Parallel Trace Register Allocation[*]

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

David Leopoldseder
Institute for System Software
Johannes Kepler University Linz
Austria
david.leopoldseder@jku.at

Hanspeter Mössenböck
Institute for System Software
Johannes Kepler University Linz
Austria
hanspeter.moessenboeck@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. Parallelization can help here. We developed a theoretical model for parallel register allocation and show that it can be used in practice without a negative impact on the quality of the allocation result. Doing so reduces compilation latency, i.e., the duration until the result of a compilation is available.

Our analysis shows that parallelization can theoretically decrease allocation latency by almost 50%. We implemented an initial prototype which reduces the register allocation latency by 28% when using four threads, compared to the single-threaded allocation.

CCS CONCEPTS

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

KEYWORDS

trace register allocation, concurrent register allocation, parallel register allocation, just-in-time compilation, dynamic compilation, virtual machines, task scheduling

ACM Reference Format:
Josef Eisl, David Leopoldseder, and Hanspeter Mössenböck. 2018. Parallel Trace Register Allocation. In *15th International Conference on Managed Languages & Runtimes (ManLang'18), September 12–14, 2018, Linz, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3237009.3237010

1 INTRODUCTION

Compilation latency, i.e., the duration required to compile a given method, is an important metric of just-in-time compilers. If compilation happens on the main thread, latency has a significant impact on response time, since the execution of the application is delayed. Even for systems with one or more background compilation threads [Krintz et al., 2001], short latencies mean that the compiled code is available earlier and can therefore reduce the warm-up time of an application. In contrast to most other optimizations, register allocation, i.e., the task of mapping a potentially unlimited number of variables to a fixed set of physical registers, is mandatory for all compilers that target a register machine, which is the predominant architecture today. The quality of the register allocation has a high impact on the performance of the generated machine code [Hennessy and Patterson, 2003], even on modern architectures [Eisl et al., 2017]. Thus, most optimizing compilers spend a significant amount of time on register allocation.

Traditionally, high quality register allocation is done on a global scope, i.e., on the whole compilation unit (method) at once. Graph coloring [Chaitin et al., 1981] or linear scan [Poletto and Sarkar, 1999] are common global approaches. In contrast to that, Eisl [2015] proposed *trace register allocation* as a *non-global* alternative. The idea is to divide a control-flow graph into linear code segments, so-called *traces*, and to solve the register allocation problem for those independently. Later, the intermediate results are merge to build a solution for the whole compilation unit. Recent results suggest that the approach is *on par* with *state-of-the-art* global approaches in terms of allocation quality [Eisl et al., 2016] as well as compile time [Eisl et al., 2017].

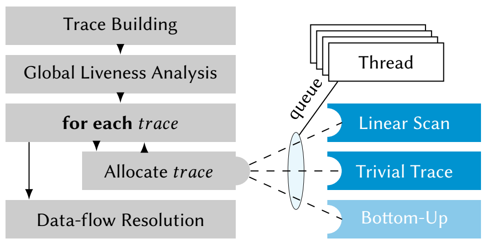
Since traces are allocated independently, this approach offers more flexibility than global allocators. Eisl et al. [2017] selected different allocation strategies which either yield good allocation quality or fast allocation time, based on the expected trace execution frequency. They evaluated different policies which reduced allocation time by up to 43% on average. However, this reduction comes at a peak performance penalty of about 11%.

In this work we leverage the flexibility of trace register allocation to reduce compilation latency without impacting peak performance. To do so, we use multiple threads to allocate traces concurrently. We built a prototype based on the idea and were able to reduce the register allocation latency by 28%. In summary, we contribute the following:

- A trace dependency model for minimizing synchronization effort and avoiding a negative performance impact due to parallelization.
- An analysis of the parallelization potential in trace register allocation.
- A *proof-of-concept* implementation of a parallel trace register allocator.
- An empirical evaluation of our prototype using the DaCapo [Blackburn et al., 2006] and Scala-DaCapo [Sewe et al., 2011] benchmarks.

