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 is an integral part of compilation, regardless of whether a compiler 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 (i.e., method) 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 provides us with fine-grained control over the trade-off between compile time and peak performance in a just-in-time compiler.

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*.

To demonstrate the flexibility of the framework, we select 8 allocation policies and show their impact on compile time and peak performance. This approach can reduce allocation time by 7%–43% at a peak performance penalty of about 1%–11% on average.

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

ManLang 2017, September 27–29, 2017, Prague, Czech Republic

ACM ISBN 978-1-4503-5340-3/17/09…$15.00
https://doi.org/10.1145/3132190.3132209

CCS CONCEPTS

• **Software and its engineering → Just-in-time compilers**; **Dynamic compilers**; **Virtual machines**;

KEYWORDS

Trace Register Allocation, Trace Compilation, Linear Scan, Just-in-Time Compilation, Dynamic 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 ManLang 2017, Prague, Czech Republic, September 27–29, 2017,* 13 pages.
https://doi.org/10.1145/3132190.3132209

1 INTRODUCTION

Register allocation is an integral part of 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 make a trade-off between the time spent on 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 large compilation units, which are a challenge for global register allocation 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 for each trace 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 predominant goal. In this paper, we evaluate the flexibility of our framework by applying different heuristics to decide which algorithm to use on a per-trace basis.

Eisl et al. already applied two 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. In this paper, we added the *bottom-up allocator* as a third algorithm. It is 43% faster than the trace-based linear scan strategy, with a peak-performance penalty of 11% on average. We also extended the above framework with *allocator selection strategies*, which are based on policies that exploit properties of the traces.

The new framework is implemented in the Graal compiler [Duboscq et al., 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 allocation policies to decide whether to use the *linear scan* or the *bottom-up* register allocation strategy for a trace of a compilation unit. This enables us to make fine-grained trade-off decisions between compile time and peak performance.
- An extension of the existing trace-based allocator with a fast bottom-up register allocation strategy for arbitrary traces. It requires only a single pass backwards through the instructions of the trace and is therefore significantly faster than the preexisting linear scan strategy.
- A set of 8 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 14 different configurations using the DaCapo and the Scala-DaCapo benchmark suites.

The rest of this paper is organized as follows. We based our approach on previous work on trace-based register allocation by Eisl et al., 2016. In Section 2 we review their approach and present a system overview of GraalVM, the virtual machine we used for our implementation. Section 3 introduces our bottom-up allocation strategy that was added to the existing framework to increase the fine-tuning capabilities of our selective register allocation approach. In Section 4 we describe our so-called *trace register allocation policies*, which are heuristics to decide which allocator should be used for a specific trace. We selected a set of 8 policies for empirical evaluation using different parameters. The results are outlined in Section 5. In Section 6 we discuss related work and how it compares to our contribution. We conclude the paper with a summary of our results and propose directions for future extensions.

[1] http://www.oracle.com/technetwork/articles/javase/index-jsp-136373.html

2 BACKGROUND

The work in this paper 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 is part of the upcoming Java 9 release.

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 *front end* Graal performs optimizations such as inlining, dead code elimination, conditional elimination, partial escape analysis [Stadler et al., 2014], and loop unrolling [Stadler et al., 2013] to name just a few. It uses a high-level representation (HIR), which is graph-based [Duboscq et al., 2013] and in static single assignment (SSA) form [Cytron et al., 1991; Brandis and Mössenböck, 1994]. Although Java bytecode can describe irreducible programs [Aho et al., 2006], Graal handles only 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 to a low-level intermediate representation (LIR) before entering the *back end*. 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 back end's main responsibility is register allocation and code generation. 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. Figure 1a depicts an example. Block $B1$ has two successors and $B3$ has two predecessors. Therefore the edge between those two blocks is *critical*. We insert an empty block to *split* this edge. The result is shown in Figure 1b. This property is crucial for data-flow resolution.

