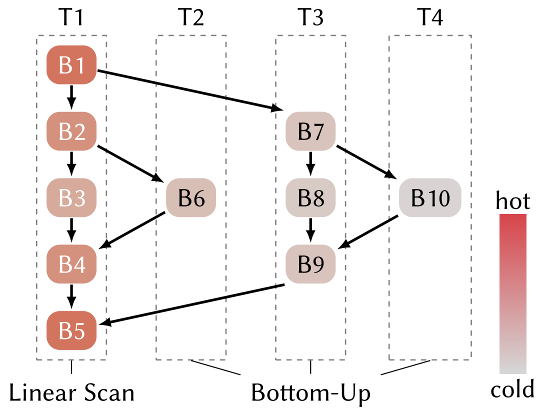
Divide and Allocate: The Trace Register Allocation Framework[*]

Extended Abstract

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

```
void accessArray(Object o, int i) {
  /* B1 */  if (o != null) {
  /* B2 */    if (i >= 0 && i < len(o)) {
  /* B3 */      normalAccess(o, i);
              } else {
  /* B6 */      indexOutOfBoundsEx(o, i);
              }
  /* B4 */  } else {
  /* B7 */    if (SHOULD_DEOPT) {
  /* B8 */      toInterpreter(o, i);
              } else {
  /* B10 */     nullPointerEx();
              }
  /* B9 */  }
  /* B5 */}
```

(a) Java source code for arrayAccess



(b) Control-flow graph divided into traces

The source code and control-flow graph for an `accessArray` *snippet. Red blocks are* frequently executed *(hot), gray blocks are* less important *(cold). The path through the* `normalAccess` *branch (B3) is the common case. 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. A Motivating Example

Abstract

Compilers often use global register allocation approaches such as linear scan or graph coloring. The flexibility of these approaches is limited 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 the whole method, our allocator processes linear code segments (traces) independently and is therefore able 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.

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

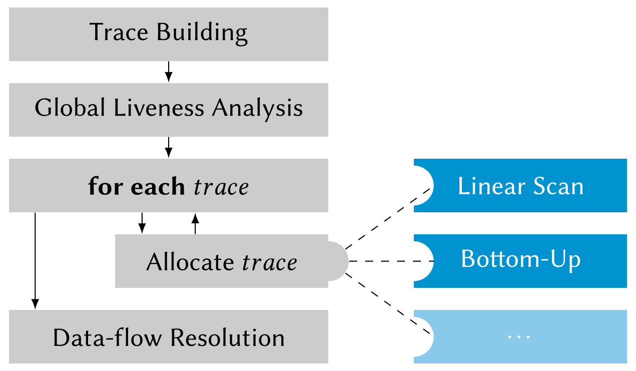
Keywords trace register allocation, trace compilation, linear scan, just-in-time compilation, dynamic compilation, virtual machines

ACM Reference Format:
Josef Eisl. 2018. Divide and Allocate: The Trace Register Allocation Framework: Extended Abstract. In *Proceedings of International Symposium on Code Generation and Optimization Student Research Competition (CGO'18 SRC)*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Motivation

When looking at the example in Figure 1, we see that not all parts of the method are equally important. Most optimizing

[*]This research project is partially funded by Oracle Labs.
[†]Advisor: Hanspeter Mössenböck, Johannes Kepler University Linz



Left (gray): Phases that are only executed once per method. Right (blue): Allocation strategies *that are used for processing a single trace.*

Figure 2. Overview of our framework

compilers use *global* register allocation [3, 4, 8, 9, 13–15, 22–25], i.e., they process a whole method at once. Compiler optimizations, such as *inlining* or *code duplication* [11, 16], cause methods to become large. This poses two problems:

- Register allocation time increases with method complexity, often in a non-linear fashion [15].
- Different regions contribute differently to the overall performance of the compiled code [1].

We assume that most time is spent in a small portion of the method [1]. Global allocators do not differentiate between *important* and *unimportant* parts, or only in a limited way.

2 Idea

We solved the problems with a *non-global* approach based on *traces*, i.e., a sequence of sequentially executed blocks [12]. Traces are constructed using profiling feedback (Figure 1b). They are allocated independently, potentially using different *strategies*. We use strategies that yield good *code quality* for *important traces* and *fast* strategies for the others.

Figure 2 shows the components of our framework [6, 7].

Trace Building partitions the blocks of the control-flow graph into traces (Figure 1b).

Global Liveness Analysis captures the liveness of variables at trace boundaries.

Allocate Traces: For each trace, we select the most suitable register *allocation strategy*, i.e.:

- *Linear Scan* for *high-quality code*
- *Bottom-Up* for *fast allocation*

Due to the linear structure of traces, strategies are significantly simpler compared to a global algorithm.

Data-flow Resolution is required since the locations of variables might be different across an inter-trace edge.

3 Results

To validate our approach, we need to answer the following questions:

- Can a trace-base approach achieve peak performance similar to that of a global approach [6]?
- Can we improve compile time and/or peak performance by switching allocation strategies within a method [7]?

We implemented our approach in GraalVM [5, 10, 16, 18, 21] and evaluated it using standard benchmarks, including DaCapo [2, 17], SPECjvm2008 [20] and SPECjbb2015 [19].

Figure 3 depicts our results. It shows that our approach can compete with a global allocator. In addition, the flexibility allows us to save up to 40% allocation time.