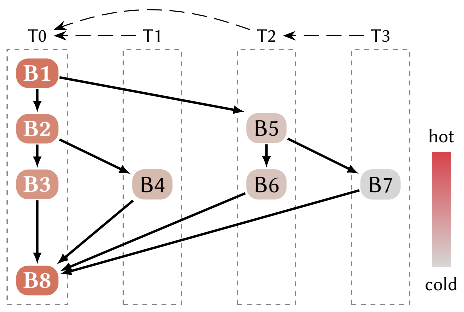
In the remainder of the paper we will recap the trace register allocation approach, describe how we can allocate registers concurrently with minimal synchronization effort, show that there is potential

[*]Work-in-Progress Research Paper



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

Figure 1: Overview of our framework



Control-flow graph (blocks and solid edges), trace (dashed boxes) and trace dependency graph (dashed arrows).

Figure 2: Control-flow graph divided into traces

for parallelization and present preliminary results of our implementation prototyped in GraalVM, a production quality Java virtual machine developed by Oracle.[1]

2 BACKGROUND ON TRACE REGISTER ALLOCATION

Our idea extends previous work on trace register allocation by Eisl et al. [2016, 2017]. In this section we review trace register allocation before we propose our extensions in Section 3.

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 1 shows the components of the trace register allocation framework.

Trace Building. The trace building algorithm takes the basic blocks of a control flow graph as its input and returns a set of traces. Traces are non-empty and non-overlapping and every basic block is contained in exactly one trace. Figure 2 illustrates the result of the trace building step.

Global Liveness Analysis. To capture the liveness of variables at trace boundaries, a global liveness analysis is required. For every inter-trace edge, i.e., an edge that connects two traces, we compute a *live* set. The analysis is done in a single iteration over the basic blocks, similar to the liveness analysis described by Wimmer and Franz [2010] for SSA-based linear scan register allocation.

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. Eisl et al. [2017] proposed 3 allocation strategies:

The *trace-based linear scan algorithm* is an adaption of the global approach by Wimmer and Mössenböck [2005] to the properties of a trace. The main difference of the trace-based approach is that there is no need to maintain a list of live ranges for each lifetime interval, since there are no *lifetime holes* [Traub et al., 1998] in trace intervals. Trace-based linear scan register allocation produces code that is on average as fast as code produced by global linear scan register allocation [Eisl et al., 2016].

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*,[2] 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. Eisl et al. [2017] showed that using the *trace-based linear scan* algorithm in combination with the *trivial trace* allocator is as fast as a state-of-the-art *global linear scan* approach [Wimmer and Mössenböck, 2005] in terms of allocation time.

In order to decrease compilation time, Eisl et al. [2017] proposed using a third allocation strategy, the *bottom-up* allocator. Its goal is to allocate a trace as fast as possible, potentially sacrificing code quality. Eisl et al. [2017] reported that the compile time of the bottom-up allocator is up to 40% faster than for the trace-based linear scan algorithm. However, this comes with an 11% slowdown in peak performance on average. We do not want to sacrifice peak performance for this work, thus we do not use the bottom-up allocation strategy.

Data-flow Resolution. 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.

3 PARALLEL TRACE REGISTER ALLOCATION

Traces can be allocated in arbitrary order. However, traces that are processed later can incorporate decisions from already allocated traces. Eisl et al. [2016] described two optimizations based on that principle, *inter-trace hinting* and *spill information sharing*. *Inter-trace hints* are *suggestions* to the allocator to favor a certain location for a variable. The aim is to decrease the number of resolution moves on inter-trace edges. *Spill information sharing* records if a variable that is currently stored in a register has a copy on the stack. That means it has already been spilled in a predecessor trace. Due to static-single assignment (SSA) form [Cytron et al., 1991], the value stored in a variable never changes statically. Therefore, if it has been spilled to the stack it will stay the same during the lifetime of the variable. It is never necessary to respill it. These optimizations

[1]https://github.com/oracle/graal

[2]An edge is *critical* if the source block has multiple successors and the target block has multiple predecessors.

can improve the peak performance for some DaCapo benchmarks by up to 15% [Eisl et al., 2016].

The information the allocator requires for both optimizations is stored together with the *live* sets and is local to an inter-trace edge. Although the optimizations are fully optional, the quality of the result depends on the order in which traces are allocated. Therefore, traces are allocated in the order of their *execution frequency*.

