Trace-based Register Allocation in a JIT Compiler

Josef Eisl[†]
Institute for System Software
Johannes Kepler University
Linz, Austria

Matthias Grimmer[†]
Institute for System Software
Johannes Kepler University
Linz, Austria

Doug Simon[‡]
Oracle Labs
Switzerland

Thomas Würthinger[‡]
Oracle Labs
Switzerland

Hanspeter Mössenböck[†]
Institute for System Software
Johannes Kepler University
Linz, Austria

[†]{josef.eisl,matthias.grimmer,hanspeter.moessenboeck}@jku.at

[‡]{doug.simon,thomas.wuerthinger}@oracle.com

ABSTRACT

State-of-the-art dynamic compilers often use global approaches, like Linear Scan or Graph Coloring, for register allocation. These algorithms consider the complete compilation unit for allocation, which increases the complexity of the implementation (e.g., support for lifetime holes in Linear Scan) and potentially also affects compilation time. We propose a novel non-global algorithm, which splits a compilation unit into traces based on profiling feedback and subsequently performs register allocation within each trace individually. Traces reduce the problem size to a single linear code segment, which simplifies the problem a register allocator needs to solve. Additionally, we can apply different register allocation algorithms to each trace. We show that this non-global approach can achieve results competitive to global register allocation.

We present an implementation of Trace Register Allocation based on the Graal VM and show an evaluation for common Java benchmarks. We demonstrate that performance of this non-global approach is within 3% (on AMD64) and 1% (on SPARC) of global Linear Scan register allocation.

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

DRAFT — to be submitted to PPPJ '16
(created on October 2, 2017)

1. INTRODUCTION

Register allocation is an important compiler optimization [1–4] that has been the focus of intensive research. Its task is to map *variables*[1] to physical *registers* of the processor. If the number of variables that are live at the same time exceeds the number of available registers, some variables need to be stored in memory (i.e., *spilled*). A variable can also be stored in multiple locations during its lifetime (i.e., *split*).

Chaitin et al. [5] introduced Register Allocation via Graph Coloring, which was the first widely applied approach for *global register allocation*, in which registers are allocated for the whole compilation unit (method) at once instead of allocating them per block. Later numerous refinements and improvements were proposed including Briggs et al. [1], Smith et al. [6], or George et al. [7]. Chaitin et al. showed that optimal register allocation is in general NP-complete [5], so all graph-coloring approaches use heuristics to achieve polynomial run-time behavior.

In spite of these heuristics, the worst-case complexity of graph-coloring register allocation is usually quadratic, which makes it sub-optimal for just-in-time (JIT) compilation. JIT compilers therefore often use the simpler Linear Scan approach, which was introduced by Poletto et al. [2] and extended in many subsequent contributions [3, 4, 8]. Linear Scan is used in numerous dynamic compilers including the HotSpot client compiler [9], the Jikes RVM [10], Google's JIT compiler for JavaScript (V8), and initially also in LLVM [11].

Like Graph Coloring, Linear Scan is a global register allocation approach meaning that the algorithm works on the whole compilation unit. The idea is to bring basic blocks of the control-flow graph into a linear order. The liveness of a variable is expressed as an interval along this linear order.

[1] Others use the term *temporaries* to distinguish between intermediate results of the evaluation of an expression and variables in the source code of the program. We call every non-physical, non-constant operand (i.e. all operands that are subject to register allocation) in the intermediate representation of the compiler a *variable*.

The intervals are then traversed based on their start positions, from earliest to latest. In the original proposal [2] an interval only consists of a single live range. An example is shown in Figure 1(b). In general, liveness information cannot be expressed as a continuous range on a linear list of blocks. The approach by Poletto and Sarkar overestimates liveness which can lead to unnecessary spilling in cases where a register is still available.

To mitigate this problem approaches with exact lifetime information were proposed [3, 4, 8] where intervals can contain so-called *lifetime holes*, i.e. ranges where the value of a variable is not needed. This information allows these approaches to find better allocations compared to the original Linear Scan. However, lifetime holes introduce a more complex representation which leads to an allocation algorithm which is no longer linear [3, 8]. Figure 1(c) depicts an example of intervals with lifetime holes.

Another issue with Linear Scan is that, due to the linearization of the basic blocks, control-flow is not directly visible to the algorithm. For example, finding optimal positions for inserting spill moves or for deciding where to split an interval needs control-flow context.

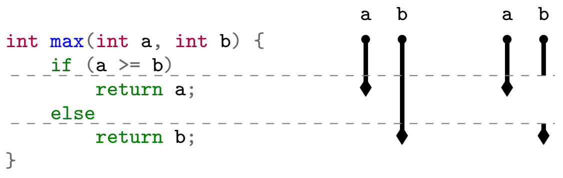
Since Linear Scan is a global register allocation approach all parts of a compilation unit influence each other. Moreover, uncommon (*cold*) code parts can have a negative impact on frequently executed (*hot*) parts. For example, a sub-optimal spilling position in a cold part might be preferable to a globally better spilling position in a hot path.

Trace Register Allocation is a novel register allocation approach which is specifically designed for JIT compilation with run-time feedback. It addresses the shortcomings described above. In contrast to global approaches, Trace Register Allocation divides the blocks of the control-flow graph into disjoint subsets of sequentially executed blocks (so-called *traces*) and allocates registers for each subset independently.

We present the details of a successful Trace Register Allocation [12] implementation for the Graal compiler [13–18], an optimizing compiler for the Java HotSpot VM [19]. We show that our new implementation is capable of finding register allocations that are comparable to those of the Linear Scan implementation currently used by Graal [3, 4].

This paper contributes the following:

- We present Trace Register Allocation, a modular, non-global framework for allocating registers. It allows doing register allocation independently on different traces of a compilation unit (possibly using different algorithms).
- Two simple algorithms for finding traces in a control-flow graph that are suitable for trace register allocation.
- An extension of our low-level intermediate representation (LIR) to capture liveness information at basic block boundaries that simplifies working with traces.
- Two local inter-trace optimizations: *inter-trace hints* to reduce the number of data-flow resolution moves and *spill information sharing* avoid unnecessary spilling.
- We evaluate our Trace Register Allocation implementation and show that the approach is capable of finding allocations comparable to Linear Scan.



```
int max(int a, int b) {
    if (a >= b)
        return a;
    else
        return b;
}
```

(a) Source Code **(b) Original** **(c) Holes**

Figure 1: Lifetime Intervals

The rest of the paper is organized as follows. We start with a system overview. Section 3 describes the Trace Register Allocation approach and our implementation of it. In Section 4 we present inter-trace optimizations which improve the quality of the code generated by the allocator. We evaluate our implementation in Section 5. Section 6 discusses related work and compares it to our approach. We conclude the paper with a summary and an outline of future work in Section 7.

