Trace Register Allocation *

Josef Eisl
Johannes Kepler University Linz, Austria
josef.eisl@jku.at

Abstract

This paper proposes the idea of Trace Register Allocation, a register allocation approach that is tailored for just-in-time (JIT) compilation in the context of virtual machines with run-time feedback.

The basic idea is to offload costly operations such as spilling and splitting to less frequently executed branches and to focus on efficient registers allocation for the hot parts of a program. This is done by performing register allocation on traces instead of on the program as a whole.

We believe that the basic approach is compatible to Linear Scan, the predominant register allocation algorithm for just-in-time compilation, in both code quality and allocation time, while our design leads to a simpler and more extensible solution. This extensibility allows us to add further enhancements in order to optimize the allocation based on the run-time profile of the application and thus to outperform current Linear Scan implementations.

Categories and Subject Descriptors D.3.4 [*PROGRAMMING LANGUAGES*]: Processors—*Code generation, Compilers, Optimization*

General Terms Performance, Algorithms, Experimentation

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

1. Introduction

Register allocation is considered as an important compiler optimization [4, 16, 24] that has been researched intensively. Chaitin et al. [6] 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 improvement where proposed including Briggs et al. [4], Smith et al. [19], or George et al. [8]. Chaitin et al. showed that register allocation is in general NP-complete [6], so all 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 suboptimal for just-in-time compilation. JIT compilers therefore often use the simpler Linear Scan approach, which was introduced by Poletto et al. [16] and extended in many subsequent contributions [22, 24, 23]. Linear Scan is used in numerous dynamic compilers including the HotSpot client compiler [12], Jikes RVM [2], Googles JIT compiler for JavaScript (V8), and initially also in LLVM [13].

2. Problem

As the name suggests, Linear Scan works on a linearization of the control-flow graph. Although a carefully selected block ordering can improve the code generated by the algorithm [23], a linearization of blocks is in general a poor approximation of the program structure and lead to suboptimal decisions because unrelated control-flow paths influence each other. This is even more problematic in the context of dynamic compilation where we have run-time feedback about the program behavior available.

Spilling and splitting is difficult in Linear Scan because the natural positions, such as control splits, are not directly visible to the algorithm. Heuristics that were proposed to overcome this (e.g. Wimmer et al. [24]), give some remedy but still spilling and splitting does not fit nicely into the simple Linear Scan model.

In addition to that, exact lifetime information is complicated to compute and maintain because an interval might contain holes [22], i.e. ranges where the value is not live. This increases the complexity of the algorithm [24].

3. Approach

When thinking about register allocation in a dynamic compiler we can make two key observations:

* This research project is funded by Oracle Labs

SPLASH Companion'15, October 25–30, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3722-9/15/10...$15.00
http://dx.doi.org/10.1145/2814189.2814199

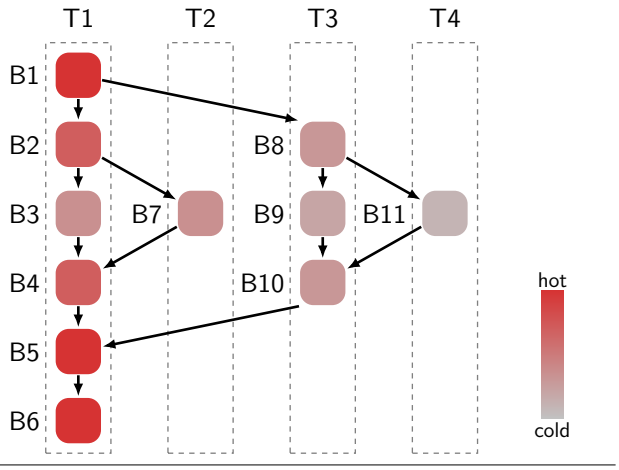

Figure 1. Trace representation of a control-flow graph

- The register allocation is done at run time so it must be fast. We cannot afford complex approaches. The goal is an algorithm with linear or near to linear time complexity.
- The compiler is invoked dynamically by a runtime system that can provide profiling information about the application. The register allocation algorithm should utilize this knowledge for improving the quality of the generated code, assuming that the program behavior does not change.

