243 (visited on 05/06/2016).

Karp, Richard M (1972). *Reducibility among Combinatorial Problems*. Springer.

Koes, David Ryan and Seth Copen Goldstein (2006a). *A better global progressive register allocator*. URL: http://www.cs.cmu.edu/~dkoes/research/lctes06_tracealloc.pdf. *(Poster at LCTES 2006)*

Koes, David Ryan and Seth Copen Goldstein (2006b). "A global progressive register allocator". In: *PLDI'06*. DOI: 10.1145/1133981.1134006.

Koes, David and Seth Copen Goldstein (2005). "A Progressive Register Allocator for Irregular Architectures". In: *CGO '05*. IEEE Computer Society. DOI: 10.1109/CGO.2005.4.

Kotzmann, Thomas, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox (2008). "Design of the Java HotSpot™ client compiler for Java 6". In: *TACO'08*. DOI: 10.1145/1369396.1370017.

Krall, Andreas (1998). "Efficient JavaVM Just-in-Time Compilation". In: *PACT'98*. IEEE Computer Society. DOI: 10.1109/PACT.1998.727250.

Krintz, Chandra J., David Grove, Vivek Sarkar, and Brad Calder (2001). "Reducing the overhead of dynamic compilation". In: *Software: Practice and Experience*. DOI: 10.1002/spe.384.

Lattner, Chris and Vikram Adve (2004). "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *CGO'04*. IEEE Computer Society. DOI: 10.1109/CGO.2004.1281665.

Lee, Jonathan K., Jens Palsberg, and Fernando Magno Quintão Pereira (2007). "Aliased Register Allocation for Straight-Line Programs Is NP-Complete". In: DOI: 10.1007/978-3-540-73420-8_59.

Lekkeikerker, C. and J. Boland (1962). "Representation of a finite graph by a set of intervals on the real line". In: *Fundamenta Mathematicae*. URL: http://eudml.org/doc/213681.

Lengauer, Philipp, Verena Bitto, Stefan Fitzek, Markus Weninger, and Hanspeter Mössenböck (2016). "Efficient Memory Traces with Full Pointer Information". In: *PPPJ '16*. ACM. DOI: 10.1145/2972206.2972220.

Leopoldseder, David, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck (2018). "Dominance-based Duplication Simulation (DBDS) – Code Duplication to Enable Compiler Optimizations". In: *CGO'18*. ACM. DOI: 10.1145/3168811. *(best paper finalist)*

Lindholm, T., F. Yellin, G. Bracha, and A. Buckley (2015). *The Java Virtual Machine Specification: Java SE 8 Edition*. URL: http://docs.oracle.com/javase/specs/jvms/se8/html/.

Lowney, P. Geoffrey, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'donnell, and John C. Ruttenberg (1993). "The Multiflow Trace Scheduling Compiler". In: *Journal of Supercomputing*. DOI: 10.1007/BF01205182.

Lozano, Roberto Castañeda, Mats Carlsson, Frej Drejhammar, and Christian Schulte (2012). "Constraint-Based Register Allocation and Instruction Scheduling". In: *Principles and Practice of Constraint Programming*. DOI: 10.1007/978-3-642-33558-7_54.

Lozano, Roberto Castañeda, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte (2014). "Combinatorial spill code optimization and ultimate coalescing". In: *LCTES'14*. ACM. DOI: 10.1145/2597809.2597815.

Lueh, Guei-Yuan, Thomas Gross, and Ali-Reza Adl-Tabatabai (1997). "Global register allocation based on graph fusion". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. DOI: 10.1007/BFb0017257.

Mayoh, B., E. Tyugu, and J. Penjam (2013). *Constraint Programming. Nato ASI Subseries F:* Springer Berlin Heidelberg. ISBN: 9783642859830.

McIlroy, Ross (2018). *V8 JavaScript Engine: Background compilation*. URL: https://v8project.blogspot.co.at/2018/03/background-compilation.html (visited on 03/28/2018).