2. SYSTEM OVERVIEW

Graal is an optimizing compiler for the Java HotSpot VM that translates Java Bytecode to native machine code. It incorporates two different intermediate representations. In the *frontend* Graal uses a graph-based representation (i.e., high-level IR or HIR) [14]. On this level Graal performs optimizations such as (speculative) inlining, dead-code elimination, conditional elimination, partial escape analysis [17] and loop unrolling to name just a few [15]. After applying all optimizations, the graph-based representation is scheduled and converted into a low-level intermediate representation (i.e., LIR). This IR is in SSA (Static Single Assignment [20]) form and is based on a control-flow graph with basic blocks. Blocks contain a list of (LIR) instructions that are close to the actual machine operations. The *backend* uses this LIR for register allocation and code emission. Note that, although the instructions in the LIR already represent machine operations, the backend itself is machine-independent.

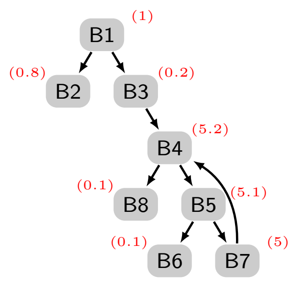
3. TRACE REGISTER ALLOCATION

Register allocation in a dynamic compiler is done at application run time, hence, the algorithm needs to find a trade-off between compile time and quality of the allocation. Second, since the compiler is invoked dynamically, the execution environment can provide profiling information about the behavior of the application.

Most of an application's run time is spent in a fraction of its code [21]. Ideally, we should therefore invest more time optimizing these important parts of the compilation unit and should not spend too much time on insignificant parts [22]. In addition, we should offload spill and split code into these cold parts if possible.

3.1 Overview

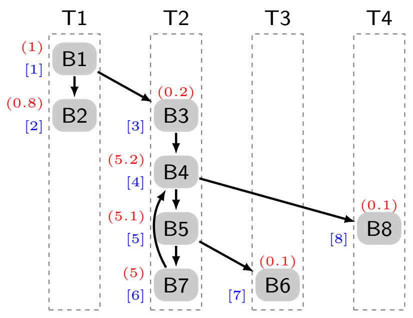
The key idea of our approach is to divide the control-flow graph into distinct regions of sequentially executed blocks. We use the profiling feedback provided by the runtime system to find a suitable partition. Due to similarities to trace scheduling [23, 24] we call these regions *traces*. Figure 2(c) and Figure 2(d) show possible traces of the control-flow graph in Figure 2(a).



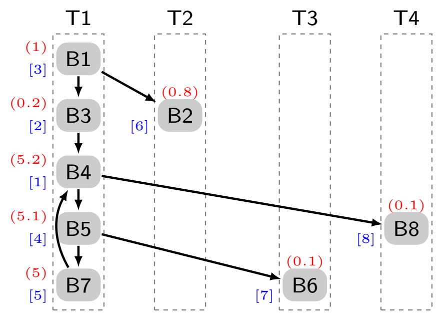
(a) Control-flow Graph

```
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;
}
```

(b) Java Source



(c) Unidirectional Trace Builder



(d) Bidirectional Trace Builder

Figure 2: Trace-building Example – Source code, control-flow graph and trace building results of `java.utils.Arrays#equals` from the OpenJDK. (`null`-checks omitted for simplicity.) The values in parentheses (e.g. (0.8)) denote relative execution frequencies. The numbers in square brackets (e.g. [2]) is the order the trace builder processed the block.

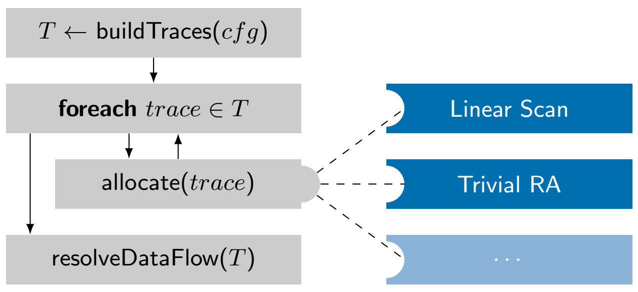


Figure 3: Trace Register Allocation Overview

For each of these traces we perform register allocation independently. This allows mixing different approaches, for instance Linear Scan and Graph Coloring, during a single compilation. Allocating registers for a trace includes *lifetime analysis* as well as replacing variables with the assigned locations (i.e. registers or spill-slots). After we have allocated all traces we need to resolve data-flow mismatches between traces since a variable might have been allocated to different locations in different traces. Figure 3 depicts an overview of the algorithm. We will cover the details for each step in the following sections.

3.2 Trace Building

Definition of Traces.

A trace is a distinct sequence of basic blocks in a control-flow graph. We call the first block in a trace its *head*. Every block is part of *exactly one* trace and all traces are *non-empty*. Note that the blocks of a trace can have incoming or outgoing edges to other traces. These are called *inter-trace edges*. We identify so-called *critical edges* in the control-flow graph, i.e., edges between blocks X and Y where X has more than one successor and Y has more than one predecessor. For critical edges we insert an empty block (e.g. B4 in Figure 4(c)). If this block is still empty after register allocation it is removed before code emission.

In the simplest case, every block can be treated as a separate trace. This would be equivalent to local register allocation. However, longer traces offer more opportunities for moving spill code (e.g., out of loops) and thus to improve the quality of the register allocation, so we need more sophisticated trace building strategies.

In the following, we propose two trace building strategies that utilize the run-time feedback provided by the execution environment.

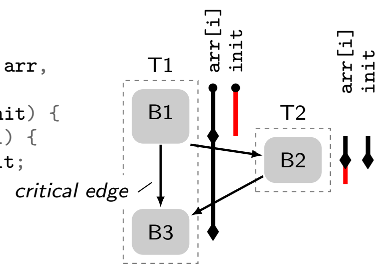
Unidirectional Trace Building.