3.1 Dependency Model

For this study, we want to ensure that concurrent allocation does not influence allocation quality. Therefore, we define dependencies in a way that all the information available in the serial mode is also available in the parallel mode. The dashed arrows in Figure 2 show the dependency graph. Traces are numbered in the order of their expected execution frequency. In serial mode, they would be allocated in ascending order, i.e., T0, T1, T2, T3. However, since T1 and T2 are independent of each other, they can be allocated in parallel. The dependencies form a partial ordering of the traces.

4 CONCURRENCY POTENTIAL

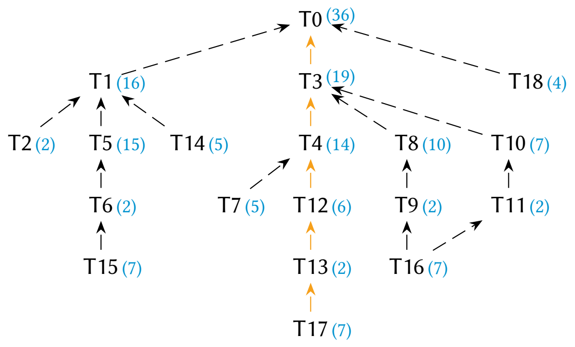
Before considering implementation details, we want to gain more insight whether there is potential for parallelization. Therefore, we simulated the potential improvements when using 2, 4 and 8 threads for register allocation of the benchmarks from DaCapo and Scala-DaCapo, two commonly used Java virtual machine benchmark suites based on real-world applications [Blackburn et al., 2006; Sewe et al., 2011].

Not all traces require the same time for allocation. We use the *number of instructions* as a compile time estimator since it correlates with the time required for register allocation [Eisl et al., 2017]. The lower bound for register allocation latency is the length of the *critical path* in the dependency graph. Our experiments show that the geometric mean [Fleming and Wallace, 1986] of the critical path length is 51% ($min = 44\%$, $max = 57\%$) of the number of instructions in the compilation unit. That means that ideally, with infinite threads and ignoring all overheads, the register allocation step could be done in about half the time.

To simulate the potential with a given number of threads, we need to find a schedule that satisfies the dependencies. Finding an optimal schedule with minimal duration is NP-hard [Pinedo, 2016]. Therefore, we apply a simple heuristic for finding a schedule. Whenever a thread is idle, we select the *longest* trace in terms of instructions.

For 2 threads, the simulated allocation latency goes down to 68% (64%, 71%). This is already an interesting result since adding only a single thread to the system can potentially improve register allocation latency by about 30%. Another noteworthy metric is the *utilization* of the threads, that is the ratio between *work* and *idle* time. For 2 threads this ratio is 74% (70%, 78%), which means that allocation threads are idle only one fourth of their run time.

If 4 threads are used, the simulated allocation latency is 56% (50%, 61%) of the single threaded case. However, the utilization also decreases to 45% (41%, 50%). The threads are idle more than half of the time.



Dependency graph for the method `PrintStream#write`. The values in parentheses are the lengths of the traces, i.e., the number of instructions. The critical path (T0–T17, orange edges) is 84 instructions long.

Figure 3: Trace Dependency Graph

With 8 allocation threads the allocation latency is 52% (45%, 58%) and is almost on the level of the optimal critical path case (51%). As expected, the utilization further decreases to 24% (22%, 28%).

4.1 Example

Let us illustrate the simulation approach with an example. We chose the method `PrintStream#write`[3] from the Java standard library. After high-level optimizations (e.g. inlining) there are 19 traces. Their dependencies are depicted in Figure 3. The values in parentheses are the trace lengths in *number of instructions*. The compilation unit consists of 168 instructions in total, which is also the *length* of the single threaded schedule.