Mohr, M., A. Grudnitsky, T. Modschiedler, L. Bauer, S. Hack, and J. Henkel (2013). "Hardware acceleration for programs in SSA form". In: DOI: 10.1109/CASES.2013.6662518.

OpenJDK (2017a). *Chaitin Allocator in C2*. URL: http://hg.openjdk.java.net/jdk/hs/file/5caa1d5f74c1/src/hotspot/share/opto/chaitin.hpp.

OpenJDK (2017b). *Linear Scan Register Allocator in C1*. URL: http://hg.openjdk.java.net/jdk/hs/file/5caa1d5f74c1/src/hotspot/share/c1/c1_LinearScan.hpp.

OpenJDK (2018). *OpenJDK: Project Metropolis*. URL: http://openjdk.java.net/projects/metropolis/ (visited on 05/28/2018).

Oracle Corporation (2016). *Java SE HotSpot at a Glance*. URL: http://www.oracle.com/technetwork/articles/javase/index-jsp-136373.html (visited on 05/06/2016).

Oracle Corporation (2012). *Oracle SPARC Architecture 2011*. Draft D0.9.5d.

Oracle Corporation (2014). *Introduction to Oracle Solaris Zones*. URL: http://docs.oracle.com/cd/E36784_01/pdf/E36848.pdf (visited on 05/26/2016).

Oracle Corporation (2015). *JRockit to HotSpot Migration Guide: Compilation Optimization*. URL: https://docs.oracle.com/javacomponents/jrockit-hotspot/migration-guide/comp-opt.htm (visited on 03/28/2018).

Oracle Corporation (2017). *Java HotSpot™ Virtual Machine Performance Enhancements*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html (visited on 01/09/2018).

Oracle Corporation (2018). *Java Platform, Standard Edition Tools Reference – java*. URL: https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html (visited on 05/28/2018).

Paleczny, Michael, Christopher Vick, and Cliff Click (2001). "The Java HotSpot™ Server Compiler". In: *JVM'01*. USENIX Association. URL: https://www.usenix.org/events/jvm01/full_papers/paleczny/paleczny.pdf.

Pereira, Fernando Magno Quintão (2008). "Register Allocation by Puzzle Solving". PhD thesis. University of California Los Angeles.

Pereira, Fernando Magno Quintão and Jens Palsberg (2005). "Register allocation via coloring of chordal graphs". In:

Pereira, Fernando Magno Quintão and Jens Palsberg (2008). "Register allocation by puzzle solving". In: *PLDI'08*. DOI: 10.1145/1375581.1375609.

Pinedo, Michael L. (2016). *Scheduling: Theory, Algorithms, and Systems*. 5th ed. Springer International Publishing. DOI: 10.1007/978-3-319-26580-3.

Poletto, Massimiliano, Dawson R. Engler, and M. Frans Kaashoek (1997). "tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation". In: *PLDI'97*. ACM. DOI: 10.1145/258915.258926.

Poletto, Massimiliano and Vivek Sarkar (1999). "Linear Scan Register Allocation". In: *TOPLAS'99*. DOI: 10.1145/330249.330250.

Prokopec, Aleksandar, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger (2017). "Making Collection Operations Optimal with Aggressive JIT Compilation". In: *SCALA 2017*. ACM. DOI: 10.1145/3136000.3136002.

Rastello, Fabrice, ed. (2013). *SSA-based Compiler Design*. Springer. URL: ssabook.gforge.inria.fr/latest/book.pdf. *(Not Yet Published)*

Rong, Hongbo (2009). "Tree Register Allocation". In: *MICRO 42*. ACM. DOI: 10.1145/1669112.1669123.

Sarkar, Vivek and Rajkishore Barik (2007). "Extended Linear Scan: An Alternate Foundation for Global Register Allocation". In: *CC'07*. Springer-Verlag. DOI: 1759937.1759950.