To build a new trace we select a block with no predecessors or one where all predecessors are already part of another trace. If there are multiple candidates we choose the block with the highest execution frequency. The block is added to the trace and we proceed with the successor of highest probability. We repeat this until there is no successor that is not already part of a trace. Figure 2(c) depicts an example. Note that this strategy tries to minimize the trace exit probability.

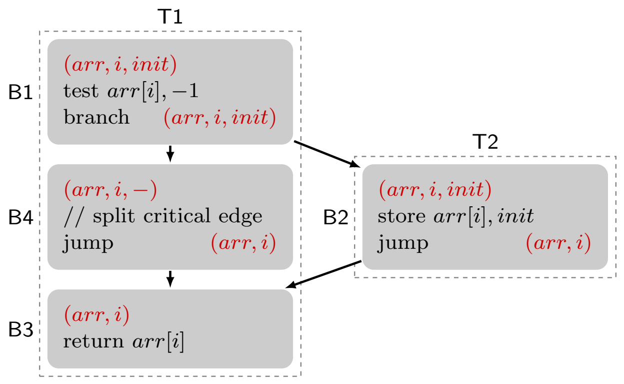
```
int getOrInit(int[] arr,
              int i,
              int init) {
    if (arr[i] == -1) {
        arr[i] = init;
    }

    return arr[i];
}
```

(a) Source Code



(b) Lifetime



(c) Annotated IR

Figure 4: Lifetime on Traces

Bidirectional Trace Building.

For the bidirectional trace builder we select the block with the highest execution frequency from the set of *candidate blocks*, i.e., the blocks that are not yet part of a trace. From this initial block, we first grow the trace upwards. Among all predecessors from the candidate set we select the one with the highest execution frequency and prepend it to the trace. Note that we do not follow loop back-edges. This means after we have processed a loop header we always continue with the block entering the loop (or stop), never with the loop end block. Once there is no candidate we start the downwards pass, again starting at the initial block. We proceed in a way that is similar to the unidirectional trace builder. Bidirectional trace building has already been described by Ellis [23] and Lowney et al. [24] although they used it for other purposes than register allocation.

Figure 2(d) shows that the traces formed by the two strategies can differ.

We evaluated both approaches and came to the conclusion that the unidirectional trace builder is not only simpler but achieves better results than its bidirectional counterpart (see Section 5).

3.3 Intermediate Representation

Liveness Information.

In Trace Register Allocation, we can use different allocation algorithms for our traces (e.g., Graph Coloring or Linear Scan). These algorithms should operate locally on a single trace without considering other traces. However, the liveness of a variable needs to be considered globally. Let us assume the following example (Figure 4(c)): In trace T1 we need to know that variable `init` is used in trace T2 and therefore we need to keep it alive until the end of block B1.

We can solve this by performing a global liveness analysis before register allocation and by annotating each basic block in the IR with this information. We call this information *incoming* and *outgoing live-sets* of a block. Figure 4(c) shows our extended intermediate representation of the code in Figure 4(a). After this analysis, the allocation algorithm can work on a single trace without considering other traces, hence breaking the inter-dependency between traces.

Handling inter-trace edges.

We apply register allocation to each trace independently. At points where traces are split or merged (*inter-trace edges*), the variables might have been allocated to different physical locations. We need to resolve this by inserting moves from the location in the predecessor block to the location in the successor block. This is similar to data-flow resolution (and φ-deconstruction) in the (SSA-based) Linear Scan register allocator [3, 4]. Since the register allocation algorithm updates the *live-sets* of the blocks with the actual physical locations we can use this information to resolve data-flow mismatches.

Both, the global liveness analysis as well as the data-flow resolution are independent of the register allocation algorithm.

3.4 Allocating Registers for a Trace

One advantage of our approach is that allocating registers in one trace is completely independent of register allocation in other traces. In general, any approach can be used for register allocation within a trace. Also, traces have properties that are worth exploiting. Since there are no lifetime holes in a linear code sequence the liveness information is simpler to compute and to maintain than in a global register allocation algorithm.

Currently, we use two different allocation strategies. The main strategy is a Linear Scan approach derived from the global register allocator that is our baseline. The second one is an allocator for *trivial traces*, i.e., traces consisting of just a single `jump` instruction.

Linear Scan Register Allocation.

We process our traces with the same Linear Scan register allocation algorithm that is used by our baseline for global allocation.

Linear Scan register allocation on a single trace simplifies the algorithm as follows: First, due to the extension of our intermediate representation the analysis can be done in a single backwards pass over the trace. Second, since there are no lifetime holes in a trace, an interval is defined by a single live-range with a *from* and a *to* position. In contrast to that, the global approach needs to maintain a list of live-ranges for every interval. Consequently, we do not need to track the set of *inactive intervals*, a source of non-linearity in the global Linear Scan algorithm [3, 4].

Performing Linear Scan allocation on single traces also simplifies spilling and splitting of intervals. If we run out of registers we select an interval and spill it to the stack. Since a trace is a sequence of blocks we can place the move anywhere between the definition of the variable and the position where the interval is to be split, i.e., the position where we

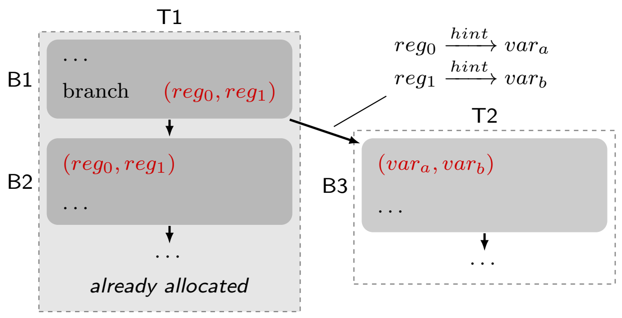


Figure 5: Inter-trace Hints

ran out of registers. We place it in the basic block with the lowest execution frequency. Splitting creates a new single-live-range interval for the variable on the stack. If this variable is needed in a register again later, we use the same strategy for splitting the interval and for deciding where to reload the variable from the stack back into a register.

Trivial Trace Allocator.