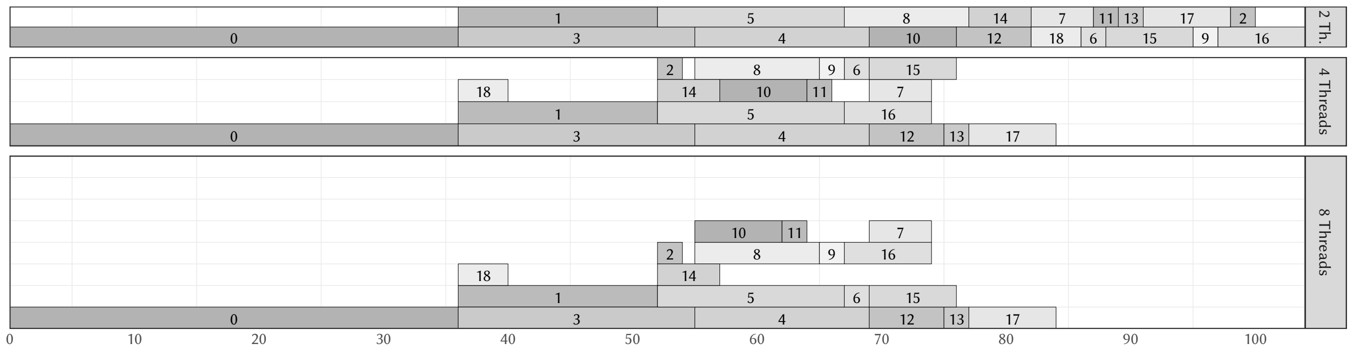
Figure 4 shows the calculated schedule as a *Gantt chart* [Gantt, 1913] for 2, 4 and 8 threads (top, middle, bottom, respectively). Each rectangle represents a trace. The horizontal axis depicts the length in terms of *instructions*.

The length of the schedule with 2 threads is 104 instructions with a utilization of 81%. With 4 threads the utilization decreases to 50%. However, the length of 84 is already optimal, i.e., the *critical path* length. Therefore, increasing the number of threads to 8 cannot yield any improvements and utilization drops to 25%. Figure 4 also shows that although 8 threads are available, only 5 are used.

5 PRELIMINARY EMPIRICAL EVALUATION

To verify that the theoretic results presented in Section 4 also apply in practice, we did a proof-of-concept implementation of the parallel trace register allocation approach. More specifically, we wanted to ensure that parallel register allocation (1) can improve register allocation latency, and (2) does not impact allocation quality. To see whether these goals can be fulfilled, we implemented the parallelization approach in the trace register allocator which is part of GraalVM [Eisl et al., 2016, 2017].

[3]See `java.io.PrintStream#write(byte[], int, int)`



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

Figure 4: Trace Allocation Scheduling for `PrintStream#write`

5.1 GraalVM

The GraalVM is a Java virtual machine based on the HotSpot VM. The performance of GraalVM is similar or even better than HotSpot VM [Prokopec et al., 2017; Simon et al., 2015; Stadler et al., 2013].

Graal uses two different intermediate representations (IR). In the *front end*, Graal performs optimizations such as inlining, partial escape analysis [Stadler et al., 2014], and code duplication [Leopoldseder et al., 2018], to name just a few. It uses a graph-based IR in static single assignment (SSA) form [Duboscq et al., 2013].

After applying all optimizations, the graph-based representation is converted to a low-level control-flow-graph-based IR before entering the *back end*. First, the LIR still adheres to SSA form. The back end's main responsibility is register allocation and code generation. The register allocator also translates the IR out of SSA form by replacing φ-functions with move instructions.

5.2 Parallel Register Allocator Implementation

To empirically verify our approach, we added a simple *proof-of-concept* implementation. Since Graal itself is written in Java, we can utilize the concurrency primitives provided by the Java standard library. We used one Java *thread pool*[4] with a fixed pool size for all allocation threads. Idle threads are kept alive to avoid the thread starting overhead. Once a trace is allocated, all its successors are added to the work queue, given that all their dependencies are already allocated. We use a *priority queue*[5] 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 and for tracking finished dependencies. 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. We are confident that most of them could be mitigated by a more advanced implementation.