Schatz, R. and H. Prähofer (2013). "Trace-Guided Synthesis of Reactive Behavior Models of Programmable Logic Controllers". In: DOI: 10.1109/SEAA.2013.37.

Scholz, Bernhard and Erik Eckstein (2002). "Register Allocation for Irregular Architectures". In: *SIGPLAN Not.* DOI: 10.1145/566225.513854.

Sewe, Andreas, Mira Mezini, Aibek Sarimbekov, and Walter Binder (2011). "Da capo con scala". In: *OOPSLA'11*. DOI: 10.1145/2048066.2048118.

Singer, Jeremy (2006). *Static program analysis based on virtual register renaming*. Tech. rep. University of Cambridge. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-660.pdf.

SPEC (2017). *SPECjbb2015 Benchmark Design Document*. URL: https://www.spec.org/jbb2015/docs/designdocument.pdf.

*SPECjbb2015: Java Server Benchmark* (2016). URL: https://www.spec.org/jbb2015/ (visited on 05/25/2016).

*SPECjvm2008: Java Virtual Machine Benchmark* (2015). URL: https://www.spec.org/jvm2008/ (visited on 06/15/2015).

Sreedhar, Vugranam C., Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam (1999). "Translating Out of Static Single Assignment Form". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. DOI: 10.1007/3-540-48294-6_13.

Stadler, Lukas, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon (2013). "An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance". In: *SCALA'13*. ACM. DOI: 10.1145/2489837.2489846.

Stadler, Lukas, Thomas Würthinger, and Hanspeter Mössenböck (2014). "Partial Escape Analysis and Scalar Replacement for Java". In: *CGO '14*. ACM. DOI: 10.1145/2544137.2544157.

Steiner, Edwin (2007). "Adaptive Inlining and On-Stack Replacement in a Java Virtual Machine". MA thesis. Vienna University of Technology.

Steiner, Edwin, Andreas Krall, and Christian Thalinger (2007). "Adaptive Inlining and On-Stack Replacement in the CACAO Virtual Machine". In: *PPPJ'07*. ACM. DOI: 10.1145/1294325.1294356.

Tavares, André L. C., Quentin Colombet, Mariza A. S. Bigonha, Christophe Guillon, Fernando M. Q. Pereira, and Fabrice Rastello (2011). "Decoupled graph-coloring register allocation with hierarchical aliasing". In: *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems - SCOPES '11*. DOI: 10.1145/1988932.1988934.

Traub, Omri, Glenn Holloway, and Michael D. Smith (1998). "Quality and Speed in Linear-scan Register Allocation". In: *PLDI'98*. ACM. DOI: 10.1145/277650.277714.

Tukey, John W. (1977). *Exploratory data analysis*. Reading, Mass.

V8 (2017). *Linear Scan Register Allocator in V8*. URL: https://github.com/v8/v8/blob/d16b45ebf8bf/src/compiler/register-allocator.h.

Weaver, David L. and Tom Germond (1994). *The SPARC Architecture Manual – Version 9*. Version 9. Prentice Hall. ISBN: 0-13-825001-4.

WebKit (2017a). *Graph Coloring Register Allocator in WebKit*. URL: https://github.com/WebKit/webkit/blob/5277f6fb92b0/Source/JavaScriptCore/b3/air/AirAllocateRegistersByGraphColoring.h.

WebKit (2017b). *Linear Scan Register Allcoator in WebKit*. URL: https://github.com/WebKit/webkit/blob/5277f6fb92b0/Source/JavaScriptCore/b3/air/AirAllocateRegistersAndStackByLinearScan.h.

Wimmer, Christian (2007). *C1visualizer*. URL: https://java.net/projects/c1visualizer (visited on 01/06/2015).

Wimmer, Christian and Michael Franz (2010). "Linear Scan Register Allocation on SSA Form". In: *CGO'10*. ACM. DOI: 10.1145/1772954.1772979.