A *trivial trace* is a trace consisting of a single basic block which contains only a *jump* instruction. Such blocks are mainly the result of splitting *critical edges*. Their purpose is to transfer the values from the predecessor to the successor block. Experiments showed that trivial traces are common. In the DaCapo benchmark suite, for instance, 44% of all processed traces are trivial. Register allocation is easy for such a trace. We simply map the incoming values to the outgoing ones by propagating their locations.

The Trivial Trace Allocator is an example of an allocator that is crafted for a sub-class of general traces. This flexibility is a key advantage of our trace-based approach over global approaches.

3.5 Data-flow Resolution

Since register allocation is performed independently for each of the traces, a variable might end up in one location in one trace and in a different location in some other trace. To fix this we need a *data-flow resolution* phase to align the mismatch on inter-trace edges. For each of these edges we compare the locations in the *outgoing live-set* of the predecessor block to the location in the *incoming live-set* of the successor block. If they differ we introduce a move. Note that all moves along an edge are semantically performed *in parallel*, so we need to order them correctly and break data-dependency cycles. Similar to Linear Scan [4], resolution also deconstructs the SSA-form.

It is important to note that due to our intermediate representation this process is independent of the allocation algorithm or the notion of liveness that is used for each trace.

4. INTER-TRACE OPTIMIZATIONS

Data-flow resolution is sufficient for the correctness of Trace Register Allocation. However, the allocation quality can be improved by performing local inter-trace optimizations. We propose two optimizations to reduce the number of moves introduced during data-flow resolution and to avoid unnecessary spill moves. Both algorithms work on a local basis and do not increase the complexity of the register allocation algorithms.

4.1 Inter-trace Hints

Although register allocation for a trace is independent from register allocation for other traces, it is still beneficial to use information from already allocated traces. Assume that we are allocating a trace T2, which has an incoming inter-trace edge from an already allocated trace T1 (Figure 5). We annotate the intervals in T1 with *hints* [3] to the location of the variable in the predecessor trace T2. In our example the hints are $reg_0 \rightarrow var_a$ and $reg_1 \rightarrow var_b$. As the name suggests, this is a *hint* for the allocator to use a specific location for a variable if possible. This optimization reduces the number of moves inserted in the data-flow resolution phase. The order in which traces are processed has an impact on the effectiveness of this optimization. We use the order in which the traces where created during trace building, which approximates their importance.

4.2 Spill Information Sharing

Since our IR is in SSA-form, every variable can hold only a single value. Due to spilling and spill-position optimization a value might be available in two locations at the same time, i.e., in a register and in a stack-slot. If this is the case at the predecessor of an inter-trace edge we can exploit it to avoid redundant spill moves. Similar to the inter-trace hints we inform the allocator that the value is not only available in a register but also in a stack slot. Intervals with this information are preferred candidates for spilling since their value is already available in memory so that no spill move needs to be inserted.

5. EVALUATION

We evaluated our Trace Register Allocation approach in the context of the Graal compiler [13–16, 18] which runs on top of the HotSpot VM [19]. The HotSpot VM comes with two just-in-time compilers, the *client compiler* [9] and the *server compiler* [25]. 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. We use the JVM Compiler Interface [26], which will be part of the upcoming Java 9 release, to use Graal instead of the server compiler as the second-tier compiler.

The code produced by Graal is faster than the code generated by the client compiler and comparable to the code generated by the server compiler (2–10% slower depending on the benchmark [15, 18]).

We used Graal revision `a563a1d51507` for our evaluation.[2] For the experiments we deployed Graal on top of a modified Java 8 VM which includes the JVMCI.[3] We executed the HotSpot VM in *tiered-mode* meaning that execution of a method starts in the interpreter. After hitting an execution frequency threshold, hot methods are compiled by the client compiler. If the compiled method exceeds another threshold it is compiled by Graal. The raw data for all experiments presented in this paper is available from our website.[4]

Benchmarks.

We evaluated our results using the DaCapo 9.12 [21, 27], the Scala-DaCapo [28], the SPECjvm2008 [29] and the

[2]https://github.com/graalvm/graal-core
[3]http://hg.openjdk.java.net/graal/graal-jvmci-8/
[4]http://ssw.jku.at/General/Staff/Eisl/TraceRA/PPPJ16/

name	value
Type	Sun Server X3-2 [31]
CPU Model	Xeon E5-2690 @ 2.90GHz
CPU Config	2 packages; 2x8 cores; 2x8x2 threads
Memory	192GB
Hard Disk	2x 300GB HDD (RAID1)
OS	Ubuntu 12.04
CPU (lxc)	8 cores; 8x2 threads
Mem (lxc)	96GB
Other (lxc)	ramdisk

Table 1: Sun Server X3-2 (AMD64)

name	value
Type	Sparc T5-2 Server [33]
CPU Model	Sparc T5 3.60GHz
CPU Config	2 packages; 2x16 cores; 2x16x8 threads
Memory	256GB
Hard Disk	2x 300GB HDD (zfs mirror)
OS	Oracle Solaris 11.2
CPU (zone)	14 cores; 14x8 threads (2 cores for host)
Mem (zone)	96GB

Table 2: SPARC T5-2 (SPARC)

SPECjbb2015 [30] benchmark suites. For DaCapo, Scala-DaCapo and SPECjvm2008 we ran the individual benchmarks in a distinct VM process. We omitted the `startup` benchmarks for SPECjvm2008 since they are not relevant for this work. Also, we did not run the `compiler.sunflow` benchmark from SPECjvm2008 as well as the `eclipse` benchmark from DaCapo due to Java 8 compatibility issues.

Environment.

Since register allocation is highly influenced by the underlying processor, we performed the experiments on AMD64 (Intel X86 64bit) as well as on a SPARC T5 to gain confidence that our approach is applicable to different architectures.

For the AMD64 experiments we used a Sun Server X3-2 [31] with an Intel Xeon E5-2690 processor running Ubuntu 12.04. We used Linux Containers (lxc) [32] to restrict the VM to a single package of the machine to avoid negative influence due to switching nodes during benchmark execution. The details of the machine can be found in Table 1.

The second machine is a Sparc T5-2 Server [33] running Oracle Solaris 11.2. Similar to the containers in the AMD64 setup we used Solaris Zones [34] for isolation of the sockets. More information is shown in Table 2.

Configurations.

The baseline for our experiments is the global Linear Scan implementation [3, 4] (denoted by LSRA) that is used in Graal by default. If not stated otherwise, all results are normalized to the mean of Linear Scan. We use a box plot [35] to visualize the results of our experiments.

