Trace Register Allocation Policies

Compile-time vs. Performance Trade-offs

Josef Eisl
Institute for System Software
Johannes Kepler University
Linz, Austria
josef.eisl@jku.at

Stefan Marr
Institute for System Software
Johannes Kepler University
Linz, Austria
stefan.marr@jku.at

Thomas Würthinger
Oracle Labs
Zürich, Switzerland
thomas.wuerthinger@oracle.com

Hanspeter Mössenböck
Institute for System Software
Johannes Kepler University
Linz, Austria
hanspeter.moessenboeck@jku.at

ABSTRACT

Register allocation has to be done by every compiler that targets a register machine, regardless of whether it aims for fast compilation or optimal code quality. State-of-the-art dynamic compilers often use global register allocation approaches such as linear scan. Recent results suggest that non-global trace-based register allocation approaches can compete with global approaches in terms of allocation quality. Instead of processing the whole compilation unit at once, a trace-based register allocator divides the problem into linear code segments, called traces.

In this work, we present a register allocation framework that can exploit the additional flexibility of traces to select different allocation strategies based on the characteristics of a trace. This allows fine-grained control over the compile time vs. peak performance trade-off.

Our framework features three allocation strategies, a linear-scan-based approach that achieves good code quality, a single-pass bottom-up strategy that aims for short allocation times, and an allocator for trivial traces. We present 6 allocation policies to decide which strategy to use for a given trace. The evaluation shows that this approach can reduce allocation time by 3–43% at a peak performance penalty of about 0–9% on average.

For systems that do not mainly focus on peak performance, our approach allows adjusting the time spent for register allocation, and therefore the overall compilation timer, finding the optimal balance between compile time and peak performance according to an application's requirements.

CCS CONCEPTS

•Software and its engineering →Just-in-time compilers;

KEYWORDS

Trace Register Allocation, Register Allocation, Trace Compilation, Linear Scan, Just-in-Time Compilation, Virtual Machines, Compile Time vs. Performance Trade-off

ACM Reference format:
Josef Eisl, Stefan Marr, Thomas Würthinger, and Hanspeter Mössenböck. 2017. Trace Register Allocation Policies. In *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems, Barcelona, Spain, June 2017 (LCTES 2017),* 10 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Register allocation is a mandatory task in compilers that produce code for register machines, which is the predominant type of architectures found in computers today. Its task is to map an arbitrary number of *variables* to a limited set of physical *registers* of the processor. Many sub-problems of register allocation are NP complete in general, for instance *spill free register allocation* (Chaitin et al., 1981), *minimizing spill costs* (Farach and Liberatore, 1998), or *register coalescing* (Bouchez et al., 2007). Therefore, register allocation needs to find a trade-off between the time spent for finding a solution and the resulting code quality. One of these trade-offs is whether to perform register allocation *locally*, i.e. on the scope of a basic block, or *globally* by looking at the whole compilation unit, i.e., a method. The advantage of local approaches is that they are simple since they do not need to handle control-flow. However, optimization potential is limited by the narrow scope. Global algorithms, on the other hand, offer more opportunities for improving code-quality. However, due to the problem size, compile time easily becomes a bottleneck. In modern JIT compilers, compile-time trade-offs become especially important, because aggressive inlining leads to rather large compilation units, which is a challenge for global approaches.

Trace-based register allocation, proposed by Eisl et al. (2016), solves the problem with an approach that is neither global nor local. Instead of processing a whole method at once, the basic blocks of the control-flow graph are partitioned into traces, i.e., linear sub-graphs of sequentially executed blocks. For each trace, register allocation is performed without interaction with other parts of the compilation unit. This simplifies the problem of register allocation since control-flow can be ignored.

Register allocation of traces can be done independently. Therefore, it allows the use of different allocation algorithms for different traces within one compilation unit. This enables control over the trade-off between compile time and code quality on a very fine-grained level. It allows fine-tuning JIT compilation and optimizing application performance, which is essential for systems where resources are constrained and peak performance is not the sole goal. In this paper, we evaluate the impact of heuristics to decide which algorithm to use on a per-trace basis.

Eisl et al. already applied two different allocation approaches, a *simplified linear scan* algorithm for general traces, and a special purpose allocator for *trivial traces*, i.e., traces that consist of a single, empty basic block. We propose a novel third algorithm, called *bottom-up allocator*, which is 43% faster than the trace-based linear scan strategy, with a peak-performance penalty of 9% on average.

The proposed framework is implemented in the Graal compiler (Duboscq, Würthinger, and Mössenböck, 2014; Simon et al., 2015), an optimizing compiler for the Java HotSpot VM.[1]

The contributions of this paper are:

- A framework for using different register allocation strategies within a compilation unit, based on the structure of a trace. This enables us to make fine-grained trade-off decisions between compile time and peak performance.
- A novel bottom-up register allocation strategy for traces, which only requires a single pass backwards through the instructions of the trace.
- Six different policies for selecting allocation strategies based on the properties of a trace. Each heuristic exhibits different compile-time vs. peak-performance behavior.
- A thorough compile time and peak performance evaluation of the bottom-up allocator and the policies using the DaCapo and the Scala-DaCapo benchmark suites.

2 BACKGROUND

This work is based on the trace-based register allocation approach proposed by Eisl et al. (2016), which is publicly available as part of the GraalVM.[2] This section gives a brief overview of the GraalVM and details trace-based register allocation.

