Divide and Allocate: The Trace Register Allocation Framework

ACM Student Research Competition Grand Finals

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

ABSTRACT

Register allocation is a mandatory task for almost every compiler and consumes a significant portion of compile time. In a just-in-time compiler, compile time is a particular issue because compilation happens during program execution and contributes to the overall application run time. Compilers often use global register allocation approaches, such as graph coloring or linear scan, which only have limited potential for improving compile time since they process a whole method at once. We developed a novel *trace register allocation* framework which competes with global approaches in both compile time and code quality. Instead of processing a whole method, our allocator processes linear code segments (traces) independently and is therefore able to (1) select different allocation strategies based on the characteristics of a trace to control the trade-off between compile time and peak performance, and (2) to allocate traces in parallel to reduce compilation latency, i.e., the duration until the result of a compilation is available.

CCS CONCEPTS

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

KEYWORDS

trace register allocation, parallel register allocation, trace compilation, linear scan, just-in-time compilation, dynamic compilation, virtual machines

ACM Reference Format:
Josef Eisl. 2018. Divide and Allocate: The Trace Register Allocation Framework: ACM Student Research Competition Grand Finals. In *Proceedings of ACM Student Research Competition Grand Finals (ACM SRC Grand Finals).* ACM, New York, NY, USA, 5 pages.

1 MOTIVATION

Most optimizing compilers use *global* register allocation approaches, such as *graph coloring*[1] or *linear scan*,[2] which process a whole method at once. Compiler optimizations, such as *inlining* or *code duplication*,[3] cause methods to become large. This poses two problems:

- Register allocation time increases with method complexity, often in a non-linear fashion [Poletto et al., 1999].
- Different regions contribute differently to the overall performance of the compiled code [Bala et al., 2000].

Global allocators do not differentiate between *important* and *unimportant* parts of a method, or do so only in a limited way. Following our example in Figure 1, they spend the same amount of time for the *cold* blocks (B4–B7) as for the *hot trace* (B1–B3 and B8). However, we want to spend our time budget more wisely, for example spending 80% on the most frequent case, and only 20% for the rest.

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

We proposed a novel *non-global* register allocation approach, called *trace register allocation* [Eisl, 2015; Eisl et al., 2016, 2017, 2018], that tackles both issues of traditionally global allocators, namely to control the *compile time* on a fine-granular level, and to reduce *compilation latency* using parallelization.

2 TRACE REGISTER ALLOCATION

In contrast to *global register allocation*, which processes a whole method at once, *trace register allocation* divides the problem into smaller sub-problems, so-called *traces*, for which register allocation can be done independently for each trace. Figure 2 shows the components of the trace register allocation framework.

2.1 Trace Building

The trace building algorithm takes the basic blocks of a control flow graph as its input and returns a set of traces. A *trace* is a list of sequentially executed blocks [Lowney et al., 1993]. Traces are non-empty and non-overlapping, and every basic block is contained in exactly one trace. The algorithm uses profiling information provided by the virtual machine to construct long and important traces first. Figure 1b illustrates the result of the trace building step for the code snippet in Figure 1a.

*Advisor: Hanspeter Mössenböck, Johannes Kepler University Linz

[1] Graph coloring [Chaitin et al., 1981; Briggs et al., 1994; George et al., 1996] is used in GCC [2017], WebKit [2017a] or the HotSpot server compiler [Paleczny et al., 2001]

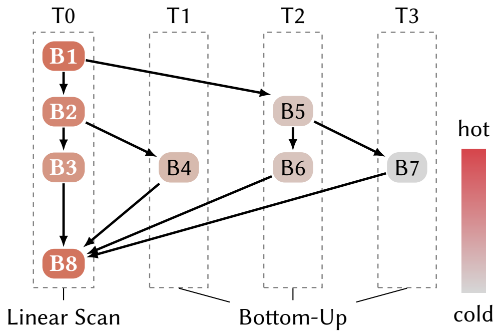
[2] Linear scan [Poletto et al., 1999; Traub et al., 1998; Wimmer et al., 2005] is used in WebKit [2017b] or the HotSpot client compiler [Kotzmann et al., 2008]

[3] See for example Prokopec et al. [2017] or Leopoldseder et al. [2018]