[4]See `java.util.concurrent.ThreadPoolExecutor`

[5]See `java.util.concurrent.PriorityBlockingQueue`

5.3 Methodology

By default, GraalVM uses multiple compiler threads to concurrently compile different methods. Compilation happens in the background, that means the application continues to execute while a method is compiled. The compiler threads compete with the application threads. Adding threads for register allocation makes the situation even more challenging. To keep this interference low, we only use one (background) compiler thread that can access up to 8 *register allocation threads*. To measure register allocation latency, we take a timestamp[6] before and after register allocation and report the difference, i.e., the *duration*. In other words, it is the time elapsed from the beginning of register allocation until it is finished and the result is available. Naturally, the numbers are influenced by the scheduling of threads by the virtual machine. For example, if the VM decides to preempt the register allocation threads in favor of an application thread, the *duration* increases although the compiler or allocator did not perform more work. More precisely, the *duration* does not represent *CPU time*. However, the metric of *duration* is what we are interested in, since the goal is not to reduce the *work* that is done by the allocator but to have the result available *earlier*.

5.4 Hardware Environment

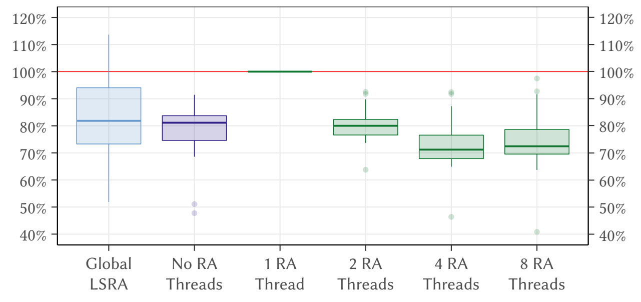
We performed the experiments on a cluster of 64 identical Sun Server X3-2 machines,[7] equipped with two Intel "Sandy Bridge" Xeon E5-2660 at 2.20GHz with 8 real cores per processor, and 256GB of DDR3-1600 memory. The machines were running an Oracle Linux Server 6.8 operating system with Linux Kernel 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 every benchmark we started a new Java VM with an initial and maximum heap size of 8GB. To avoid distortion due to node switching, we fixed the CPU and the memory of the JVM to a single NUMA node using the `hwloc-bind` utility.[8] For each configuration, we collected at least 20 results.

[6]See `java.lang.System#nanoTime()`

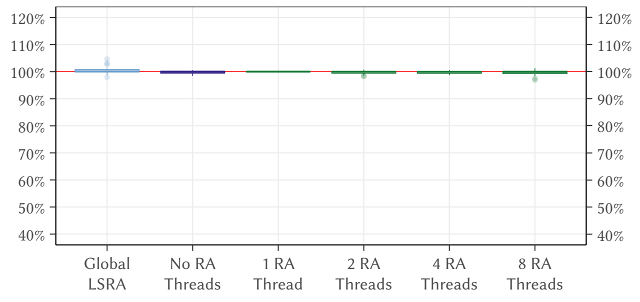
[7]Sun Server X3-2: http://docs.oracle.com/cd/E22368_01/

[8]hwloc-bind(1) — Linux man page: https://linux.die.net/man/1/hwloc-bind



DaCapo and Scala-DaCapo on AMD64. Values relative to the mean of the *one register allocation thread* configuration. Table 1 summarizes the results.

Figure 5: Register Allocation Latency (lower is better ↓)



DaCapo and Scala-DaCapo on AMD64. Values relative to the mean of the *one register allocation thread* configuration. Table 1 summarizes the results.

Figure 6: Benchmark Execution Time (lower is better ↓)

RA	Bench. Execution Time				Reg. Allocation Latency			
Threads	gm	max	med	min	gm	max	med	min
None	100	101	100	98	77	91	81	48
1	100	100	100	100	100	100	100	100
2	100	101	100	98	80	93	80	64
4	100	101	100	99	72	92	71	46
8	100	101	100	97	73	97	72	41

Values in percent (%) relative to one register allocation thread (RA Threads). We present geometric mean (gm), minimum (min), median (med), and maximum (max) results for benchmark execution time and register allocation latency.