2.1 GraalVM

The GraalVM is a Java virtual machine based on the HotSpot VM. The HotSpot VM comes with an interpreter and two just-in-time compilers, the *client compiler* (Kotzmann et al., 2008) and the *server compiler* (Paleczny et al., 2001). The goal of the client compiler is to provide fast compilation speed, whereas the server compiler aims at good code quality at the cost of a higher compilation time.

In the GraalVM, the server compiler is replaced by the Graal compiler as the second-tier compiler. This is done using the JVM Compiler Interface,[3] which will be part of the upcoming Java 9 release.

[1]http://www.oracle.com/technetwork/articles/javase/index-jsp-136373.html
[2]https://github.com/graalvm/graal-core
[3]JEP 243: Java-Level JVM Compiler Interface; http://openjdk.java.net/jeps/243

The Graal compiler is itself written in Java, which eliminates the need of recompiling the whole virtual machine for compiler development. 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 compiler uses two different intermediate representations. In the *frontend* Graal performs optimizations such as inlining, dead code elimination, conditional elimination, partial escape analysis (Stadler, Würthinger, et al., 2014), and loop unrolling (Stadler, Duboscq, et al., 2013) to name just a few. It uses a high-level representation (HIR), which is graph-based (Duboscq, Würthinger, Stadler, et al., 2013) and in static single assignment (SSA) form (Cytron et al., 1991). Although Java bytecode can describe irreducible programs (Aho et al., 2006), Graal only handles reducible control-flow. This assumption simplifies all control-flow-sensitive phases. Since Java programs are always reducible this restriction is not an issue in practice.

After applying all optimizations, the graph-based representation is converted into a low-level intermediate representation (LIR) before entering the *backend*. In the beginning, the LIR still adheres to the SSA form. For every variable there is only one definition which dominates all its usages. There are ϕ-functions to handle control-flow merges. This simplifies liveness analysis. The backend's main responsibility is register allocation and code emission. The register allocator also destructs the SSA form. The LIR consists of a control-flow graph with basic blocks. *Critical edges* are split, so that every edge is either the only edge leaving its source or the only edge entering its target block. This property is crucial for data-flow resolution.

A block contains a list of LIR instruction, which are close to the actual machine operations. Nevertheless, the backend phase are implemented in a machine-independent manner.

For *fixed register constraints*, e.g. 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 alive across a basic block boundary so these requirements can be handled locally.

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 use a memory operand, e.g. a stack slot. The register allocator is free to assign a stack slot to the latter kind in order to reduce the register pressure.

2.2 Trace-based Register Allocation

Instead of solving the register allocation problem globally for the whole compilation unit at once, the idea of trace-based register allocation is to divide the problem into smaller pieces, so-called traces, which are simpler to allocate due to their structural properties. The sub-solutions are then combined to get a valid global solution.

Eisl et al. use the term *trace* as it was used in trace scheduling papers, e.g. by Ellis, 1985 or Lowney et al., 1993, which operated on the same structure. A trace is a linear list of sequentially executed basic blocks. For programs in SSA form there are no lifetime holes when restricted to traces. This simplifies the implementation of a register allocator (Eisl et al., 2016).

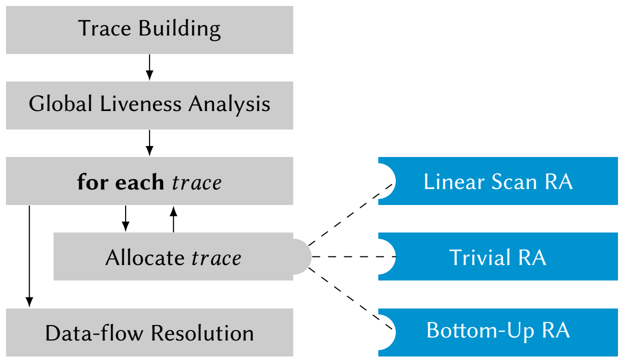


Figure 1: Trace Register Allocation Overview

The remainder of this section gives an overview of the main components of the trace register allocation approach as well as on the allocation strategies that are employed.

2.2.1 Overview. Figure 1 shows the components of the trace register allocation framework. We only cover them briefly. A detailed discussion is provided by Eisl et al. (2016).

Trace Building. The trace building algorithm takes the basic blocks of a control-flow graph as an input and returns a set of traces. Traces are non-empty and non-overlapping. Every basic block is contained in exactly one trace. For our experiments we use the *unidirectional trace building* algorithm described by Eisl et al. (2016). Figure 2 illustrates the trace-building process.

Global Liveness Analysis. To decouple the liveness of variables at trace boundaries, a global liveness analysis is required. For every inter-trace edge a $live_{out}$ and $live_{in}$ set is computed. The analysis is done in a single iteration over the blocks in reverse post order, similar to the liveness analysis described by Wimmer and Franz (2010) for SSA-based linear scan register allocation.

Allocate Traces. For each trace our algorithm selects an *allocation strategy*. The following sections detail the three strategies that are implemented in our system. Section 4 describes how we select a strategy for a trace. Note that trace can be process in arbitrary order, potentially even in parallel. However, traces that are allocated later can exploit information about already processed traces for hinting the algorithm towards a favorable solution to reduce the data-flow resolution fix-up code. Therefore, traces are ordered with respect to their *importance*. Note that this is optional and is only done to improve the resulting code.

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. This is similar to the resolution pass in linear scan allocators with interval-splitting (Traub et al., 1998; Wimmer and Mössenböck, 2005).