The TraceRA configuration is our Trace Register Allocator implementation in Graal with all optimizations enabled. The TraceRA BiTB variant uses the *bidirectional trace building* approach for finding traces. In the TraceRA noSpillSharing configuration *spill information sharing* is turned off. For the TraceRA noHints noSpillSharing variant we also disabled *inter-trace hints* (see Section 4).

name	iterations
avrora	10
batik	40
fop	40
h2	20
jython	40
luindex	15
lusearch	40
pmd	30
sunflow	30
tomcat	50
tradebeans	20
tradesoap	15
xalan	20

(a) DaCapo

name	iterations
actors	10
apparat	5
factorie	5
kiama	40
scalac	20
scaladoc	15
scalap	120
scalariform	30
scalatest	50
scalaxb	35
specs	20
tmt	12

(b) Scala DaCapo

Table 3: Benchmark Iterations

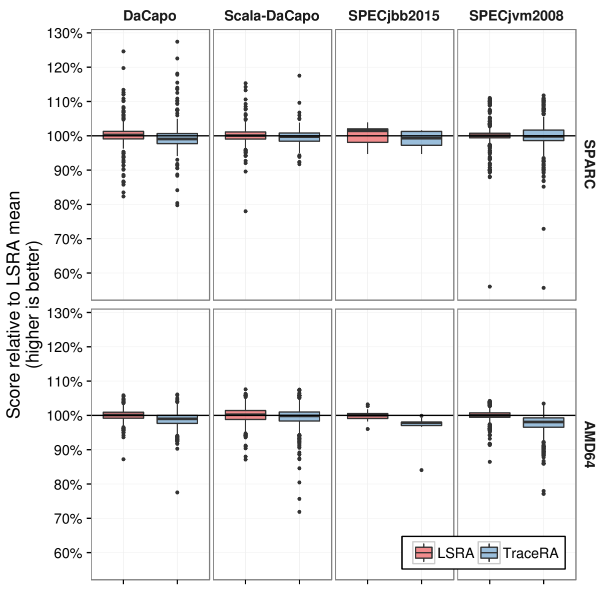


Figure 6: Performance (Composite)

5.1 Performance

For this evaluation we are interested in peak performance of long-running applications. Figure 6 depicts the distribution of the individual benchmark results for every suite. It contains all normalized data points from the different benchmark suites.

DaCapo.

The DaCapo benchmarking harness executes every benchmark a certain number of times in a single VM. The iteration count is used to warm up the virtual machine, i.e., to compile all important methods. It depends on the size of the benchmark. For all benchmarks we ran a sufficient number of warmup iterations (in order to get all important methods compiled) before we measured the performance. The iterations used for this experiment are listed in Table 3(a). In Figure 6 and Figure 7 we present the reciprocal of the run time, which is the number reported by the DaCapo (and

Scala-DaCapo) harness. This way the value can be more easily compared to the results of the SPEC benchmarks. The reported number is the score of the last iteration.

The results for the DaCapo benchmark suite on AMD64 are depicted in Figure 7. The `composite` column depicts all normalized data points of a suite. On average, Trace Register Allocation is about 1% slower then Linear Scan. The figure clearly shows that *inter-trace hints* are the most influential optimization. It also suggests that the *unidirectional trace builder* performs better than the *bidirectional* variant. The `sunflow` benchmark is interesting since it is the only one which is more than 5% slower on average (Figure 7). We will discuss the reasons for that at the end of the evaluation section. For SPARC the performance numbers are similar: Trace Register Allocation is about 1% slower than Linear Scan, on average. Also for `sunflow`, the performance difference is only 1% here.

Scala-DaCapo.

The Scala-DaCapo benchmarking harness is similar to the one of DaCapo. Again, we ran a sufficient number of warmup iterations before we measured the peak performance (Table 3(b)). For the Scala-DaCapo benchmarks Trace Register Allocator achieves results in the same range as Linear Scan, both on AMD64 and on SPARC.

SPECjvm2008.

The harness for SPECjvm2008 differs from DaCapo as it is time-based, not iteration-based. It warms up the benchmark for $120s$ followed by $240s$ interval for the actual measurement. During these time intervals the benchmark is executed repeatedly. The harness reports a score value in operations per minute. The results are presented in Figure 6. On average we are as close as 3% to the Linear Scan performance on AM64. There is measurable difference in performance on SPARC.

SPECjbb2015.

In contrast to the other benchmarking suites, SPECjbb2015 does not consist of multiple independent benchmarks but executes a single business application. The harness provides two metrics, the `critical` score which relates to response time and the `max` value which measures throughput. The results are depicted in Figure 6 and show that we are on average about 3% slower on AMD64 and 1% slower on SPARC respectively.

Known Issues.

One issue with our approach are spill moves that are introduced in a side-trace of a loop. Figure 8 shows an example: In trace `T1` the allocator is able to move the spill code for `x` out of the loop (i.e., from `B4` to `B1`). In trace `T2`, when spilling `y`, we cannot do this since the block entering the loop (`B1`) is not part of `T2`. This means that we need to execute the spill move inside the loop, every time we enter `B6`. Doing the spill also in `B1` would be preferable, but since allocation of traces is decoupled, this is not possible. (Note that in any case, we cannot remove the load of `y` at the end of `B6`.)

The effect of this issue can be seen in the results of the `sunflow` benchmark in Figure 7. The hot loop of the benchmark contains a `swich` statement where in every branch the register pressure is higher than the number of available registers. The Trace Register Allocator creates one long trace containing the loop header and the most likely branch of the switch. In this trace the spills can be moved out of the loop. For the other branches, separate traces are created where the spill moves cannot be lifted out of the loop. Our investigation showed that the number of executed moves is higher with Trace Register Allocator than with Linear Scan.

To solve this problem, we could conservatively insert spill moves in the trace with the loop header and remove them later if they turn out to be superfluous. Implementation and evaluation of this optimization remains future work.