ACM SRC Grand Finals, June 2018, San Francisico, CA, USA
2018.

```
Result accessArray(Object[] o, int i, boolean shouldNPE) {
                Result r = new Result();
    /* B1 */    if (o != null) {
    /* B2 */        if (i >= 0 && i < o.length)
    /* B3 */            r.val = o[i];
                    else
    /* B4 */            r.ex = idxOutOfBnds(o, i);
                } else {
    /* B5 */        if (shouldNPE)
    /* B6 */            r.ex = npe();
                    else
    /* B7 */            r = null;
                }
    /* B8 */    return r;
            }
```

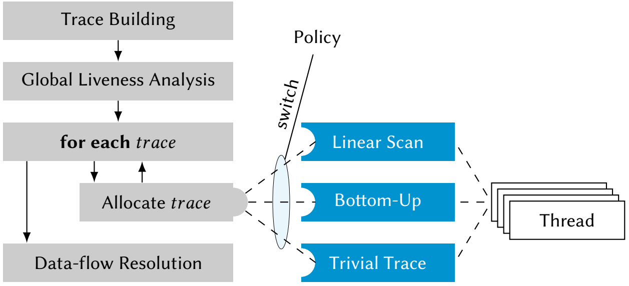
(a) Java source code



(b) Control-flow graph divided into traces

The source code and control-flow graph for an `accessArray` snippet, which might be found in an interpreter. Red blocks are *frequently executed* (hot), gray blocks are *less important* (cold). The path through the `normalAccess` branch (B3) is the common case, and should be optimized. The blocks are partitioned into traces (T1–T4); registers are allocated per trace using different *strategies* (Linear Scan or Bottom-Up) based on their probability.

Figure 1: Trace Register Allocation—A Motivating Example



Left (gray): Phases that are only executed once per method. Top: A *policy* decides which strategy should be used. Middle (blue): *Allocation strategies* that are used for processing a single trace. Right (white): A *thread pool* to allocate traces in parallel.

Figure 2: Overview of our framework

2.2 Global Liveness Analysis

To capture the liveness of variables at trace boundaries, a global liveness analysis is required. For every inter-trace edge, $live_{out}$ and $live_{in}$ sets are computed. We need these sets to (1) decouple traces and make independent register allocation possible and (2) make the allocation result available for subsequent phases.

The analysis is done in a single iteration over the basic blocks in reverse post-order, similar to the liveness analysis described by Wimmer et al. [2010] for SSA-based[4] linear scan register allocation.

2.3 Allocate Traces

For each trace, the algorithm selects an *allocation strategy*. Due to the explicit global liveness information, allocating a trace is completely decoupled from the allocation of other traces. The linear structure of traces makes the implementation of strategies significantly simpler compared to a global algorithm. We implemented three allocation strategies:

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

Bottom-Up In order to decrease compilation time, we implemented the *bottom-up* allocator [Eisl et al., 2017]. Its goal is to allocate a trace as fast as possible, potentially sacrificing code quality.

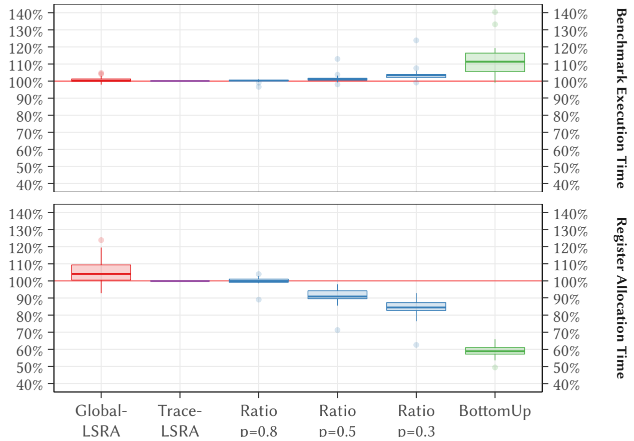
Trivial Trace The *trivial trace* allocator is a special-purpose allocator for traces which have a specific structure. They consist of a single basic block which contains only a single *jump* instruction. Such blocks are introduced by splitting *critical edges*, and are quite common. For the DaCapo benchmark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

Traces can be processed in arbitrary order. However, traces that are processed later can profit from decisions in already processed traces. We implemented two optimizations based on that principle, *inter-trace hinting* and *spill information sharing* [Eisl et al., 2016]. They reduce the number of resolution and spill moves and thus improve the quality of the allocation.