Table 1: Experimental Results (lower is better ↓)

5.5 Results

The experimental results are summarized in Figure 5 and Table 1. Note that the reported values are the duration of register allocation, including all necessary phases. For trace register allocation, this includes *trace building*, *global liveness analysis* and *global data-flow resolution*. All numbers are relative to the configuration with *one register allocation thread* (1 RA Thread). 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 a trace register allocation configuration where all work was done on the compiler thread (No RA Threads). The overhead of 23% might seem high at first sight. However, as already mentioned before, our implementation is an initial proof-of-concept prototype. We are confident that this gap can be reduced by an advanced implementation. To show that the trace register allocation approach is *on par* with state-of-the-art allocators, we also include results for Global LSRA, the global linear scan implementation used by default in GraalVM.

To verify the question whether 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 20% (7%, 36%). Increasing the thread count to 4 reduces the allocation latency by 28% (8%, 54%), compared to a single thread. As we already expected from the potential analysis, using 8 threads does not yield any advantages. In fact, the result is slightly worse than with four threads. This is due to the additional synchronization effort and the fact that we are close to the number of hardware threads.

The second assumption we wanted to verify in this evaluation is that parallel register allocation does not affect allocation quality. To do so, we report steady-state performance in Figure 6 and Table 1. For the DaCapo-style benchmarks, this is the time for an iteration after the benchmark has warmed up. Ideally, in this iteration the VM does not perform any compilation, although that is not always the case [Barrett et al., 2017]. The results in Figure 6 and Table 1 suggest that parallel register allocation has no negative impact on allocation quality.

6 RELATED WORK

Eisl et al. [2016] already suggested that the trace register allocation approach can be extended to parallel allocation. We verified their idea, described a dependency model, analyzed the potential and evaluated a prototype.

6.1 Concurrent Compilation

Just-in-time compilers often apply concurrency techniques to improve compilation performance. The HotSpot VM, for example, executes the compilers in background threads [Kotzmann et al., 2008]. So does V8, Google's JavaScript engine [McIlroy, 2018]. The advantage of this approach is that the main thread can continue executing the program, for example, in an interpreter, while a method compiles. HotSpot also uses multiple compiler threads [Oracle Corporation, 2015]. The synchronization overhead is low. Compilation threads need to synchronize at the beginning when taking a method from the compilation queue and at the end when installing the code. The compiler itself can be single-threaded. Also, adding more compiler threads scales well, as long as there are enough methods in the queue. Background compilation on multiple threads improves throughput, i.e., the number of units compiled in a given time frame [Krintz et al., 2001]. However, it cannot improve compilation latency of a single method. This is in contrast to our parallel register allocation approach.

6.2 Trace and Region-based Compilation

In this work we focused on *method-based* compilation where the input to the compiler is a method. Many method-based compilers perform inlining to enable further optimization opportunities. To keep the size of the compilation unit manageable, JIT compilers such as Graal or the HotSpot server compiler replace infrequent branches with deoptimization [Hölzle et al., 1992; Stadler et al., 2013]. Instead of compiling code for such branches the execution is continued in the interpreter. Trace-based compilers, for example Dynamo by Bala et al. [2000] or the HotSpot client compiler modifications by Häubl et al. [2013], follow a different approach. They *trace* frequent execution paths and use them as a compilation unit. Conceptionally, the idea is similar to trace register allocation but extended to the whole compilation process. Although the compilation of such a trace cannot be parallelized, multiple traces can be compiled concurrently. Therefore, trace-based compilation reduces compilation latency. The same applies to region-based compilation [Hank et al., 1995], where the compilation units consists of more complex control structures than linear traces.

6.3 Register Allocation Approaches

To the best of our knowledge, parallel register allocation approaches have not been described in literature. However, we looked at existing approaches to find out whether the same idea could be applied there. Since we do not have access to their implementation, this analysis is more a thought experiment than a thorough comparison. Traditional global approaches such as *graph coloring* [Chaitin et al., 1981] or (global) linear scan [Poletto and Sarkar, 1999] are out of question, since they work per design on the whole compilation unit.

Callahan and Koblenz [1991] proposed *hierarchical graph coloring* where the compilation unit is organized in a tree of *tiles*, that are non-overlapping sets of basic blocks. Register allocation is performed in a bottom-up (leafs-to-root) and a top-down (root-to-leafs) phase. Tiles are related to traces in our approach, and the tree representation can offer parallelization potential. However, explicit synchronization is required as information is shared between tiles.