5.2 Compile Time

Besides performance, compile time is also important for a dynamic compiler. Figure 9 shows the compile time per (LIR) instruction for DaCapo on AMD64 ($R^2 = 0.97$ [36]). We present the time required for the complete allocation stage including trace building, the global liveness analysis for our IR extension and register allocation. In our experiment 75% of time is spent on (local) register allocation of traces. The rest is consumed by global phases, i.e., trace building, global liveness analysis and global move resolution. The results show that our implementation of Trace Register Allocation exhibits a linear behavior. Improving compile time was not yet our focus and remains future work. Currently, our prototypical implementation is on average 35% slower than the Linear Scan algorithm that is used by Graal. Concrete ideas for improving the compile time include, for example, reusing data structures to avoid reallocations for every trace.

Note that the trace-based approach has two advantages over other approaches regarding compilation time. First, the lifetime information is simpler since we do not need to take care of lifetime holes. Second, we can use faster algorithms for unimportant traces.

Trace Register Allocation is already 10% faster than Linear Scan on the `jython` benchmark from the DaCapo suite. In this case Linear Scan shows a non-linear compile time behavior.

6. RELATED WORK

Trace Register Allocation is related to three separate areas of research, namely *trace scheduling*, *trace compilation* and, of course, *register allocation*.

Trace Scheduling.

Fisher [37] introduced traces as compilation units for instruction scheduling for Very Long Instruction Word (VLIW) architectures. The idea is to reorder instructions for increasing Instruction Level Parallelism (IPL). The Bulldog compiler by Ellis [23] and later Multiflow by Lowney et al. [24] are popular implementations of this approach. Their understanding of a trace being a part of the control-flow graph coincides with our use of the term *trace*. Many concepts developed in that field are similar to Trace Register Allocation such as the trace building or the sharing of information across trace boundaries. They also do register allocation but it is a by-product of the instruction scheduling algorithm while for us it is the main focus.

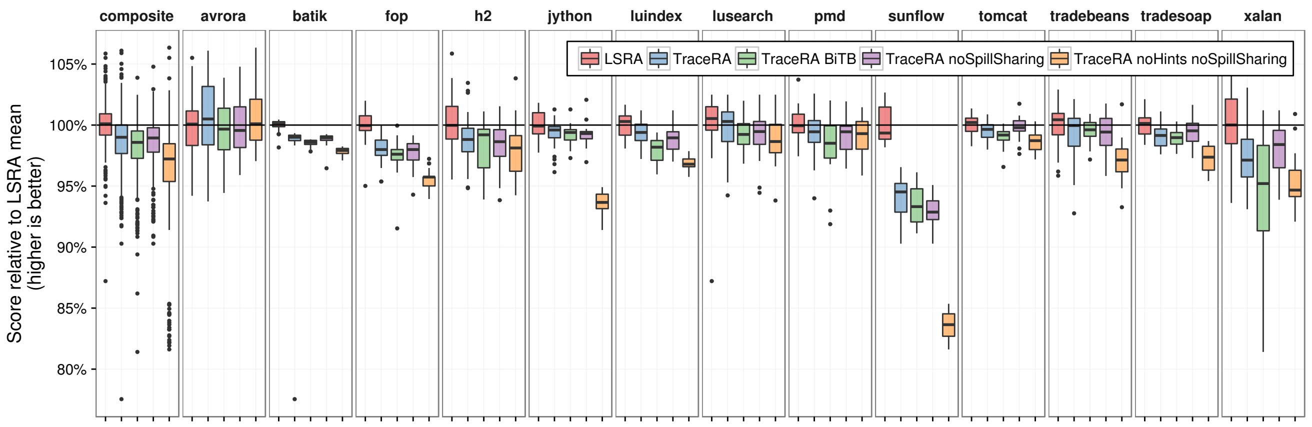


Figure 7: Performance DaCapo (AMD64)

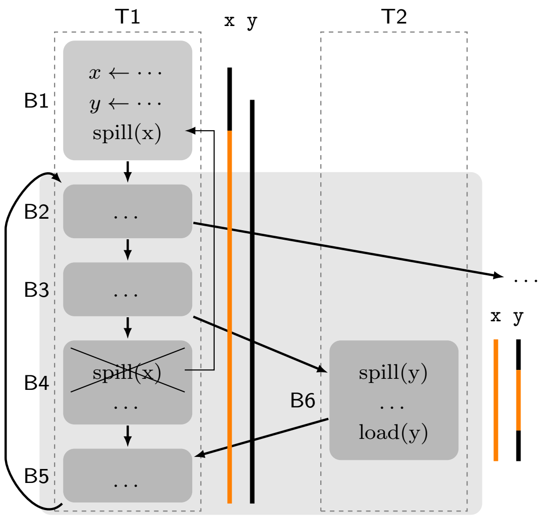


Figure 8: Spilling in Side-traces of a Loop

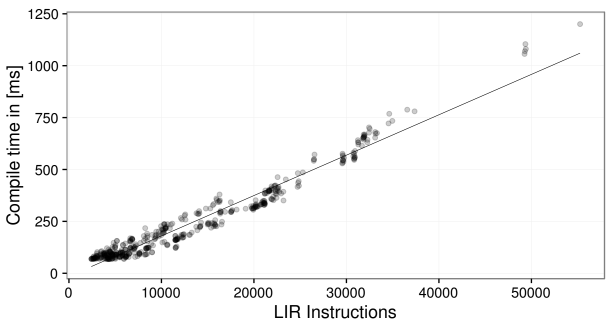


Figure 9: Compile time per LIR Instruction – 500 slowest methods of 74160 (DaCapo on AMD64)

Trace compilation.

Trace compilation is a dynamic compilation approach that uses traces as the compilation unit for generating machine code. Trace-based compilers do not depend on the structure of the program to be compiled (e.g., on its methods, conditional statements, or loops) but rather find their compilation units by recording traces during the execution of methods in an interpreter or in code that was produced by a baseline compiler.

Dynamo by Bala et al. [38] was the first widely known system using this approach for dynamic translation of compiled binaries. Gal, Probst, and Franz proposed the HotPathVM [39], a trace-based Java VM for resource-constrained devices. The HotPathVM performs *trace merging* where different traces are stitched together. When compiling the child trace they initialize the state of the variables to the mapping present at the end of the parent trace. The idea is similar to our *inter-trace hints*. Following up on their previous work, Gal et al. applied their trace-based approach to JIT compilation of dynamically-typed languages. The result was TraceMonkey [40], a tracing VM for JavaScript that was used in Mozilla's Firefox Browser. Bolz et al. used trace-compilation for obtaining efficient bytecode interpreters that were implemented in the PyPy language implementation framework [41]. They applied the tracing compiler not to the user program but to the interpreter that was written by the language implementer. This approach is also referred to as *meta-tracing*. Häubl and Mössenböck modified the HotSpot Client Compiler in order to perform trace-based compilation [42]. Especially the context-sensitive inlining of traces [43] showed significant improvements.