The remainder of this section discusses the trace-based linear scan and the trivial trace allocator proposed by Eisl et al. (2016). The bottom-up strategy is part of our contributions and is detailed in Section 3.

2.2.2 Trace-based Linear Scan. The trace-based linear scan algorithm is an adaption of the global approach by Wimmer and Franz (2010) to the properties of a trace. The main difference is that there is no nee to maintain a list of live ranges for each interval, since there are no lifetime holes in trace intervals. A *from* and *to* position is sufficient to describe an interval.

First, the algorithm creates the lifetime interval in a backward pass over the instructions of the trace. Following the linear scan principles, these intervals are then visited in order of their start position. Note that due to spilling optimizations the actual location of a variable is not yet known during this iteration (Wimmer and Mössenböck, 2005). Therefore the algorithm performs another pass over the instructions to replace the variables with the actual locations. Eisl et al. (2016) showed that the trace-based linear scan algorithm is capable of producing code that achieves peak performance which is close to the on of the global linear scan approach.

2.2.3 Trivial Trace Allocator. The trivial trace allocator is a special-purpose allocator for *trivial traces* which have a specific structure. They consist of a single basic block which does only contain a *jump* instruction. These blocks are introduced by splitting *critical edges*, and are common. For the DaCapo benchmark suite about 40% of the traces are trivial (Eisl et al., 2016). A trivial trace can be allocated by mapping the variable location at the beginning of the trace to the locations at the end of the trace.

3 BOTTOM-UP ALLOCATOR

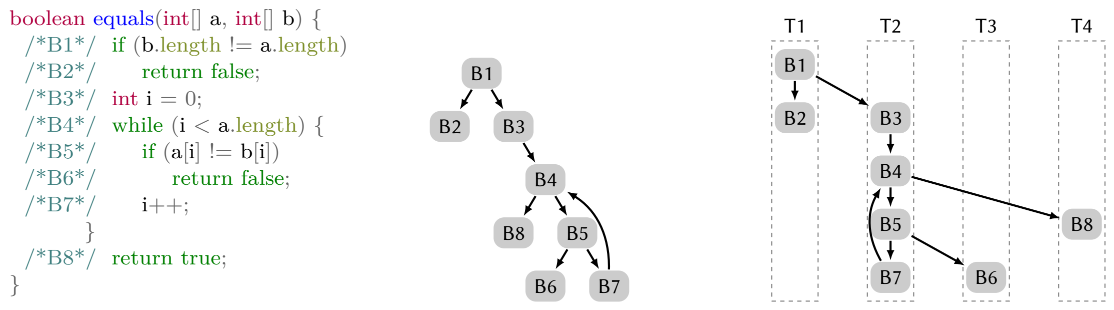
The bottom-up allocator is a novel general-purpose allocation strategy that aims at fast allocation times. It requires only a single combined backward pass over the instructions to compute the liveness requirements, select a register if required, and replace variables by the assigned location.

3.1 Tracking Liveness Information

The 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 content of every register. The entry for a register points to a *variable* if the variable is currently stored in this register. The entry can also point to the 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 whether a register is used in the current instruction. The memory requirement is therefore linear in the number of registers and the number of variables. Only the second map depends on the compilation unit while the first one is fixed for a given architecture.

3.2 Register Allocation

Register allocation is done in a single backward pass over the instruction of a trace. If the last block of the trace has a successor that has already been allocated, we use the allocation information to initialize the variable location and register content maps, to reduce data-flow mismatch.



```
boolean equals(int[] a, int[] b) {
  /*B1*/  if (b.length != a.length)
  /*B2*/      return false;
  /*B3*/  int i = 0;
  /*B4*/  while (i < a.length) {
  /*B5*/      if (a[i] != b[i])
  /*B6*/          return false;
  /*B7*/      i++;
          }
  /*B8*/  return true;
}
```

(a) Java Source **(b) Control-flow Graph** **(c) After Trace-building**

Figure 2: Trace-based Register Allocation

When visiting an instruction, we first process fixed register usages to mark them as *used* in the register content map. Next, we iterate the variable operands of the instruction. For variables that are *defined* by the current instruction, we already have a location since the algorithm iterates the instructions in reverse order. We replace the variable with the corresponding location in the variable location map. If the location happens to be a register, we mark it as free by setting the entry in the register content map to *empty*. For variables that are *read* by the current instruction, we query the variable location map the current location. There are three cases to cover.

- The variable might already be in a register. In this case we only need to replace the occurrence of the variable in the instruction with the register and are done.
- If the location of the variable is not yet defined, i.e., it is the first usage of the variable, we need to find a free register. To do so, we iterate the list of registers and look up their register content entry. If we find a register that is currently unused, i.e., its entry is empty, we can assign it to the current variable.
- If the variable is currently stored on the stack, but the instruction cannot directly use memory operands, we also need to find a register. In addition to that, we insert a *move* from the register to the stack slot where the variable is currently stored.

If all registers are occupied, we need to spill a variable. If the current operand can directly address memory, we assign it to a stack slot. Otherwise we search the available registers for one that can be spilled. We skip registers that are used in the current instruction as well as those with a fixed register constraint. The variable that was previously contained in the register is now stored in a stack slot. We insert a *move* from the stack to the selected register after the current instruction to fix the data flow.