A related idea was proposed by [Lueh et al., 1997] under the name *graph-fusion-based register allocation*. Their allocator works on *regions* of the control-flow, for example a basic block or a loop, and builds the interference graph for it. The graphs for regions that are connected via an edge are then *fused* to get the combined graph. Although there is some potential for parallelizing the graph building and simplification, most of the work is delayed until the complete graph is available. In contrast to that, in trace register allocation, the tasks are independent and are only connected via value-location-maps at trace boundaries.

7 FUTURE DIRECTIONS

For this parallel trace register allocation experiment, we applied a strict dependency model to guarantee the same results in serial and parallel mode. *All* preceding traces with higher probability must have been allocated before processing the succeeding trace. This could be relaxed to *only the most important predecessor*, which would open up more potential for parallelization. We used the number of instructions as the priority function for selecting the next trace to be processed. An alternative would be the number of successors in the dependency graph, to enable more concurrency opportunities. Eisl et al. [2017] proposed using the *bottom-up allocator* to improve compile time based on allocation policies. In this work, we did not investigate combining this idea with parallelization.

8 CONCLUSION

We presented a *parallel register allocator* based on *trace register allocation*. Since traces which do not depend on each other can be allocated concurrently by different threads, the approach reduces compilation latency without impacting the allocation quality, i.e., peak performance. The analysis of common Java benchmarks revealed that there is almost 50% improvement potential for reducing register allocation latency. Compared to a single thread, our prototypical implementation reduces register allocation latency by 20%, 28%, or 27% for 2, 4, or 8 allocation threads, respectively.

Although our prototype is in an early stage, the results show that parallel register allocation is practical and can improve compilation latency. It further underlines the flexibility of trace register allocation.

ACKNOWLEDGMENTS

We thank the GraalVM 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. We also thank the anonymous reviewers for their valuable feedback. Josef Eisl and David Leopoldseder are 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.

Barrett, Edd, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt (2017). Virtual Machine Warmup Blows Hot and Cold. In: Proc. ACM Program. Lang. DOI: 10.1145/3133876.

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

Callahan, David and Brian Koblenz (1991). Register Allocation via Hierarchical Graph Coloring. In: SIGPLAN Not. DOI: 10.1145/113446.113462.

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

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

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 (2015). Trace Register Allocation. In: *SPLASH Companion 2015*. ACM. DOI: 10.1145/2814189.2814199.

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

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. DOI: 10.1145/5666.5673.

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

Hank, R.E., W.W. Hwu, and B.R. Rau (1995). Region-based compilation: an introduction and motivation. In: Proceedings of the 28th Annual International Symposium on Microarchitecture. DOI: 10.1109/micro.1995.476823.

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

Hennessy, John L. and David A. Patterson (2003). *Computer Architecture: A Quantitative Approach*. 3rd ed. Morgan Kaufmann Publishers Inc. ISBN: 1558607242.

Hölzle, Urs, Craig Chambers, and David Ungar (1992). Debugging Optimized Code with Dynamic Deoptimization. In: SIGPLAN Not. DOI: 10.1145/143103.143114.

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.

Krintz, Chandra J., David Grove, Vivek Sarkar, and Brad Calder (2001). Reducing the overhead of dynamic compilation. In: Software: Practice and Experience. DOI: 10.1002/spe.384.

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*. ACM. DOI: 10.1145/3168811.

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

McIlroy, Ross (2018). *V8 JavaScript Engine: Background compilation*. URL: https://v8project.blogspot.co.at/2018/03/background-compilation.html (visited on 03/28/2018).

Oracle Corporation (2015). *JRockit to HotSpot Migration Guide: Compilation Optimization*. URL: https://docs.oracle.com/javacomponents/jrockit-hotspot/migration-guide/comp-opt.htm (visited on 03/28/2018).

Pinedo, Michael L. (2016). *Scheduling: Theory, Algorithms, and Systems*. 5th ed. Springer International Publishing. DOI: 10.1007/978-3-319-26580-3.

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.

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.

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.