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

[2] https://github.com/graalvm/graal-core

[3] JEP 243: Java-Level JVM Compiler Interface; http://openjdk.java.net/jeps/243

(a) Critical Edge (b) After Splitting

Figure 1: Critical Edge Splitting

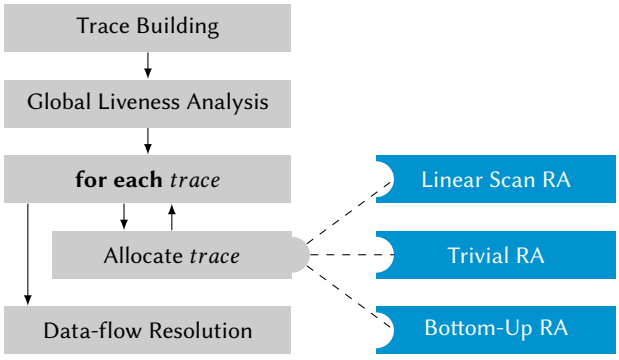


Figure 2: Trace Register Allocation Overview

For *fixed register constraints*, e.g. 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.

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 in traces. This simplifies the implementation of a register allocator [Eisl et al., 2016].

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 2 shows the components of the trace register allocation framework. We cover them only 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]. A new trace is started by selecting the block with the highest execution frequency, that is not already part of a trace. The algorithm continues with the most likely successor block that is not yet included in a trace. This procedure continues until there is no more successor that is not in a trace already. Figure 3 illustrates the trace-building process.

Global Liveness Analysis. To capture 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.

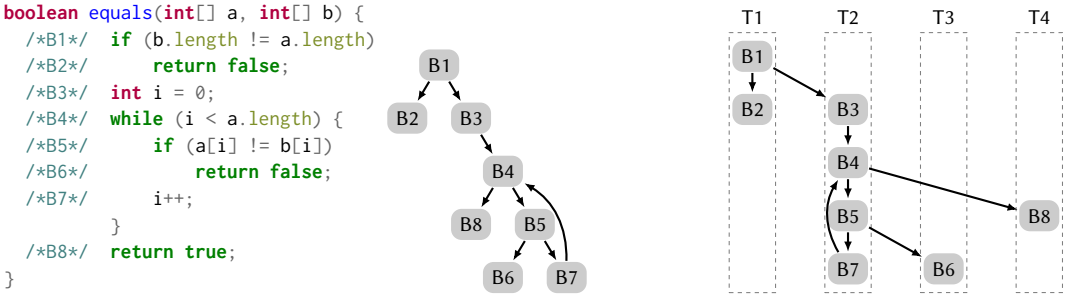
Allocate Traces. For each trace our algorithm selects an *allocation strategy*. The following sections detail the three strategies that are currently implemented in our system. Section 4 describes how we select a strategy for a trace. Note that traces can be processed in arbitrary order, potentially even in parallel. However, traces that are processed later can exploit information about already processed traces for hinting the algorithm towards a favorable solution to reduce the data-flow resolution at trace boundaries. Therefore, traces are ordered with respect to their *importance*. Note that this is optional and is done only 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]. In addition, data-flow resolution performs SSA destruction, i.e., it replaces ϕ-functions with move instructions.

The remainder of this section discusses the trace-based linear scan allocator 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 need to maintain a list of live ranges for each lifetime interval, since there are no lifetime holes in trace intervals. A *from* and *to* position are sufficient to describe an interval.

First, the algorithm creates the lifetime intervals of all variables in a backward pass over the instructions of the trace. Following the linear scan principle, these intervals are then visited in order of their start position. Note that due to possible spilling 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 comparable to that of the global linear scan approach.



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

Figure 3: Trace-based Register Allocation

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 contains only a single *jump* instruction. These blocks are introduced by splitting *critical edges*, and are quite 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 locations at the beginning of the trace to the locations at the end of the trace.

3 BOTTOM-UP ALLOCATOR

Not all traces of a method are equally important for peak performance. Eisl et al. [2016], for instance, processed traces in the order of decreasing *execution frequency* to shift spill code to less frequently executed parts of the method. We pursue a similar idea to reduce the register allocation time. The goal is to spend time only on traces that are worth it, i.e., that contribute to peak performance. The other traces still need a valid allocation, but the quality is not critical. Therefore, we aim for a fast, general purpose allocation strategy that sacrifices peak performance for allocation time.

The trace-based linear scan allocator exhibits a linear time behavior with respect to the number of instructions [Eisl et al., 2016], which is the asymptotic lower bound for the problem. However, the constant factors are relevant in practice. As outlined in the previous section, the algorithm iterates over the list of instructions three times: once for liveness analysis, once for allocating registers, and a third time for replacing variables with the assigned registers. The algorithm is guided by the set of intervals, which are maintained throughout all passes. All these components are required for improving the allocation quality, not for correctness. They are unnecessary for a fast allocation where run-time performance is not the main focus.