4 Conclusion

We presented the trace register allocation framework, a novel, flexible, non-global and extensible register allocation approach. It eliminates the limitations of global allocators while exhibiting similar or better compile time and peak performance results.

References

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

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

[3] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. In: TOPLAS'94 (1994). ISSN: 0164-0925. DOI: 10.1145/177492.177575.

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

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

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

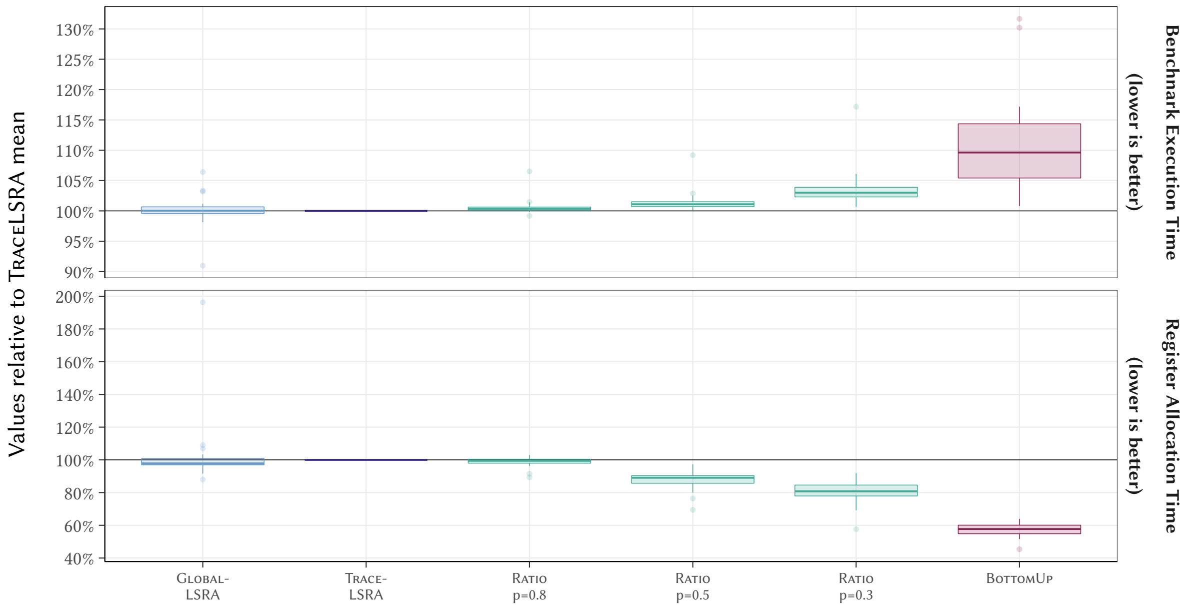
[7] Josef Eisl, Stefan Marr, Thomas Würthinger, and Hanspeter Mössenböck. Trace Register Allocation Policies: Compile-time vs. Performance Trade-offs. In: *ManLang 2017*. 2017. DOI: 10.1145/3132190.3132209.

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

[9] Lal George and Andrew W. Appel. Iterated register coalescing. In: TOPLAS'96 (1996). ISSN: 0164-0925. DOI: 10.1145/229542.229546.

[10] Graal Authors. *Graal Compiler & Truffle Partial evaluator*. 2016. URL: https://github.com/graalvm/graal-core (visited on 05/06/2016).

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



The TraceLSRA configuration (baseline) uses only the linear scan algorithm. The results show that it can compete with the global linear scan algorithm (GlobalLSRA) in both compile time and code quality. In the bottom-up only configuration (BottomUp) we reduce register allocation time by 40% by only using the bottom-up strategy. We also show results for mixed policies (Ratio): $p = 0.3$ means that we use linear scan for 30% of the traces and the bottom-up strategy for the others. The results illustrate the flexibility of our approach. For more details see our previous work [7].

Figure 3. Benchmark results for (Scala) DaCapo on AMD64

[12] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'donnell, and John C. Ruttenberg. The Multiflow Trace Scheduling Compiler. In: Journal of Supercomputing (1993). DOI: 10.1007/BF01205182.

[13] OpenJDK. *Chaitin Allocator in C2.* 2017. URL: http://hg.openjdk.java.net/jdk/hs/file/5caa1d5f74c1/src/hotspot/share/opto/chaitin.hpp.

[14] OpenJDK. *Linear Scan Register Allocator in C1.* 2017. URL: http://hg.openjdk.java.net/jdk/hs/file/5caa1d5f74c1/src/hotspot/share/c1/c1_LinearScan.hpp.

[15] Massimiliano Poletto and Vivek Sarkar. Linear Scan Register Allocation. In: TOPLAS'99 (1999). ISSN: 0164-0925. DOI: 10.1145/330249.330250.

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

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

[18] Doug Simon, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, and Thomas Würthinger. Snippets: Taking the High Road to a Low Level. In: TACO'15 (2015). ISSN: 1544-3566. DOI: 10.1145/2764907.

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

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

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

[22] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and Speed in Linear-scan Register Allocation. In: *PLDI '98.* ACM, 1998. DOI: 10.1145/277650.277714.

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

[24] WebKit. *Linear Scan Register Allcoator in WebKit.* 2017. URL: https://github.com/WebKit/webkit/blob/5277f6fb92b0/Source/JavaScriptCore/b3/air/AirAllocateRegistersAndStackByLinearScan.h.

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