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

3.3 Example

Figure 3 depicts bottom-up allocation of a simple trace with two blocks, B1 and B2. For readability, we omitted the details of the instructions and only show the operand mode *use*, def and use_{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 chronological order. On the right-hand side of Figure 3, 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 B2. The successor has already been allocated so the algorithm can match the incoming variable location $live_{in}(reg_0)$ of block B0 with the outgoing variable locations $live_{out}(a)$ in B2. This initializes the variable location entry of a to reg_0 and the register content of reg_0 to a. In addition to that, 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 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 we insert a move from the stack slot st_a to reg_0 right 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 register are occupied, we need to select one for spilling. We cannot use 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 used to contain 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 form 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

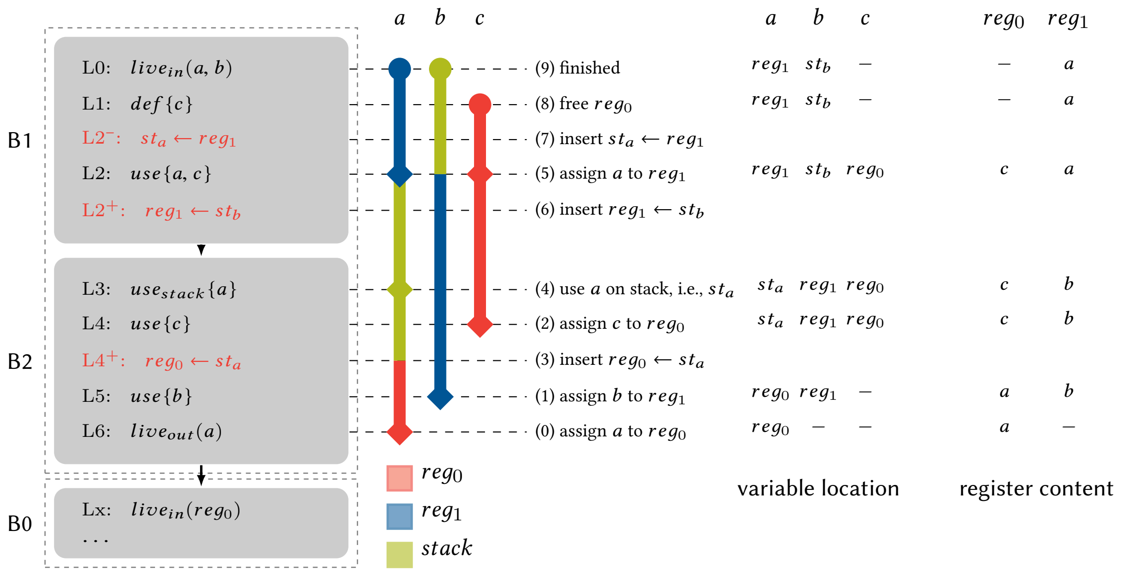


Figure 3: Bottom-Up Allocation Example

and b. Operands of the instruction are replaced with the current location of the variables.

4 TRACE REGISTER ALLOCATION POLICIES

The main goal of this work is to exploit the advantage of trace-based register allocation, namely that we can decide per trace, whether to use the linear scan, the bottom-up, or the trivial trace strategy for allocation. This is a major difference to other approaches that are restricted to a per-method choice.

First, we identify *properties* of traces. Based on these properties, we describe 6 different decision methods to select either the linear scan, the bottom-up, or the trivial allocator. We call these conditions *allocation policies*. The list of properties and policies is non-exhaustive. We will discuss alternatives in the Conclusion.

4.1 Properties

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

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 often this block is executed per invocation of the compilation unit. A value of 0.5 means that the block is executed every second time the compiled code is entered. For blocks inside of loops this value can be above 1. For example a block with a frequency of 10 is executed in a loop with an estimated iteration count of 10. Note that these number are relative to the invocation. Therefore, the frequency of the method entry block is always 1. We cannot infer absolute execution counts from these numbers. The block frequency is 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 level of the loop tree a block is located. 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.

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

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

We also consider the *trace building order*, denoted by $id(trace)$. The trace building algorithm identifies important traces first (Eisl et al., 2016). That means a trace with a lower number is in general more performance critical than one with a higher number.

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 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.

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*, or *count*.

4.2 Policies

Based on the identified properties, we developed a set of 6 *allocation policies*. 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 therefore only need 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 register allocation.

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 (i.e., the compilation unit).

TraceLSRA. This policy uses the linear scan strategy for all traces that are not *trivial.* The configuration is equivalent to the one evaluated by Eisl et al. (2016).

BottomUp. The BottomUp policy always uses the bottom-up strategy for non-trivial traces. Due to implementation reasons there is one exception to this rule, namely traces with edges to *compiled exception handlers.* These edges require a slightly different handling. 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 entries for the BottomUp policy in Figure 5 show that the fraction of *linear scan compiled* traces is indeed marginal.

MixedPol. This 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 half of the traces created by the trace builder is allocated with linear scan (or the trivial allocator).

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

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

where $HasLoop(blocks)$ is defined as:

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

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 current compilation unit does not contain a loop at all. The rationale behind this is that the virtual machine only compiles methods which either exceed a certain invocation or loop-backedge threshold. If a method without a loop is queued for compilation, the runtime did so due to the invocation count only. This means it was called often enough to be considered important.

FreqPol. This policy considers a trace important if the maximum execution frequency of all blocks in the trace is greater than a fraction p of the maximum frequency of all blocks in the compilation unit.

$$\max_{b_1 \in trace} freq(b_1) > \max_{b_2 \in method} freq(b_2) \times p$$

Only traces with high frequency blocks are allocated with the linear scan strategy since these traces are most critical for performance.