These observations are the basis for the following considerations. The idea is to offload the code for spilling and splitting to infrequently executed branches of the compilation unit. This is similar to the approach of Lueh et al. [15], but instead of striving for the best possible global allocation, we want to find a good allocation for the fast path in acceptable time.

Trace Representation. Based on profiling information we divide the control-flow graph into distinct regions of sequentially executed blocks. Due to similarities to trace compilation [14] we call these regions *traces*.

To build a new trace we select a block with no predecessors or 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 more successor that is not already assigned to a trace. The algorithm continues as long as there are blocks that are not yet part of a trace. Figure 1 depicts an example. Each trace is associated with a probability that can be calculated from the profiling information.

After the traces are built we process one trace at a time in order of decreasing probability. For each trace we perform a liveness analysis, make spilling and splitting decisions and allocate registers. There is no need to visit blocks of other traces. Since traces of higher priority have already been allocated we can use the mappings of variables to registers or spill slots at the control-flow splits and joins in order to guide the allocation of the current trace. Values that are live on an edge to a trace that has not yet been allocated are ignored.

We use a special intermediate representation to capture the variable mappings at trace boundaries. It can be seen as a generalization of the ϕ and σ functions in Static Single Information (SSI) form as introduced by Ananian [1]. The representation adheres to the *single definition* and the *dominance* property, i.e. for variable there is just a single definition and this *definition dominates all usages*. This allows us to use simple liveness analysis algorithms. The control-flow graph does not contain *critical edges* which makes the placement of spill and split code easier.

In the course of this project we want to answer the following research questions. How do the profiles for real world applications look like? Are there many equally hot traces or is there usually just one hot trace and many less frequently executed ones? Are loops represented well enough by traces to achieve good register allocation, or do we need special handling? This is particularly interesting when there are control-flow splits inside the loop.

The trace model and the special intermediate representation create a lot of opportunities for register coalescing. Are register hints [24] sufficient or are stronger approaches required?

Register Allocation for Traces. In general, register allocation for a trace is independent of how the traces are built: any approach can be used for that. Nevertheless, traces have special properties that are worth exploiting. A trace is a linear sequence of basic blocks and can therefore be viewed as a single extended basic block. This property can be utilized as suggested by Guo et al. [10]. As there are no lifetime holes in this extended block the liveness information is much simpler to compute and maintain than in the global Linear Scan algorithm, and the analysis can be done in a single backwards pass over the trace. The heuristics for spilling and splitting can and should be adapted to move such code to less frequently executed traces. The interference graphs of programs in SSA form are chordal [5, 11] and can therefore be optimally colored in linear time [17]. This allows the application of *decoupled register allocation* as suggested by Brisk [5] or Hack et al. [11].

We want to stress again that the traces are built and treated independently, so it is possible to use different algorithms for different traces. For instance, a better but more expensive algorithm can be used for traces with higher execution frequencies, whereas a faster but less optimal algorithm can be used for others.

4. Evaluation Methodology

In order to evaluate our approach we will implement an allocator for the Graal compiler [9, 7], a research compiler for the HotSpot VM. The performance of code produced by

Graal is comparable to production compilers. The Truffle language implementation framework [25], that is build on top of Graal, allows us to evaluate our implementation with languages other than Java including JavaScript, R and Ruby. This makes it a perfect target for our experiments. The baseline is the Linear Scan implementation [24] that is currently used by Graal.

4.1 Hypothesis

The experiment will be performed in two phases. In the first phase we want to study the trace register allocation model in general. For the allocation of traces we apply the unmodified Linear Scan algorithm to each trace independently. We expect results comparable to the global Linear Scan algorithm with respect to code quality and compile time. The advantage of the proposed approach lies in the simplicity of lifetime analysis and interval representation (no lifetime holes).

In the second phase we focus on exploiting the unique properties of trace register allocation as described above. Compared to the global Linear Scan algorithm we expect to achieve better allocation for the hot parts of the program, which leads to better run-time performance. As a result the number of dynamically executed instructions is decreased due to the better placement of spill and split code. Additionally, the code density for the hot paths increases, which has a positive effect on instruction cache behavior.