Wimmer, Christian, Vojin Jovanovic, Erik Eckstein, and Thomas Würthinger (2017). "One Compiler: Deoptimization to Optimized Code". In: *CC 2017*. ACM. DOI: 10.1145/3033019.3033025.

Wimmer, Christian and Hanspeter Mössenböck (2005). "Optimized Interval Splitting in a Linear Scan Register Allocator". In: *VEE'05*. ACM. DOI: 10.1145/1064979.1064998.

Würthinger, Thomas, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer (2017). "Practical Partial Evaluation for High-performance Dynamic Language Runtimes". In: *PLDI 2017*. ACM. DOI: 10.1145/3062341.3062381.

Würthinger, Thomas, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer (2012). "Self-optimizing AST interpreters". In: *DLS'12*. ACM. DOI: 10.1145/2384577.2384587.

Yang, Byung-Sun, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Y.C. Chung, Suhyun Kim, K. Ebcioglu, and E. Altman (1999). "LaTTe: a Java VM just-in-time compiler with fast and efficient register allocation". In: *PACT'99*. DOI: 10.1109/pact.1999.807503.

Yannakakis, Mihalis and Fanica Gavril (1987). "The Maximum K-colorable Subgraph Problem for Chordal Graphs". In: *Inf. Process. Lett.* DOI: 10.1016/0020-0190(87)90107-4.

# Josef Eisl

Compilers and Virtual Machines

**CONTACT**

zapster@zapster.cc (mailto:zapster@zapster.cc)

zapster.cc (https://zapster.cc)

linkedin.com/in/zapster (https://linkedin.com/in/zapster)

github.com/zapster (https://github.com/zapster)

@zapstercc (https://twitter.com/zapstercc)

**LANGUAGES**

German (Native) , English (Professional)

**INTERESTS**

Crafting , Traveling , Cycling , Bass and Trombone , Cooking

## CAREER PROFILE

Hi, I am Josef. I am doing research at Oracle Labs (https://labs.oracle.com/pls/apex/f?p=labs:bio:0:2667). My main interests include programming languages, compilers, register allocation, and virtual machines. Since 2014, I am working on GraalVM (https://www.graalvm.org/), a high-performance polyglot virtual machine. At Oracle Labs, my current focus is on Sulong (https://github.com/graalvm/sulong), the LLVM bitcode interpreter of GraalVM.

I am about to finish my PhD at the Johannes Kepler University Linz (http://jku.at), Austria. My research topic is on register allocation for just-in-time compilers. More precisely, I proposed a novel approach called *Trace Register Allocation* (#publications). The allocator was implemented in the Graal compiler (https://github.com/oracle/graal/blob/master/compiler) within the GraalVM project. My research was carried out as part of a collaboration between the Institute for System Software (http://ssw.jku.at) at the Johannes Kepler University and Oracle Labs.

## EXPERIENCES

### Senior Researcher — 2018–Present

Oracle Labs, Austria

I am a researcher at Oracle Labs (https://labs.oracle.com/) working on GraalVM (https://www.graalvm.org/). My current focus is on Sulong (https://github.com/graalvm/sulong), the LLVM bitcode interpreter.

### University and Research Assistant — 2014–2018

Johannes Kepler University Linz, Austria

Until 2018 I was a university and research assistant at the Institute for System Software (http://ssw.jku.at) of the Johannes Kepler University Linz (https://www.jku.at).

As part of my university assistant duties, I helt the practical classes in *compiler construction* and *data structures*. In addition, I supervised *term projects*, *Bachelor's* and *Master's* theses.

My research work was done in course of a collaboration with Oracle Labs (http://ssw.jku.at/Research/Projects/JVM/). The collaboration covers around 12 PhD, Master and Bachelor students who work in close ties with Oracle Labs employees in Linz and around the globe. The research focus is on compilers and virtual machines, more specifically the GraalVM (https://www.graalvm.org). See the Projects (#projects) section for more details.