LoopFreqPol. This policy combines the LoopPol policy with the FreqPol policy. Instead of using linear scan for all compilation units without loops, we apply the FreqPol condition.

$$HasLoop(trace) \lor \big(\neg HasLoop(method) \land \text{FreqPol}(trace)\big)$$

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

BudgetPol. The BudgetPol 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 a trace. The budget is a fraction of the sum of the frequencies of all blocks in the compilation unit.

5 EVALUATION

In this evaluation we present experimental results to substantiate our claim that selective register allocation based on traces is an appropriate approach for controlling the compile time vs. peak performance trade-off on a fine-grained level.

We use the implementation of the trace-based linear scan strategy in Graal by Eisl et al. (2016). We added the bottom-up allocation strategy, the policy selection logic, and the policies described in the previous section.

The source code of our implementation is available on Github.[4] Our experiments were performed using revision 7d0e15f0e169.

5.1 GraalVM configuration

The *default* configuration of the Graal compiler is tuned for *peak performance.* The majority of the compile time is spent in the frontend on code optimizations. While some optimizations increase register pressure (e.g., partial escape analysis; Stadler, Würthinger, et al., 2014), Graal in general aims to simplify the code as much as possible. Since we want to study the compile-time behavior of the backend, more precisely of the register allocator, aggressive optimizations in the frontend are not helping. They can even distort the result since they might reduce the pressure on register allocation. We therefore focus on the *economy* configuration of Graal[5] which aims at fast compilation time instead of peak performance. In this configuration, most optimization phases are disabled. For completeness we also show the results for the *default* configuration. However, we will not discuss these results in detail but give a general comparison between the behavior of the two configurations at the end of the section.

[4]https://github.com/zapster/graal-core/tree/tracera-policy-experiments
[5]To enable the *economy* configuration start the GraalVM with the following system property: -Dgraal.CompilerConfiguration=economy

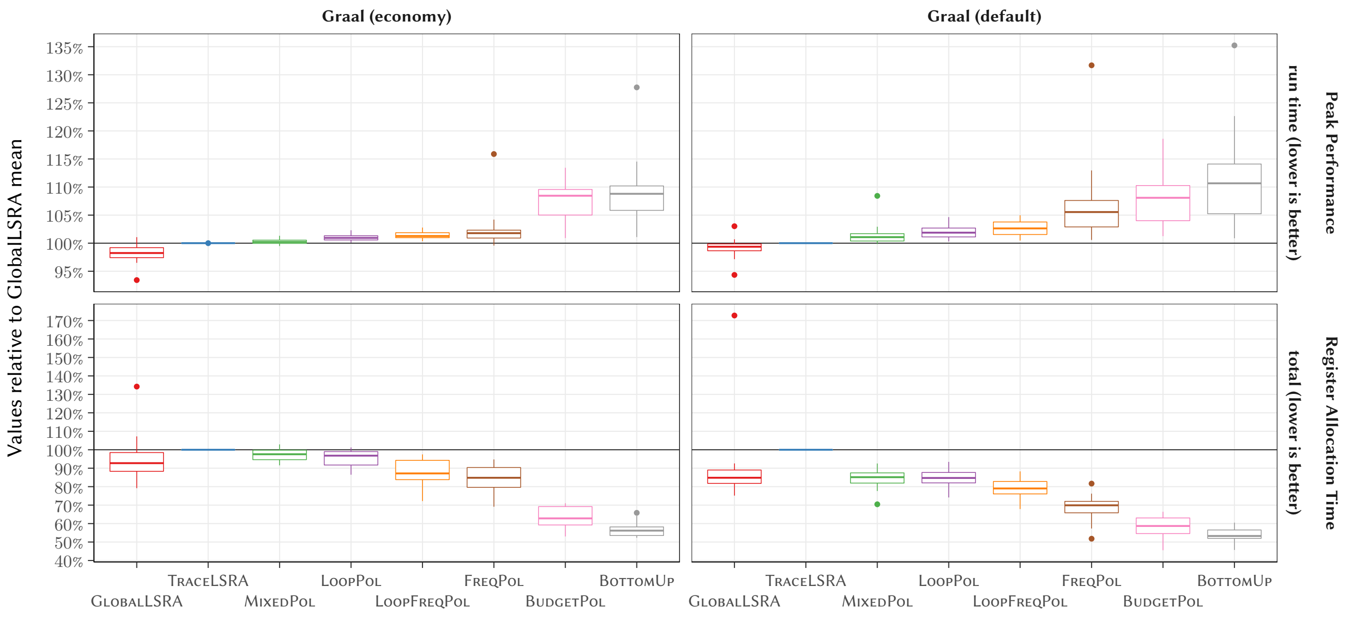


Figure 4: Peak Performance and Register Allocation Time

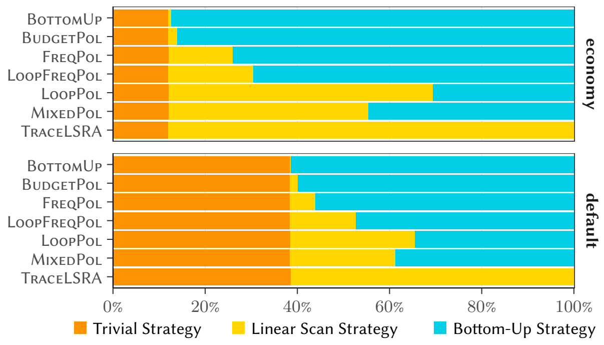


Figure 5: Distribution of the Allocation Strategy per Policy

5.2 Benchmark Suites

We evaluated our results using the DaCapo 9.12 (Blackburn et al., 2006) as well as the Scala-DaCapo (Sewe et al., 2011) benchmark suites. We excluded the eclipse, tomcat, tradebeans, and tradesoap benchmarks from DaCapo due to Java 8 compatibility issues. Together with Scala-DaCapo we have 22 different benchmarks in total. The DaCapo-style benchmarks are iteration-based, meaning that they run the same workload for a predefined number of times in order 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 8 to 180. The run time of the last iteration is the performance result of the benchmark.