A difference between trace compilation and our approach is that, in general, traces have no side entries, i.e., a trace is always entered through its head (although there are exceptions [42]). In the context of trace compilation, code pieces (e.g. a basic-block or an instruction) can usually occur in multiple traces, i.e. there is some kind of code duplication. This is not the case in our implementation.

Although register allocation is not the focus of trace compilation, allocation strategies used in this area can also be applied to Trace Register Allocation. We assume that further research of trace-based register allocation will impact trace-based compilation and *vice versa*.

Register Allocation.

Poletto and Sarkar introduced Linear Scan [2] as a simple and fast method for global register allocation. In their approach, intervals are not split, so either the whole interval is on the stack or in a register. Also, they do not support lifetime holes. Both decisions make the algorithm simpler but at the cost of allocation quality. The *Second-chance Binpacking* algorithm by Traub et al. [8] added interval splitting and lifetime holes. Wimmer and Mössenböck improved this approach and suggested optimized spilling and splitting [3]. The SSA adoption of this work by Wimmer et al. [4] is the allocator currently used by Graal[5] and the baseline we compare against. The trace-based Linear Scan Algorithm used by our approach is also derived from this work. Since traces do not contain control-flow we do not need to maintain lifetime holes. Also, due to our intermediate representation, we can do the lifetime analysis in a single backwards pass.

Callahan and Koblenz proposed *Hierarchical Graph Coloring* as a technique to minimize the number of dynamically executed spills. Their approach shares some similarities with our Trace Register Allocator. They also divide the control-flow graph in what they call *tiles* and (partially) solve the register allocation problem for each of them. The difference to our approach is that tiles are not independent but organized in a hierarchical tree. Allocation starts at the innermost tile (the leaf of the tree). Once all children are processed the parent tile uses the information of its children to continue. It is also worth noting that the algorithm does not assign registers in the first pass but uses pseudo registers to record the requirements of a tile. In a second top-down pass physical register are assigned and spill code is inserted if needed. Furthermore, Callahan and Koblenz use the graph coloring algorithm for allocating a tile.

The motivation for *Register Allocation based on Graph-Fusion* by Lueh et al. [45] is similar to Callahan and Koblenz, namely to minimize the number of dynamically executed spill instructions. Their allocator works on regions of the control-flow, where a region can be a basic block, a loop-nest, a superblocks [46], or other combinations of basic blocks. They build up the interference graph for each part and fuse together graphs of connected regions to build up the graph for the complete compilation unit. Their approach differs from ours as it is a global approach where decisions are made on a global basis. In contrast to that, we solve the problem locally and independently for each trace. Furthermore, the graph fusion approach uses the interference graph as the single model for liveness, whereas the Trace Register Allocator can use a different representation for every trace.

Koes and Goldstein proposed a *Global Progressive Register Allocator* [47] using multi-commodity network flow (MCNF). They first formulate a solution for the local problem (i.e. for a basic block) and later extend it to the global case. Later, Koes and Goldstein proposed an interesting extension where they used traces instead of basic blocks as the scope for the local problem, which further improved the quality and the speed of their approach [48]. To the best of our knowledge, they did not further investigate this idea and there are no scientific publications about it.

[5]Although, the implementation in Graal does not yet use the optimized lifetime analysis proposed in by Wimmer et al. [4].

Intermediate Representation.

An inspiration for our intermediate representation was the *Static Single Information* (SSI) form proposed by Ananian [49] and later refined by others [50, 51]. SSI is an extension of the SSA form that not only has φ functions at control-flow joins but also σ (in some work called π) functions at control-flow splits. Pereira and Palsberg exploited a variant of the SSI form in their register allocator based on *puzzle-solving.* Although we borrowed ideas from the SSI representation, our IR is strictly speaking not in (pruned) SSI form.

Lozano et al. introduced *Linear Static Single Assignment* (LSSA) form as an intermediate representation for their *Constraint-based Register Allocation and Instruction Selection* approach [53]. The *in- and out-delimiters* in LSSA form are similar to our representation. In our approach, however, they are not only used to mark lifetimes of variables but also to support inter-trace data-flow resolution.

7. CONCLUSION AND OUTLOOK

7.1 Future Work

In this paper we listed open issues such as spilling in a loop side-trace. Also, our current implementation stores the *live set* information for every block, although we only really need it for inter-trace edges. In the short term, our future work will focus on these issues.

Our long term research vision, based on Trace Register Allocation as presented in this paper, will focus on the following aspects: First, we want to experiment with using different algorithms for different traces. For example, a fast algorithm with good compilation speed might be preferable for unimportant traces at the cost of code quality. On the other hand, we plan to use algorithms that find the optimal or near to optimal solution for important traces. Second, Trace Register Allocation reduces the problem size by splitting it into distinct traces, which can be processed individually. Independent traces allow register allocation of one compilation unit in parallel, executed by multiple threads. Concurrent register allocation would potentially improve the throughput and compile-time significantly.

7.2 Summary

We presented an implementation of a novel register allocation approach, called Trace Register Allocation. Our results suggest that a global view on the register allocation problem is not strictly required to find solutions that are comparable to those of register allocators used in JIT compilers today.

Our non-global approach is based on traces, i.e., on linear segments of code. Each of these traces can be processed independently, possibly using different register allocation algorithms. The problem that has to be solved by these algorithms is simpler than the general problem of register allocation. Hence, trace-based register allocation algorithms can be simplified compared to their global counterparts.

Our results lay the foundation for future research in this new area of trace-based register allocation. We believe that the flexibility of our approach can push the boundaries of current register allocation techniques and can have an impact on both research and production compilers.

8. ACKNOWLEDGEMENTS

We thank all members of 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 contributions. Special thanks go to Gilles Duboscq, Stefan Marr, Tom Rodriguez, Roland Schatz and Christian Wimmer. The authors from Johannes Kepler University are funded in part by a research grant from Oracle.