2.4 Data-flow Resolution

After allocating registers for traces, the location of a variable can be different across an inter-trace edge. The data-flow resolution phase fixes up those mismatches and ensures a valid solution. It

[4] Static Single Assignment from, see Cytron et al. [1991]



DaCapo and Scala-DaCapo on AMD64. Results normalized to TraceLSRA configuration. For more details see our previous work [Eisl et al., 2017].

Figure 3: Allocation Policy Results (lower is better)

also replaces the φ-functions of the SSA form with move instructions [Sreedhar et al., 1999]. This is similar to the resolution pass in linear scan allocators with interval-splitting, for example in those by Traub et al. [1998] or Wimmer et al. [2005].

3 RESULTS

To validate our approach, we need to answer the following questions positively. Can the *trace-based* approach

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

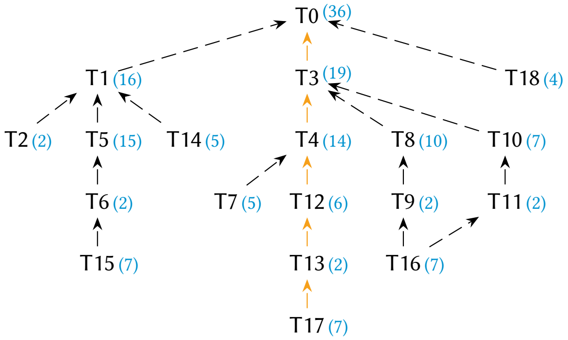
RQ2: be *as fast as* a global allocator for the same quality of allocation? [Eisl et al., 2017]

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

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

To answer these questions, we implemented our approach in GraalVM.[5] GraalVM is a Java virtual machine based on the HotSpot VM [Paleczny et al., 2001; Kotzmann et al., 2008]. The Graal compiler produces code that is *on par* with HotSpot, or is even better [Stadler et al., 2013; Simon et al., 2015; Prokopec et al., 2017]. Using our implementation, we evaluated the trace register allocation approach using standard benchmarks, including DaCapo [Blackburn et al., 2006], Scala-DaCapo [Sewe et al., 2011], SPECjvm2008,[6] and SPECjbb2015.[7]

Figure 3 depicts our results for the DaCapo and Scala-DaCapo benchmarks on AMD64. The TraceLSRA configuration uses the *linear scan* and the *trivial trace* strategy. The results show that it is *on par* with the global linear scan algorithm (GlobalLSRA) regarding *benchmark execution time* (code quality) and *register allocation time*. Therefore, we can answer research questions RQ1 and RQ2 positively. For results on SPECjvm2008 and SPECjbb2015, as well as for the SPARC architecture, see our previous work [Eisl et al., 2016].



The values in parenthesis are the length of the traces, i.e., the number of instructions. The critical path (T0–T17, orange edges) is 84 instructions long.

Figure 4: Trace Dependency Graph for `PrintStream.write()`

3.1 Trace Register Allocation Policies

The flexibility of trace register allocation allows switching the allocation algorithm on a per-trace basis. We exploit this to answer RQ3, i.e., to get fine-grained control over compile-time vs. peak-performance. The idea is use the *linear scan* strategy for *important* traces, and the faster *bottom-up* strategy for the others.

We conducted a case study of 8 decision heuristics, so-called *allocation policies*, based on properties of a trace like execution probability or the number of variables [Eisl et al., 2017]. In Figure 3, we show the results for the Ratio policy, which orders the traces based on their *probability* and allocates the first p% with the linear scan strategy. The other traces are allocated with the bottom-up approach. The results show that, for example, a parameter of $p = 0.3$ reduces the register allocation time by almost 20% with a performance degradation of about 4%. For reference, we also show the BottomUp policy, which only uses the *bottom-up* strategy. It saves 40% allocation time at a performance penalty of 12% on average.