5.3 Hardware Environment

We performed the experiments on a cluster of 64 identical Sun Server X3-2,[6] equipped with two Intel "Sandy Bridge" Xeon E5-2660 @ 2.20GHz with 8 cores per processor, and 256GB of DDR3-1600 memory. The machines were running an Oracle Linux Server 2.6 operating system with Linux Kernel version 2.6.32. For the experiments we disabled all frequency scaling modes (e.g. scaling governors or Intel Turbo Boost).

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 each benchmark we started a new Java VM with an initial and maximum heap size of 8GB. To improve the precision of the results we fixed the CPU and the memory of the process to a single NUMA node using the hwloc-bind utility.[7]

To minimize the effect of disk I/O we executed the benchmarks on a 10GB ram disk. For some benchmarks, for instance lusearch, luindex, h2, or batik, this is necessary to get stable results.

5.4 Evaluation Metrics

We repeated every experiment 30 times to compensate for variation factors we cannot control, such as low-level hardware differences or non-determinism of the virtual machine. For every metric we calculated the *mean* for each benchmark and every configuration. We present these means as box plots (Tukey, 1977) to give an unbiased impression of the distribution of results.

Peak Performance. The reported performance result for the DaCapo-style benchmarks is the time required for the last iteration. Ideally, in this iteration the VM does no perform any compilation. In reality,

[6]Sun Server X3-2: http://docs.oracle.com/cd/E22368_01/
[7]hwloc-bind(1) - Linux man page: https://linux.die.net/man/1/hwloc-bind

however, we cannot omit compilations completely due to the behavior of the harness and the benchmark code. The peak performance is shown in the top half of Figure 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 intermingled. 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 timing.

The meta-circular aspect of the GraalVM adds another layer of challenges to the problem. Since the compiler itself is subject to compilation, changes in the compiler influence not only the generated machine code, but also the compile time it takes to translate the compiler itself. To minimize this effect, Graal avoids self-compilation, i.e. methods in the Java packages jdk.vm.ci and org.graalvm.compiler are only compiled by the HotSpot client compiler.

The total compile time results for all Graal compilations are depicted at the bottom of Figure 4.

5.5 Analysis of the Results

The baseline for all our experiments is the trace-based linear scan configuration, denoted by TraceLSRA. To visualize all benchmarks on the same scale, we show the numbers relative to the mean of the baseline of a given benchmark. Regarding compile time, we are interested in the time spent for register allocation. In case of the trace-based register allocator we include *trace-building, global liveness analysis* as well as the actual allocation algorithm including the allocation strategy selection.

Figure 4 shows the total register allocation time relative to trace-based linear scan. We include all compilations of the benchmark run, including warm-up iterations, since they all contribute to the peak-performance result in the last iteration.

GlobalLSRA. For comparison, we also show the global linear scan results. It completes about 5% faster than the TraceLSRA policy. TraceLSRA's overhead is mostly due to extra compiler phases required for trace register allocation, i.e. the global liveness analysis and trace building. Figure 4 shows an allocation time outlier in favor of the trace-based policies. This is the result for the jython benchmark from the DaCapo suite where the global linear scan implementation exhibits a non-linear behavior.

TraceLSRA. The TraceLSRA configuration performs best with respect to peak performance. Figure 4 shows that on average the benchmark take about 3% more time to execute than with the global linear scan register allocator. The reason for this slowdown are disadvantages of the trace-based approach, especially with respect to spilling in hot loops, which were detailed previously by Eisl et al. (2016).

The TraceLSRA policy marks the upper bound in terms of register allocation time and in terms of performance for all policies.

BottomUp. The BottomUp policy, on the other hand, is the lower bound with respect to allocation time. It requires only about 43% of the time compared to TraceLSRA. In terms of peak performance, this policy is the slowest with an average performance decrease of about 9%. For the sunflow benchmark from the DaCapo suite, however, the performance penalty is 28%.

MixedPol. In our experiment we used the MixedPol policy with a parameter $p = 0.5$. From the peak-performance perspective this configuration is almost on the same level as the TraceLSRA approach. Register allocation time went down by about 3% compared to TraceLSRA. This improvement is due to the reduced number of linear scan allocations, which is depicted in Figure 5.

LoopPol. The LoopPol configuration is a sightly slower than the MixedPol when it comes to peak performance. The register allocation time is on same level although it seems to vary more. Figure 5 shows that when using the LoopPol policy more traces are compiled with linear scan than with the MixedPol configuration.

FreqPol. For the FreqPol we also used a factor of $p = 0.5$. Compared to the trace-based linear scan algorithm it is 2% slower on peak performance on average. Again, sunflow exhibits a worst-than-average behavior with a performance decrease of 16%. Allocation time, on the other hand, is 16% better than TraceLSRA.