9. REFERENCES

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

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

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

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

[5] Gregory J Chaitin et al. "Register Allocation via Coloring". In: *Computer languages* (1981). DOI: 10.1016/0096-0551(81)90048-5.

[6] Michael D. Smith, Norman Ramsey, and Glenn Holloway. "A generalized algorithm for graph-coloring register allocation". In: *SIGPLAN Not.* (2004). DOI: 10.1145/996893.996875.

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

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

[9] Thomas Kotzmann et al. "Design of the Java HotSpot™client compiler for Java 6". In: *TACO'08* (2008). DOI: 10.1145/1369396.1370017.

[10] Matthew Arnold et al. "Adaptive Optimization in the Jalapeño JVM". In: *SIGPLAN Not.* (2000). DOI: 10.1145/1988042.1988048.

[11] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *CGO'04*. IEEE Computer Society, 2004. DOI: 10.1109/CGO.2004.1281665.

[12] Josef Eisl. "Trace Register Allocation". In: *SPLASH Companion 2015*. ACM, 2015. DOI: 10.1145/2814189.2814199.

[13] Graal Project. OpenJDK Community. URL: http://openjdk.java.net/projects/graal/.

[14] Gilles Duboscq et al. "An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler". In: *VMIL'13* (2013). DOI: 10.1145/2542142.2542143.

[15] Lukas Stadler et al. "An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance". In: *SCALA'13*. ACM, 2013. DOI: 10.1145/2489837.2489846.

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

[17] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. "Partial Escape Analysis and Scalar Replacement for Java". In: *CGO '14*. ACM, 2014. DOI: 10.1145/2544137.2544157.

[18] Doug Simon et al. "Snippets: Taking the High Road to a Low Level". In: *TACO'15* (2015). DOI: 10.1145/2764907.

[19] Oracle Corporation. *Java SE HotSpot at a Glance.* URL: http://www.oracle.com/technetwork/articles/javase/index-jsp-136373.html (visited on 05/06/2016).

[20] Ron Cytron et al. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Trans. Program. Lang. Syst.* (1991). DOI: 10.1145/115372.115320.

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

[22] Michael Bebenita et al. "Trace-based compilation in execution environments without interpreters". In: *PPPJ'10* (2010). DOI: 10.1145/1852761.1852771.

[23] John R. Ellis. "Bulldog: A Compiler for VLIW Architectures". PhD thesis. Yale University, 1985.

[24] P. Geoffrey Lowney et al. "The Multiflow Trace Scheduling Compiler". In: *Journal of Supercomputing* (1993). DOI: 10.1007/BF01205182.

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

[26] *JEP 243: Java-Level JVM Compiler Interface.* 2014. URL: http://openjdk.java.net/jeps/243 (visited on 05/06/2016).

[27] DaCapo Project. *The DaCapo Benchmark Suite.* 2012. URL: http://dacapobench.org/ (visited on 05/26/2016).

[28] Andreas Sewe et al. "Da capo con scala". In: *OOPSLA'11* (2011). DOI: 10.1145/2048066.2048118.

[29] *SPECjvm2008: Java Virtual Machine Benchmark.* URL: https://www.spec.org/jvm2008/ (visited on 06/15/2015).

[30] *SPECjbb2015: Java Server Benchmark.* URL: https://www.spec.org/jbb2015/ (visited on 05/25/2016).

[31] Oracle Corporation. *Sun Server X3-2.* 2013. URL: http://www.oracle.com/us/products/servers-storage/servers/x86/sun-server-x3-2-ds-1683091.pdf (visited on 05/25/2016).

[32] *Linux Containers.* URL: http://linuxcontainers.org/ (visited on 05/26/2016).

[33] Oracle Corporation. *SPARC T5-2 Server*. 2013. URL: http://www.oracle.com/us/products/servers-storage/servers/sparc/oracle-sparc/t5-2/sparc-t5-2-ds-1922871.pdf (visited on 05/25/2016).

[34] Oracle Corporation. *Introduction to Oracle Solaris Zones*. 2014. URL: http://docs.oracle.com/cd/E36784_01/pdf/E36848.pdf (visited on 05/26/2016).

[35] John W. Tukey. *Exploratory data analysis*. Reading, Mass., 1977.

[36] John M. Chambers. *Statistical Models in S*. CRC Press, Inc., 1991.

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

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

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

[40] Andreas Gal et al. "Trace-based Just-in-Time Type Specialization for Dynamic Languages". In: *PLDI'09*. ACM, 2009. DOI: 10.1145/1542476.1542528.

[41] Carl Friedrich Bolz et al. "Tracing the meta-level". In: *ICOOOLPS'09* (2009). DOI: 10.1145/1565824.1565827.

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

[43] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. "Context-sensitive trace inlining for Java". In: *Computer Languages, Systems & Structures* (2013). DOI: 10.1016/j.cl.2013.04.002.

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

[45] Guei-Yuan Lueh, Thomas Gross, and Ali-Reza Adl-Tabatabai. "Global register allocation based on graph fusion". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1997. DOI: 10.1007/BFb0017257.

[46] Wen -Mei W. Hwu et al. "The superblock: An effective technique for VLIW and superscalar compilation". In: *The Journal of Supercomputing* (1993). DOI: 10.1007/bf01205185.

[47] David Ryan Koes and Seth Copen Goldstein. "A global progressive register allocator". In: *PLDI'06* (2006). DOI: 10.1145/1133981.1134006.

[48] David Ryan Koes and Seth Copen Goldstein. *A better global progressive register allocator*. 2006. URL: http://www.cs.cmu.edu/~dkoes/research/lctes06_tracealloc.pdf.

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

[50] Jeremy Singer. *Static program analysis based on virtual register renaming*. Tech. rep. University of Cambridge, 2006. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-660.pdf.

[51] B. Boissinot et al. *SSI Properties Revisited*. Tech. rep. ENS-Lyon, 2009. URL: https://hal.inria.fr/inria-00404236/file/bboissin-ssi-RR.pdf.

[52] Fernando Magno Quintão Pereira and Jens Palsberg. "Register allocation by puzzle solving". In: *PLDI'08* (2008). DOI: 10.1145/1375581.1375609.

[53] Roberto Castañeda Lozano et al. "Constraint-Based Register Allocation and Instruction Scheduling". In: *CP'12* (2012). DOI: 10.1007/978-3-642-33558-7_54.