To address these issues, we added a new allocator (called the *bottom-up allocator*) to the existing framework. It requires only a single combined backward pass over the instructions. In this pass the allocator computes the liveness requirements, selects registers if required, and replaces variables by the assigned location.

3.1 Tracking Liveness Information

In the bottom-up allocator, liveness information is never maintained for the whole trace but is known only 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 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 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 size of the second map depends on the compilation unit. The register map's size is fixed for a given architecture.

3.2 Register Allocation

Register allocation is done in a single backward pass over the instructions of a trace. 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.

When visiting an instruction, we first process fixed register usages to mark them as *used* in the register content map. Next, we iterate over the variable operands of the instruction. For variables that are *defined* by the current instruction, we already have a location since the algorithm iterates over 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 for the current location. There are three cases to cover:

- The variable might already be in a register. In this case we need only 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 last usage of the variable, we need to find a free register. To do so, we iterate over 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 the variable is stored on the stack, but the instruction cannot directly use memory operands, we need to find a

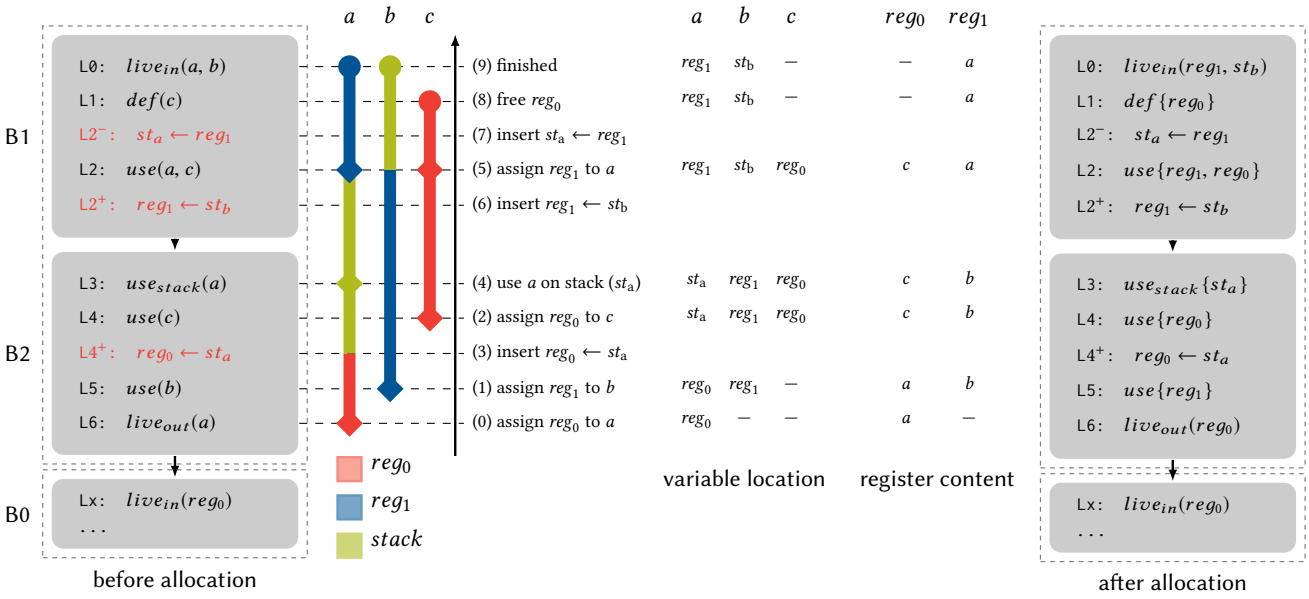
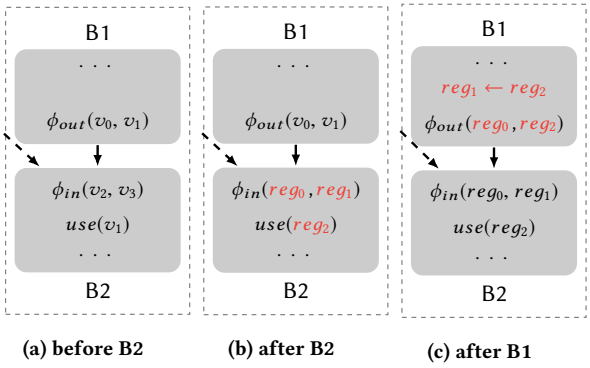


Figure 4: Bottom-Up Allocation Example



(a) before B2 **(b) after B2** **(c) after B1**

Figure 5: ϕ-resolution in the Bottom-Up Allocator

register and insert a *move* instruction to get the variable from the stack slot into the register.