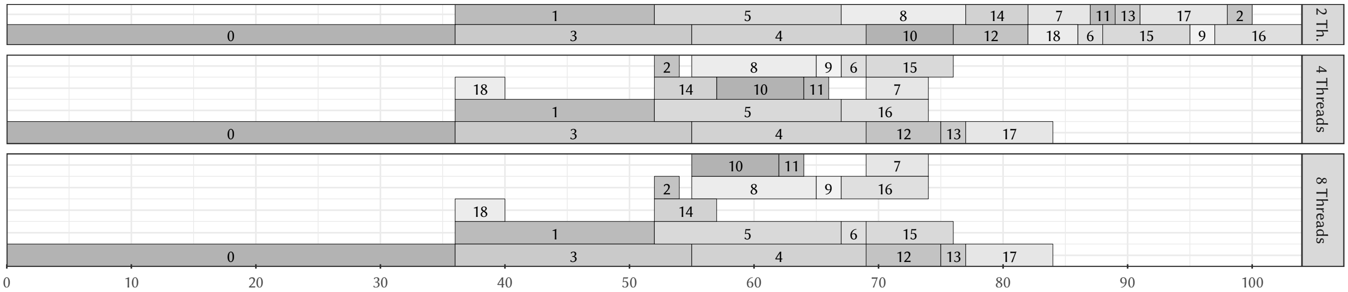
3.2 Parallel Trace Register Allocation

To answer RQ4, we are currently investigating the *parallelization potential* of trace register allocation to reduce *compilation latency*.[8] The idea is to allocate traces concurrently on multiple threads. For this experiment, we want to avoid regressions in the quality of allocation. More precisely, we need to keep the order in which *connected* traces are allocated intact, so that *inter-trace hinting* and *spill information sharing* is not affected. This requirement defines a dependency relation for traces and thus limits the parallelization potential.

Figure 4 shows the dependency graph of the `PrintStream.write()` method from the Java standard library. We visualize the calculated schedules for 2, 4 and 8 threads in Figure 5. The length of the schedule with 2 threads is 104 instructions. With 4 threads, the length of 84 instructions is already optimal, i.e., the *critical path* length.

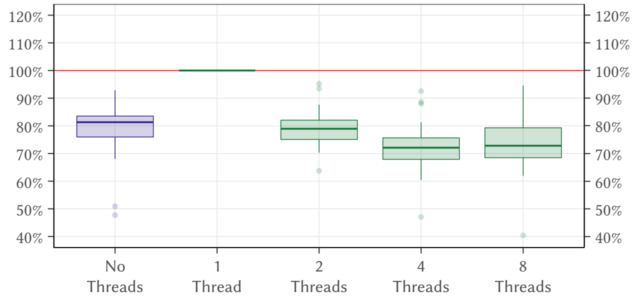
[5] https://github.com/oracle/graal
[6] https://www.spec.org/jvm2008/
[7] https://www.spec.org/jbb2015/

[8] A work-in-progress report is currently under review [Eisl et al., 2018].



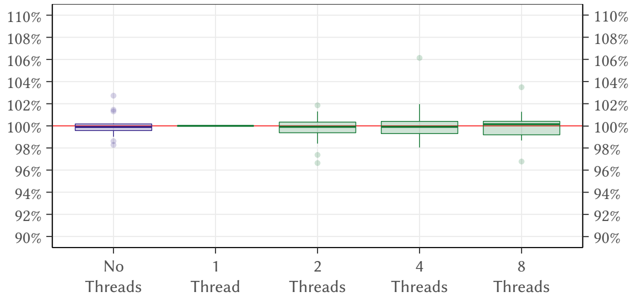
Gantt chart [Gantt, 1913] for 2, 4 and 8 threads. The horizontal axis denote the number of instructions.

Figure 5: Trace Allocation Scheduling for `PrintStream.write()`



DaCapo and Scala-DaCapo on AMD64. Values relative to 1 register allocation thread mean.

Figure 6: Parallel Register Allocation Time (lower is better)



Performance results for the Figure 7.

Figure 7: Benchmark Execution Time (lower is better)

Increasing the number of threads to 8 cannot yield any improvements. Also note that only 5 threads are used, although 8 would be available.

To empirically verify our approach, we added a simple *proof-of-concept* implementation. We used a *thread pool* and a *priority queue* where traces are ordered by decreasing instruction count, so that longer traces are allocated first. From the register allocation point of view, synchronization is only needed for the queue. Accessing and modifying traces is safe by design of the trace register allocator if the dependencies are respected. However, Graal assumes that every method is compiled by just a single thread. We worked around this assumption, for example, by pre-populating cached maps, duplicating data structures or simply recalculating results. These workarounds cause allocation time overheads which we are willing to accept for our prototype.