LoopFreqPol. The LoopFreqPol policy ($p = 0.5$) combines the advantages of LoopPol, i.e. good and stable peak performance, with the fast allocation time of the FreqPol policy. With respect to peak performance it is almost on the same level as the LoopPol configuration while compile time is more than 8% better on average. Figure 5 shows that significantly less traces are allocated with linear scan. In addition, the peak-performance outlier caused by sunflow, seen in the FreqPol configuration, is no longer present. This indicates that the benchmark is very sensitive to the register allocation in loops. A behavior that has also been observed by Eisl et al. (2016).

BudgetPol. The budget-based BudgetPol policy, with parameter $p = 0.5$, uses linear scan only for a small fraction of the traces, as depicted in Figure 5. Noteworthy is its peak-performance behavior for sunflow. Although FreqPol is superior in the average case, BudgetPol performs better on this benchmark. On average the BudgetPol policy is about 2% faster regarding peak performance than the BottomUp configuration. The allocation time is about 37% better than the TraceLSRA policy.

5.6 Graal Economy vs. Default

One evident difference between the *economy* and the *default* configuration of Graal is the relative number of *trivial traces*, which is depicted in Figure 5. While in the *economy* configuration only about 10% of the traces are trivial, it is almost 40% for the *default* configuration. The higher number indicates that the control-flow is more complicated. The entries for the LoopPol policy in Figure 5 also show that less of the non-trivial traces contain loop blocks. This results in a compile-time gap between the TraceLSRA and LoopPol policy that is almost 10% bigger in the *default* configuration. This suggests that the *default* configuration offers more opportunities for saving compile time. From the peak-performance perspective both configurations behave similar.

6 RELATED WORK

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.

6.1 Dynamic and Adaptive Compilation

Modern high-level language virtual machines use dynamic compilation to produce efficient native machine code. However, for such systems, the time constraints for the compiler are very strict. For instance, the CACAO VM (Krall, 1998), performs optimizations only on a local scope. Later systems such as the Jalapeño VM (Arnold et al., 2000) or the HotSpot VM (Paleczny et al., 2001), introduce adaptive compilation to focus compilation on relevant parts of an application. They use multiple optimization stages that are invoked for performance critical parts only. 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 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.

6.2 Trace Compilation

Instead of focussing on *methods* as the unit of operation, trace compilation systems, such as Dynamo by Bala et al. (2000) or HotPathVM by Gal et al. (2006), take a different route. They *trace* the execution of the program, potentially across method boundaries, an then select such a recorded trace for compilation. This way they only compile the parts of a program that are performance critical, which narrows the scope of the compilation unit and therefore improves compile time. However, we are not aware of any trace compilation system that chooses different algorithms based on the structure of a trace to further improve compile time.

6.3 Register Allocation

The design decisions taken for a compiler depend heavily on its application domain and intended usage. This is especially relevant for register allocation since it is typically mandatory. Compilers used for *static* compilation are less restricted in terms of compile time than a *dynamic* compiler in a virtual machine, where compilation time adds to the overall program execution time. The major design decision of a register allocator in this context is whether it should work on a *global* scope, on a *local* one, or a middle ground like the trace-based approach. We showed in this paper that a trace-based approach enables fine-grained control over how and where to spent time on register allocation.

A global approach such as graph coloring (Chaitin et al., 1981; Briggs et al., 1989; George and Appel, 1996) does not provide this flexibility. Optimizations focus here on the heuristics to improve code quality. For just-in-time compilation these approaches are often too costly.

To meet the compile-time requirements for the dynamic code generation system tcc (Poletto, Engler, et al., 1997), Poletto and Sarkar (1999) introduced *linear scan* as a simple and fast method for global register allocation. They achieved peak performance that was within 10% of a graph coloring approach. Wimmer and Mössenböck (2005) improved code quality achieved with linear scan by making it more precise and moving spill code out of loops. However, this makes the algorithm computationally more expensive. By exploiting SSA properties, Wimmer and Franz (2010) where able to decrease allocation time with virtually no peak-performance regression. However, the overall approach did not change with respect to its granularity. A single algorithm is applied to all code independent of whether it is performance critical or not.

Related to our proposed bottom-up allocator is the work by Yang et al. (1999). They describe LaTTe, a compile-only Java VM that focusses on compilation speed, including a fast, non-local register allocation. Register allocation is performed on tree regions, that is a tree 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 preferences a forwards pass performs the actual register allocation. Their handling of spilled variables is similar to the approach used by our bottom-up allocator. However, we perform allocation on traces instead of trees, and require only a single backward pass.

Our work builds on top of the trace-based register allocator of Eisl et al. (2016), as detailed in Section 2. The idea of using traces as unit of operation was introduced by Fisher (1981) for instruction scheduling for Very Long Instruction Word (VLIW) architectures to exploit Instruction Level Parallelism (IPL). Freudenberger et al. (1994) studied the connection of instruction selection and register allocation on traces. However, to the best of our knowledge, none of these approaches applied different allocation algorithms within a compilation unit and provide the flexibility of our framework. Also, since their system was designed for static compilation, compile time was not a priority.

7 CONCLUSION

The trace-based register allocation approach offers the flexibility to switch between allocation algorithms within one 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.

Our framework can choose between three different register allocation strategies: a linear-scan-based algorithm, a novel bottom-up allocator and a specialized approach for trivial traces. The bottom-up allocator is 43% faster then the trace-based linear scan implementation at a performance degradation of only 9% on average.