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 first register that is not skipped by these constraints is the chosen for spilling. The variable that was previously contained in that register is now stored in a stack slot. Thus, we insert a *move* after the current instruction that restores the variable in the selected register from the stack to fix the data flow.

At block boundaries the allocator needs to take care of ϕ-instructions. ϕ-instructions are basically *parallel moves* from the locations in the predecessor (ϕ_{out}) to the locations in the successor (ϕ_{in}) [Hack, 2007]. At the beginning of a basic block, all variables in the ϕ_{in} set have already been assigned to a location. Due to the *single definition* property of the SSA-form we know that these variables are not live in any predecessor, i.e., they are defined at the beginning of the block. Therefore, we can directly reuse their locations for those variables in the ϕ_{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.

Figure 5 shows an example for ϕ-resolution. Figure 5a shows a trace consisting of two blocks $B1$ and $B2$. Block $B2$ is a merge that contains two ϕ variables, v_2 and v_3. In the predecessor $B1$ these variables are matched to v_0 and v_1, respectively. After allocation of $B2$ (Figure 5b) we allocated v_1 to reg_2, v_2 to reg_0 and v_3 to reg_1. Before we continue with $B1$ we need to resolve the data flow between ϕ_{out} and ϕ_{in}. Namely, we want to map v_0 to reg_0 and v_1 to reg_1. Since v_0 is not yet assigned to a location we can simple replace it with reg_0. Variable v_1, on the other hand is already stored in reg_2. To resolve this data-flow mismatch, we insert a move from reg_2 to reg_1. Figure 5c shows the result of the resolution step.

We consider only the predecessor that is part of the current trace. Since there are no critical edges there can only be one. The other predecessors are handled by the *data-flow resolution* phases afterwards.

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

3.3 Example

Figure 4 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 modes *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 processing order. On the right-hand side of Figure 4, 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 $live_{\text{in}}(reg_0)$ of block B0 with the outgoing variable locations $live_{\text{out}}(a)$ in B2. 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 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 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.

4 TRACE REGISTER ALLOCATION POLICIES

Our main goal is to demonstrate that switching the register allocation algorithm on a per-trace basis enables fine-grained compile-time vs. peak-performance trade-off control not seen in other approaches. To support our claim we present a case study of 8 decision heuristics, so-called *allocation policies*.

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. 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 enclosing method is invoked. For blocks inside of loops this value can be above 1. For example, a loop header that is entered with a probability of 1 and with a frequency of 10 indicates a loop iteration count of 10. Note that these numbers 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 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 2.2.1, we can also take the $live_{in}$ and $live_{out}$ sets into account, i.e., the variables live at the beginning and the end of the block. More live variables increase the likelihood of spilling.

Trace Properties. The properties of the blocks of 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 *triviality*, i.e., the fact that a trace consists of a single block containing just a jump instruction. It determines whether or not the algorithm can use the trivial trace allocator.

We also consider the *trace building order*, denoted by $id(trace)$. The trace building algorithm constructs important traces first [Eisl et al., 2016]. That means a trace with a lower number is generally 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 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.

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*.

4.2 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 therefore need to decide only 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 (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 slightly different handling.[4] 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 7 shows that the fraction of *linear-scan-compiled* traces is indeed marginal (~0.3% as depicted in Table A in the appendix).

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 half 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 sum of the frequencies of all blocks 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:

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

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 compiles only 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 that the method was called often enough to be considered important.

[4]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.

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.

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

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 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. For example, if $p = 0.8$, a trace is compiled with the linear scan allocator if its frequency is larger than 0.8× the frequency of the most frequent block of the method. In other words, only traces with high-frequency blocks.

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

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

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.

5 EVALUATION

The goal of this evaluation is 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 previous 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.