The experimental results are summarized in Figure 6. Note that the reported values are the duration of register allocation and include all necessary phases. In addition to the allocation, it includes *trace building*, *global liveness analysis* and *global data-flow resolution*. All numbers are relative to the 1 Thread configuration. This configuration uses the priority queue and the other synchronization mechanisms, but the thread pool only consists of a single thread. To see the overhead imposed by our prototype, we also compare against the No Threads configuration, i.e., the trace register allocation where all work is done on the compiler thread. We are confident that the overhead of 23% can be reduced by a more sophisticated implementation. To verify that we can reduce allocation latency, we evaluated the prototype with 2, 4, and 8 allocation threads.

Using 2 threads instead of one decreases the latency by 21%. Increasing the thread count to 4 reduces the allocation time by 28%, compared to a single thread. Using 8 threads does not give any advantages. This is due to the restricting dependencies that limit the parallelization potential.

We also a verified that parallel register allocation does not affect allocation quality. The results in Figure 7 suggest that this is indeed the case.

4 CONCLUSION

We presented the trace register allocation framework, a novel, flexible, non-global and extensible register allocation approach, which eliminates short-comings of global allocators. In its basic configuration it exhibits similar compile time and peak performance results as a state-of-the-art global allocator. However, trace register allocation can reduce the *compile time* by providing fine-grained control over the allocation-time vs. peak-performance trade-offs. Our experiments show a reduction of register allocation time of up to 40%. In addition, due to its non-global principle, trace register allocation offers opportunities for parallelization which reduces *compilation latency*. Our initial prototype can reduce the latency of register allocation by 28% when using four threads instead of a single allocation thread.

There is more to come. In our work, we highlighted two features of the trace register allocation approach. We can think of more extensions, for example combining policies with parallelization.

Another idea is to further optimize a set of important traces, potentially in multiple phases. *Decoupled register allocation* comes to mind, as for instance proposed by Colombet et al. [2011] or Barik et al. [2013]. A decoupled register allocator first reduces the register pressure by spilling variables before the second phase performs the actual (spill-free) allocation. Combining this approach with trace register allocation allows improving the allocation quality while keeping the compile time overhead low.

The idea does not end with register allocation. In fact, similar ideas where already applied to instruction scheduling [Lowney et al., 1993] decades ago. We are confident that other optimizations can profit as well from the approaches we developed for trace register allocation.

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

ACKNOWLEDGMENTS

We thank the Graal community, the Virtual Machine Research Group at Oracle Labs and the Institute for System Software at the Johannes Kepler University Linz for their support and feedback on this work. Josef Eisl is funded in part by a research grant from Oracle Labs.

REFERENCES

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

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

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

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

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.

Colombet, Quentin, Benoit Boissinot, Philip Brisk, Sebastian Hack, and Fabrice Rastello (2011). Graph-coloring and Treescan Register Allocation Using Repairing. In: *CASES'11*. ACM. DOI: 10.1145/2038698.2038708.

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

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

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

Eisl, Josef, David Leopoldseder, and Hanspeter Mössenböck (2018). Parallel Trace Register Allocation. In: (under review)

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

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

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

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

Kotzmann, Thomas, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox (2008). Design of the Java HotSpot™ client compiler for Java 6. In: TACO'08. DOI: 10.1145/1369396.1370017.

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

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/events/jvm01/full_papers/paleczny/paleczny.pdf.

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

Prokopec, Aleksandar, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger (2017). Making Collection Operations Optimal with Aggressive JIT Compilation. In: *SCALA 2017*. ACM. DOI: 10.1145/3136000.3136002.

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

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

Sreedhar, Vugranam C., Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam (1999). "Translating Out of Static Single Assignment Form". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. DOI: 10.1007/3-540-48294-6_13.

Stadler, Lukas, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon (2013). An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance. In: *SCALA'13*. ACM. DOI: 10.1145/2489837.2489846.

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.

WebKit (2017a). *Graph Coloring Register Allocator in WebKit*. URL: https://github.com/WebKit/webkit/blob/5277f6fb92b0/Source/JavaScriptCore/b3/air/AirAllocateRegistersByGraphColoring.h.

— (2017b). *Linear Scan Register Allcoator in WebKit*. URL: https://github.com/WebKit/webkit/blob/5277f6fb92b0/Source/JavaScriptCore/b3/air/AirAllocateRegistersAndStackByLinearScan.h.

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.