### Student Teaching Assistant — 2008–2012

TU Wien, Austria

During my Bachelor's and Master's studies at the TU Wien (https://tuwien.ac.at), I worked as a student teaching assistant (tutor).

At the Algorithms and Data Structures Group, I was a tutor for the course *algorithms and data structures*. The duties include giving exercise lectures and organizing the programming assignments.

For the Compilers and Languages Group, I offered tutorial lessons to support students solving the *compiler construction* programming exercise.

Technical Support (part-time) 2007–2009
Sony DADC, Salzburg/Austria

At the beginning of my studies, I worked remotely as a part-time technical support employee for Sony DADC (https://www.sonydadc.com/) in Salzburg, Austria. The employment followed three years of summer internship as outlined below.

Technical Support (internship) Summer 2004, 2005, 2006
Sony DADC, Salzburg/Austria

During my senior classes at school, I repeatedly worked as a summer intern at Sony DADC (https://www.sonydadc.com/). I mainly worked on technical end-user support for copy control solutions. Other tasks included assembling of optical disc encryption control hardware as well as hardware compatibility testing.

Technical Support (internship) Summer 2002
SKIDATA, Salzburg/Austria

During the internship at SKIDATA (https://www.skidata.com), I serviced components of parking management solutions, including disassembling, repairing and reassembling of mechanical and electronic devices.

## EDUCATION

PhD in Computer Science 2014–2018 (expected)
Johannes Kepler University Linz, Austria

I am about to finish my PhD in computer science at the Johannes Kepler University Linz (https://www.jku.at). My research focus is on improving compilation in virtual machines. More precisely, most of my work is about *Trace Register Allocation*, a novel register allocation approach for just-in-time compilation. The idea was awarded with the first place (https://src.acm.org/winners/2018) in the graduate category of the ACM Student Research Competition (https://src.acm.org/) at the International Symposium on Code Generation and Optimization (CGO) 2018 (http://cgo.org/cgo2018/). The thesis is supervised by Prof. Hanspeter Mössenböck (http://ssw.jku.at/General/Staff/HM/).

MSc in Computer Science 2010–2014
TU Wien, Austria

I received a Master's degree in *Computational Intelligence* from the TU Wien (https://tuwien.ac.at) in January 2014. The core fields of the program were *algorithms and complexity*, *knowledge representation and artificial intelligence*, *logic, mathematics and theoretical computer science*, and *programming languages and verification*. My personal focus was on the last topic, specifically on compilation and virtual machines. I wrote my Master's thesis about an *"Optimization Framework for the CACAO VM (#publications)"*, a virtual machine for Java Bytecode. The thesis was supervised by Prof. Andreas Krall (http://www.complang.tuwien.ac.at/andi/).

BSc in Computer Science 2006–2010
TU Wien, Austria

In 2010 I got my Bachelor's degree in *Computer Engineering* from the TU Wien (https://tuwien.ac.at). In addition to the fundamental topics in computer science, the program focused on computer architecture, embedded systems engineering as well as physics and compiler construction. I wrote a Bachelor's thesis discussing *"RFID-based Event Tracking"*.

Matura (A-level) 1999–2005
HTBLA Salzburg, Austria

I attended the HTBLA Salzburg (http://www.htl-salzburg.ac.at) (secondary technical collage) at the department of *Electronics and Computer Engineering* with a focus on electronics, embedded systems and programming.

## PROJECTS

GraalVM (https://www.graalvm.org) — Just-in-time compiler and polyglot runtime. 2014–Present
GraalVM is an Oracle Labs (https://labs.oracle.com) project developing a new just-in-time compiler and polyglot runtime for the Java Virtual Machine. The project, led by Thomas Würthinger (https://labs.oracle.com/pls/apex/f?p=labs:bio:0:137), mainly consists of the Graal (https://github.com/oracle/graal/blob/master/compiler) just-in-time compiler, the Truffle (https://github.com/oracle/graal/blob/master/truffle) language implementation framework and Substrate VM (https://github.com/oracle/graal/blob/master/substratevm), an ahead-of-time compilation system. Most parts of the system are open source and available on GitHub (https://github.com/oracle/graal).