We used the implementation of the trace-based linear scan strategy in Graal by Eisl et al. [2016] to which 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.[5] Our experiments were performed using revision `f5cad2eda111`.

5.1 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

[5]https://github.com/zapster/graal-core/tree/tracera/policies

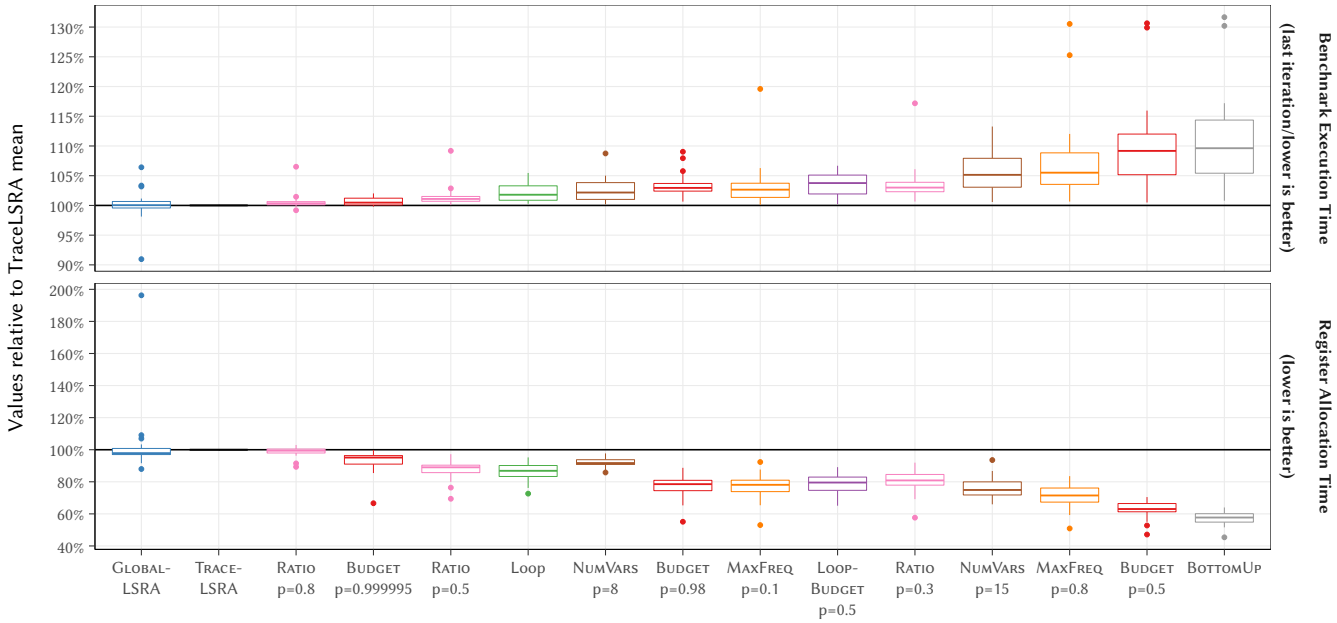


Figure 6: Peak Performance and Register Allocation Time

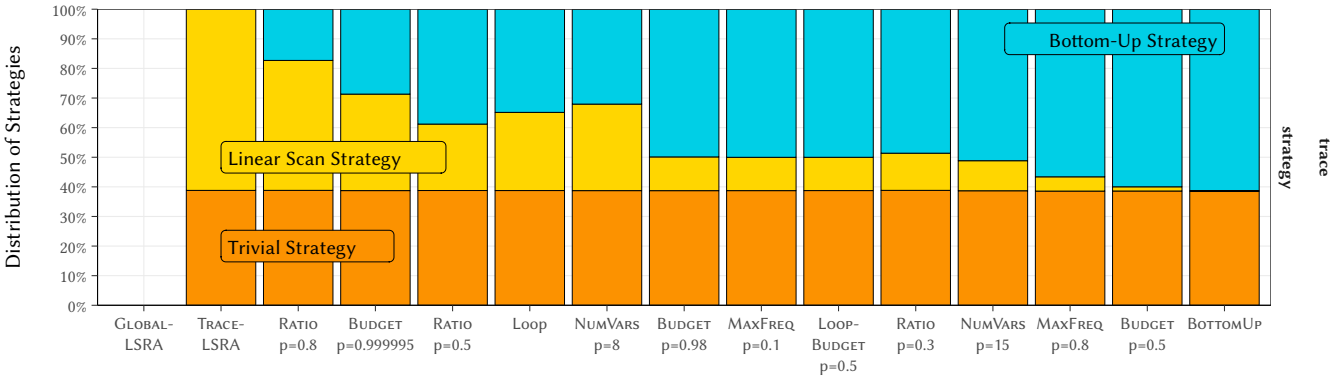


Figure 7: Distribution of the Allocation Strategy per Policy

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

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 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. The run time of the last iteration is the performance result of the benchmark.