We implemented and evaluated 6 different policies for deciding which register allocator is to be used for a specific trace. The BudgetPol policy, for instance, improves register allocation time by 37% on average compared to the trace-based linear scan approach with an average peak-performance slowdown of only 7% but a better worst-case behavior than the bottom-up-only approach. On the other hand, the LoopFreqPol policy decreases allocation time by about 13% with a performance degradation of only 1% compared to TraceLSRA. Most policies can be parameterized, which allows adjusting the trade-off between compile time and peak performance on a fine-grained level.

This paper evaluates 6 allocation policies to show the flexibility of our approach. Future work could investigate other policies that might have better performance trade-offs. One specific aspect is that most of our policies can be parameterized. While we experimented with different setting we did not evaluate the tuning potential exhaustively. Furthermore, combining existing policies can result in new useful configurations, as suggested by our evaluation of the LoopFreqPol policy. Since the search space for policies is large, we believe that utilizing auto-tuning tools, such as OpenTuner (Ansel et al., 2014), is an idea that is worth investigating.

It should also be further explored, on which properties policies should be based. We focused on trace properties that are exposed in our experimentation platform or are simple to compute. For example, while we have direct access to (bytecode) branch probabilities, we do not have access to the global execution count of a method. Therefore, evaluating such metrics is left for future work. It could also be explored whether considering the specific instructions in a trace can be exploited to select an allocation policy.

The trace-based approach in general, and our policy model in particular, is not restricted to the problem of register allocation. Other optimizations, such as instruction scheduling or instruction selection could apply the same idea to benefit from a fine-grained control over the compile-time vs. quality-of-result balance. Furthermore, the proposed policies are not specific to register allocation but can be applied to other problems in compiler design and optimization.

ACKNOWLEDGMENTS

We thank the Graal community, the Virtual Machine Research Group at Oracle Labs and the Institute for System Software at the Johannes Kepler University Linz for their support and feedback on this work. Josef Eisl is funded by a research grant from Oracle Labs. Stefan Marr is funded by a grant of the Austrian Science Fund (FWF), project number I2491-N31.

REFERENCES

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. URL: http://dragonbook.stanford.edu/.

Ansel, Jason, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe (2014). "OpenTuner: An Extensible Framework for Program Autotuning". In: PACT '14. DOI: 10.1145/2628071.2628092.

Arnold, Matthew, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney (2000). "Adaptive Optimization in the Jalapeño JVM". In: *SIGPLAN Not.* DOI: 10.1145/1988042.1988048.

Bala, Vasanth, Evelyn Duesterwald, and Sanjeev Banerjia (2000). "Dynamo: A Transparent Dynamic Optimization System". In: *SIGPLAN Not.* DOI: 10.1145/358438.349303.

Blackburn, S. M. et al. (2006). "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA'06*. ACM Press. DOI: 10.1145/1167473.1167488.

Bouchez, Florent, Alain Darte, and Fabrice Rastello (2007). "On the Complexity of Register Coalescing". In: *CGO'07*. DOI: 10.1109/cgo.2007.26.

Briggs, P., K. D. Cooper, K. Kennedy, and L. Torczon (1989). "Coloring Heuristics for Register Allocation". In: *SIGPLAN Not.* DOI: 10.1145/74818.74843.

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.

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: *ACM Trans. Program. Lang. Syst.* DOI: 10.1145/115372.115320.

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.

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.

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

Farach, Martin and Vincenzo Liberatore (1998). "On Local Register Allocation". In: *SODA'98*. Society for Industrial and Applied Mathematics. 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.

Freudenberger, Stefan M., Thomas R. Gross, and P. Geoffrey Lowney (1994). "Avoidance and Suppression of Compensation Code in a Trace Scheduling Compiler". In: *ACM Transactions on Programming Languages and systems.*

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.

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

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.

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.

Paleczny, Michael, Christopher Vick, and Cliff Click (2001). "The Java HotSpot™ Server Compiler". In: *JVM'01*. USENIX Association. URL: https://www.usenix.org/legacy/events/jvm01/full_papers/paleczny/paleczny.pdf.

Poletto, Massimiliano, Dawson R. Engler, and M. Frans Kaashoek (1997). "tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation". In: *SIGPLAN Not.* DOI: 10.1145/258916.258926.

Poletto, Massimiliano and Vivek Sarkar (1999). "Linear Scan Register Allocation". In: *TOPLAS'99*. DOI: 10.1145/330249.330250.

Sewe, Andreas, Mira Mezini, Aibek Sarimbekov, and Walter Binder (2011). "Da capo con scala". In: *OOPSLA'11*. DOI: 10.1145/2048066.2048118.

Simon, Doug, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, and Thomas Würthinger (2015). "Snippets: Taking the High Road to a Low Level". In: *TACO'15*. DOI: 10.1145/2764907.

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.

Traub, Omri, Glenn Holloway, and Michael D. Smith (1998). "Quality and Speed in Linear-scan Register Allocation". In: *SIGPLAN Not.* DOI: 10.1145/277652.277714.

Tukey, John W. (1977). *Exploratory data analysis*. Reading, Mass.

Wimmer, Christian and Michael Franz (2010). "Linear Scan Register Allocation on SSA Form". In: *CGO'10*. ACM. DOI: 10.1145/1772954.1772979.

Wimmer, Christian and Hanspeter Mössenböck (2005). "Optimized Interval Splitting in a Linear Scan Register Allocator". In: *VEE'05*. ACM. DOI: 10.1145/1064979.1064998.

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.