Please note that we explicitly accept suboptimal code for less frequently executed parts (i.e. code that is possibly slower and larger than the code generated with the global Linear Scan allocation). The focus of our approach is on the frequently executed parts and thus on increased overall performance.

4.2 Experimental Setup

In order to ensure that our implementation is not biased towards a specific architecture we perform the evaluation on Intel x86_64 as well as on the SPARCv9 platform.

Using the HotSpot VM as a platform for our experiments we can verify our approach with a wide range of well-known benchmarks including SPECjvm2008 [21], SPECjbb2013 [20], DaCapo [3], and Scala DaCapo [18]. Truffle further extends the space of benchmarks to languages that do not directly target the JVM.

The ultimate metrics are the performance numbers of the benchmarks which are, for instance, run time, score, or operation per time unit, depending on the harness.

In order to gain further insights into the characteristics of the algorithm, other metrics are also of interest. To verify that our approach is offloading code to the less frequently parts we count the *dynamically executed* move operations. Additionally, the number of emitted instructions should be monitored to avoid regressions due to increased code size. To gain confidence that the complexity of our implementation is linear or nearly linear we record the compilation time per number of source instructions.

References

[1] C. Scott Ananian. “The Static Single Information Form”. MA thesis. Princeton University, 1997.

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

[3] S. M. Blackburn et al. “The DaCapo Benchmarks: Java Benchmarking Development and Analysis”. In: *OOPSLA ’06*. 2006. DOI: 10.1145/1167473.1167488.

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

[5] Philip Brisk. “Advances in Static Single Assignment Form and Register Allocation”. PhD thesis. 2006.

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

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

[8] Lal George and Andrew W. Appel. “Iterated register coalescing”. In: *TOPLAS’96* (1996). DOI: 10.1145/229542.229546.

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

[10] Jia Guo, Maria Jesus Garzaran, and David Padua. “The Power of Belady’s Algorithm in Register Allocation for Long Basic Blocks”. In: *LCPC’03*. 2003. DOI: 10.1007/978-3-540-24644-2_24.

[11] Sebastian Hack, Daniel Grund, and Gerhard Goos. “Register Allocation for Programs in SSA-Form”. In: *CC’06*. 2006. DOI: 10.1007/11688839_20.

[12] Thomas Kotzmann et al. “Design of the Java HotSpotTM client compiler for Java 6”. In: *TACO’08* (2008). DOI: 10.1145/1369396.1370017.

[13] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *CGO ’04*. 2004. DOI: 10.1109/CGO.2004.1281665.

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

[15] Guei-Yuan Lueh, Thomas Gross, and Ali-Reza Adl-Tabatabai. “Fusion-based Register Allocation”. In: *TOPLAS’00* (2000). DOI: 10.1145/353926.353929.

[16] Massimiliano Poletto and Vivek Sarkar. “Linear Scan Register Allocation”. In: *TOPLAS’99* (1999). DOI: 10.1145/330249.330250.

[17] Hongbo Rong. “Tree Register Allocation”. In: *MICRO 42*. 2009. DOI: 10.1145/1669112.1669123.

[18] Andreas Sewe et al. “Da capo con scala”. In: *OOPSLA’11* (2011). DOI: 10.1145/2048066.2048118.

[19] 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.

[20] *SPECjbb2013: Java Server Benchmark*. URL: https://www.spec.org/jbb2013/.

[21] *SPECjvm2008: Java Virtual Machine Benchmark*. URL: https://www.spec.org/jvm2008/.

[22] 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.

[23] Christian Wimmer and Michael Franz. “Linear Scan Register Allocation on SSA Form”. In: *CGO ’10*. 2010. DOI: 10.1145/1772954.1772979.

[24] Christian Wimmer and Hanspeter Mössenböck. “Optimized Interval Splitting in a Linear Scan Register Allocator”. In: *VEE ’05*. 2005. DOI: 10.1145/1064979.1064998.

[25] Thomas Würthinger et al. “One VM to rule them all”. In: *Onward! ’13*. 2013. DOI: 10.1145/2509578.2509581.