5.2 Hardware Environment

We performed the experiments on a cluster of 64 identical Sun Server X3-2 machines,[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 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).

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.

[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

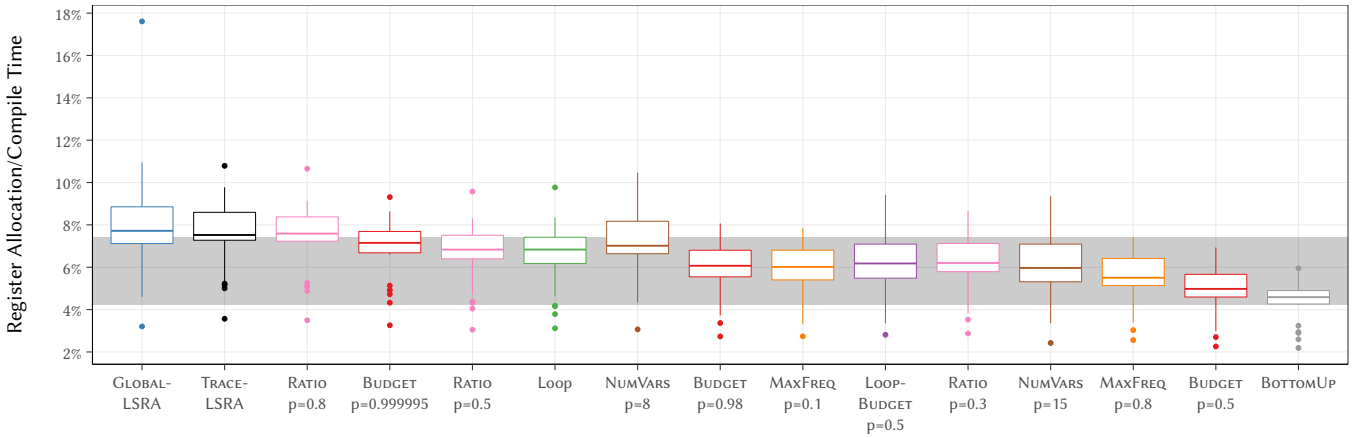


Figure 8: Register Allocation Time Relative to Overall Compilation Time

The gray area highlights the range between the mean of TraceLSRA and BottomUp, i.e., the potential range for improving compile time.

5.3 Evaluation Metrics

We repeated every experiment at least 30 times to compensate for variation factors that we cannot control, such as low-level hardware differences or non-determinism of the virtual machine. For every metric we use the *arithmetic mean* for each benchmark and for every configuration. Since we average raw numbers, the *arithmetic mean* is appropriate [according to Fleming and Wallace, 1986; Smith, 1988]. These means are then normalized to the trace-based linear scan allocator. We present the normalized numbers as box plots [Tukey, 1977] to give an unbiased impression of the distribution of results across the benchmarks.

Peak Performance. The reported performance result for the DaCapo-style benchmarks is the time required for the last iteration (*Benchmark Execution Time* in Figure 6). Ideally, in this iteration the VM does not perform any compilation. However, we cannot exclude compilations completely due to the behavior of the harness and for instance the use of bytecode generation in a benchmark. The peak performance is shown in the top half of Figure 6.

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 timing.

The meta-circular aspect of the GraalVM adds another layer of challenges to the problem. Since the compiler itself (which is written in Java) is subject to compilation, changes in the compiler influence not only the generated machine code, but also the time it takes to translate the compiler itself. 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. Figure 8 depicts the fraction of overall compile time that is used for register allocation.

5.4 Analysis of the Results

The baseline for all our experiments is the trace-based linear scan allocator, denoted by TraceLSRA. To visualize all benchmarks on the same scale, we show the numbers relative to the geometric mean of the baseline of a given benchmark. For 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*, the time used by the allocation algorithm, and the time used for allocation strategy selection.