I am currently working on Sulong (https://github.com/graalvm/sulong), the LLVM bitcode interpreter.

During my PhD, I worked on the Graal compiler where my key area coincided with my research topic, namely *register allocation*. I implemented the novel *Trace Register Allocation* (#publications) approach on top of Graal where it has proven production quality as it is tested and benchmarked as part of Graal continuous integration pipeline since 2015. All parts of the implementation are in Graal and publicly available (https://github.com/oracle/graal).

I also participate in the development of GraalVMs build system *mx* (https://github.com/graalvm/mx), as well as the benchmarking and continuous integration systems, which are used throughout the project.

During my involvement, Graal has grown from a research project into a production quality system. Many core artifacts of the project are now part of mainstream Java, such as Java-level Compiler Interface (JDK9) (http://openjdk.java.net/jeps/243), the Graal Compiler (JDK10) (http://openjdk.java.net/jeps/317), and more are about to come (http://openjdk.java.net/projects/metropolis/).

CACAO VM (http://www.cacaojvm.org) — JIT-only Java virtual machine. 2012–2014

CACAO was among the first Java virtual machines that followed a compile-only approach. That means all code is compiled to native code prior execution. Therefore, CACAO features a *baseline compiler* for fast compilation.

In course of my Master's thesis, I implemented a second stage compilation framework, to improve the performance of the VM by adaptively compiling important methods with a higher degree of optimizations, compared to the baseline compiler. The work laid the foundation for further improvements (https://doi.org/10.1145/3178372.3179501) of CACAO.

While working on the project, I established a continuous integration infrastructure, organized the transition from a legacy code hosting model to Bitbucket (https://bitbucket.org/cacaovm/cacao-staging), and implemented code quality measures such as coverage reports, unit testing and source code documentation.

JorthVM (https://github.com/JorthVM/JorthVM) — A (minimal) Java virtual machine written in Forth. 2011–2012

JorthVM was implemented as a term project for the course *stack-based languages*, together with two colleagues. The motivation was to map the execution of the stack-based Java Bytecodes to Forth (https://www.gnu.org/software/gforth/), a stack-based programming language. Although JorthVM was at the end only able to execute the most basic snippets of Bytecodes, we learned a lot about JVM internals and the Classfile formats, as well as about untyped stack-based languages. The project sparked my interest in virtual machine design and implementation.

## SKILLS & PROFICIENCY

Since Graal (https://github.com/oracle/graal) is written in Java, this was my main programming language over the last years, although writing a compiler *for Java in Java* is not the most idiomatic use of the language. The build tool *mx* (https://github.com/graalvm/mx) and a lot of the benchmarking infrastructure is written in Python, so this is a close second.

In the course of publishing my research results, I learned to master *R* for data preparation and LaTeX as a scientific typesetting system.

When I was working on CACAO VM (http://www.cacaojvm.org), I gained profound experience with C and C++. However, the C++ language underwent drastic improvements since then so my know-how might be a bit dated.

I am a happy GNU/Linux user for 15+ years.

Compiler Construction, Java, Linux & Unix, Bash, LaTeX, Python, R & Tidyverse, C & C++, HTML & CSS, JavaScript

## ADDITIONAL INFORMATION

I am a member of a brass band where I play the trombone. In the past, I played bass in several bands. I was a mentor at JugendHackt.at (https://jugendhackt.org/), a *youth hackathon* for 12-18 year olds. In 2006, I completed my mandatory military service. I hold a driving license for the categories A, B, and C.

# Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

# Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

This printed thesis is identical with the electronic version submitted.

Linz, October 16, 2018

Josef Eisl