Figure 6 shows the total register allocation time relative to the trace-based linear scan. We include all compilations of the benchmarks, including warm-up iterations, since the peak-performance result of the last iteration depends on all these compilations.

Figure 7 depicts the distribution between the allocation strategies for a given configuration. The numbers suggest that there is a correlation between the percentage of linear-scan-compiled traces and the register allocation time in Figure 6.

Unless otherwise noted, the numbers mentioned in this section represent the geometric mean of the averaged benchmark results relative to TraceLSRA.

GlobalLSRA. For comparison, we also show the results for global linear scan, which is the default allocator used by Graal. On average, GlobalLSRA behaves similar to the TraceLSRA policy for both allocation time and peak performance. Figure 6 shows allocation time outliers for GlobalLSRA, which are worse than the trace-based policies. The most severe outlier is the `jython` benchmark from the DaCapo suite where the global linear scan implementation shows a non-linear behavior.

On the other hand, the peak performance for the `sunflow` benchmark is 9% better than for TraceLSRA. This benchmark is very sensitive to spilling decisions and triggers the worst case behavior of the trace-based register allocation [Eisl et al., 2016].

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 experiment we evaluated the Ratio policy with the parameters $p \in \{0.8, 0.5, 0.3\}$. Although the number of linear-scan-allocated traces decreased significantly 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. 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 the 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. Around 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 MaxFreq 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 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 went down to 76%.

One interesting observation is that the NumVars policy seems to be more robust against performance outliers than policies with similar average values. For $p = 15$ the worst performance degradation is 13%, while, for example, for the MaxFreq ($p = 0.1$) it is as high as 20%, although the MaxFreq performs better on average (5% vs. 3%).

5.5 Impact on Overall Compile Time

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 8 shows how much of the overall compile time can be accounted to register allocation. With TraceLSRA, 7% of the time is used for register allocation, while in the BottomUp configuration this number goes down to 4%. The shaded area in Figure 8 visualizes the range of tuning possibilities. For a Graal configuration that is tuned towards compile time rather than peak performance, register allocation would make up a significantly larger portion of the overall compile time.

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 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, i.e., dynamic compilation of the most relevant parts based on the current execution profile. 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 focusing on *methods* as the unit of operation, trace compilation systems, such as Dynamo by Bala et al. [2000], HotPathVM by Gal et al. [2006] or HotSpot VM adaptions by Häubl and Mössenböck [2011], take a different route. They *trace* the execution of the program, potentially across method boundaries, and then select such a recorded trace for compilation. This way they compile only the parts of a program that are performance-critical, which narrows the scope of the compilation unit and therefore improves compile time. In our approach, the compilation unit is a method (not a trace),

but we use different register allocation strategies for different traces of a method based on structural properties of the traces. To the best of our knowledge, this has not been tried before.

6.3 Register Allocation

The design decisions taken for a compiler depend heavily on its application domain and its intended usage. This is especially relevant for register allocation since it is mandatory. Compilers used for *static* compilation are less restricted in terms of compile time than *dynamic* compilers in a virtual machine, where compilation time contributes 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* scope, or on 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 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 by 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.

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*. The 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. However, we can change the allocation algorithm for each trace even within a method. This allows more fine-grained control over the compile-time vs. peak-performance trade-off.

Approaches similar to our bottom-up register allocator were described previously for *local register allocation*, e.g., by Cooper and Torczon [2011, Chapter 13]. A major difference is that we apply the algorithm to a trace, i.e., to a list of basic blocks, instead of to a single block only. While we have to deal with data-flow between blocks, this requires only minor adaptions, due to the simple structure of our traces. Also, we initialize our variable/location map to match the successor trace to avoid data-flow mismatches, which is usually 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. 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 excluded such optimizations.

Also 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 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.

Our work builds on the trace-based register allocator of Eisl et al. [2016], as detailed in Section 2. The idea of using traces as the unit of operation was introduced by Fisher [1981] for instruction scheduling in 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 none of them provides the flexibility of our framework. Also, since their system was designed for static compilation, compile time was not a priority.

7 CONCLUSION AND FUTURE WORK

Our 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 currently choose between three register allocation strategies: a linear-scan-based algorithm, a fast bottom-up allocator and a specialized approach for trivial traces. The bottom-up allocator is 43% faster than the trace-based linear scan implementation at a performance degradation of 11% on average.

To assess how flexibly we can trade compile time against peak performance, we implemented and studied 8 policies (14 configurations in total) for deciding which register allocator to use for a specific trace. The Budget policy with a parameter $p = 0.999995$, for instance, improves register allocation time by 7% on average compared to the trace-based linear scan approach with an average peak-performance slowdown of only 1%. On the other hand, the NumVars policy decreases allocation time by about 24% with a performance degradation of 5% but exhibits a better worst-case behavior than the bottom-up approach. 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.

Future work will investigate further policies that might have even better performance trade-offs. One specific aspect is that most

of our policies can be parameterized. While we experimented with different settings, 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 LoopBudget policy. Since the search space for policies is large, we believe that using auto-tuning tools, such as OpenTuner [Ansel et al., 2014], is an idea that is worth investigating.

The *rule induction* technique, used by Cavazos et al. [2006] for their *hybrid optimizations* approach, is another option that should be considered. However, the generation of *training data* for our trace-based setting is an open question. Cavazos et al. allocate every method twice, once using the graph coloring allocator and once with linear scan. For both invocations they collect the number of *spill moves* to decide which strategy is preferred. This is not feasible for our approach since we would need to evaluate all combinations for a method (i.e., $\#traces^{\#strategies}$).

We plan to further explore on which properties policies should be based. So far, 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. We also plan to explore whether considering specific instructions in a trace can be exploited to select an allocation policy.

In this paper we focused on improving compile time. In the future, the same ideas could be applied to achieve better peak performance, i.e., add allocation strategies that find better solutions than linear scan. Because of the properties of traces, it might even be feasible to do an optimal register allocation for a trace or a set of traces. For such an approach, our policy framework can be used to keep the register allocation time within bounds.

The trace-based approach in general and our policy model in particular are 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. Special thanks to Doug Simon for his comments on the draft version of this paper. We also thank the anonymous reviewers for their valuable feedback. Josef Eisl is funded in part 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. ISBN: 0321486811. 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: *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.

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.

Brandis, Marc M. and Hanspeter Mössenböck. 1994. Single-pass Generation of Static Single-assignment Form for Structured Languages. In: TOPLAS'94. ISSN: 0164-0925. DOI: 10.1145/197320.197331.

Briggs, P., K. D. Cooper, K. Kennedy, and L. Torczon. 1989. Coloring Heuristics for Register Allocation. In: *PLDI '89*. ACM. DOI: 10.1145/73141.74843.

Cavazos, John, J. Eliot B. Moss, and Michael F. P. O'Boyle. 2006. Hybrid Optimizations: Which Optimization Algorithm to Use? In: *CC '00*. Springer Berlin Heidelberg. DOI: 10.1007/11688839_12.

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.

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. ISSN: 0164-0925. 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. ISSN: 0018-9340. DOI: 10.1109/TC.1981.1675827.

Fleming, Philip J. and John J. Wallace. 1986. How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results. In: Communications of the ACM. ISSN: 0001-0782. DOI: 10.1145/5666.5673.

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. ISSN: 0164-0925. DOI: 10.1145/229542.229546.

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.

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

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. ISSN: 1544-3566. 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: *PLDI '97*. ACM. DOI: 10.1145/258915.258926.

Poletto, Massimiliano and Vivek Sarkar. 1999. Linear Scan Register Allocation. In: TOPLAS'99. ISSN: 0164-0925. 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. ISSN: 1544-3566. DOI: 10.1145/2764907.

Table A: Experimental Results

	Register Allocation Time (Δ%)						Peak Performance (Δ%)						Strategy (%)		
Policy	mean	median	min	min	max	max	mean	median	min	min	max	max	LS	BU	TT
GlobalLSRA	1.3	−2.1	−12.0	scalatest	96.3	jython	0.0	0.0	−9.0	sunflow	6.4	apparat			
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 peak performance (lower is better). For *min* and *max* we also show the corresponding benchmark. The given numbers are the difference 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).

Smith, J. E. 1988. Characterizing Computer Performance with a Single Number. In: Commun. ACM. ISSN: 0001-0782. DOI: 10.1145/63039.63043.

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: *PLDI '98*. ACM. DOI: 10.1145/277650.277714.

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.