

Graph-Coloring and Treescan Register Allocation Using Repairing

Quentin Colombet
INRIA/LIP*
quentin.colombet@ens-
lyon.fr

Benoit Boissinot
ENS/LIP*
benoit.boissinot@ens-
lyon.fr

Philip Brisk
University of California
Riverside
philip.brisk@ucr.edu

Sebastian Hack
Saarland University
hack@cs.uni-saarland.de

Fabrice Rastello
INRIA/LIP*
fabrice.rastello@ens-
lyon.fr

ABSTRACT

Graph coloring and linear scan are two appealing techniques for register allocation as the underlying formalism are extremely clean and simple. This paper advocates a decoupled approach that first lowers the register pressure by spilling variables, and then performs live ranges splitting/coalescing/coloring in a separate phase; this enables the design of simpler, cleaner, and more efficient register allocators.

This paper gives a new and more general approach to deal with register constraints. This approach called *repairing* does not require pre live range splitting and does not introduce additional spill code. It ignores register constraints during coloring/coalescing, and repairs violated constraints afterwards.

We applied this method to both graph based and scan based allocators into a decoupled approach. Here, the Iterated Register Coalescer (IRC) and a scan algorithm that uses Static Single Assignment (SSA) properties, the treescan. Moreover, this paper provides a survey on existing and new techniques of bias coloring during scan approaches.

Our experimental evaluation shows for the graph based approach, that we reduced the number of vertices (edges) in the interference graph by 26% (33%) without compromising the quality of the generated code. The treescan algorithm improved the compile time of the allocation process by 6.97x over IRC while providing comparable results for the quality of the generated code.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—Code generation, Compilers, Optimization

*LIP, Université de Lyon - ENS Lyon - CNRS - INRIA - UCBL, Lyon, France

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CASES'11, October 9–14, 2011, Taipei, Taiwan.

Copyright 2011 ACM 978-1-4503-0713-0/11/10 ...\$10.00.

General Terms

Algorithms, Design, Performance

Keywords

Fast register allocation, Coloring, Coalescing, SSA form, Register constraints

1. INTRODUCTION

Register allocation assigns processor registers to variables in a program that can be held in registers. Besides assigning registers, register allocation also performs several others tasks. First, *spilling* splits live ranges of variables around program points where more variables are live than registers are available (we also say the register *pressure* is excessive). Second, *coalescing* tries to eliminate copy instructions that are left over from earlier compilation phases by assigning the source and the target of the copy the same register.

An important detail of the register assignment process is register constraints imposed by the instruction set architecture or the application binary interface (ABI). Certain variables must be in certain registers at certain program points. For example, the first integer argument of a function call on the ARM Linux ABI must be passed in register R0. Similarly, certain instruction sets require that the source or destination operands of certain instructions reside in specific registers (e.g., division in IA32).

These constraints are local to instructions and are usually handled by the allocator using copy instructions, i.e., by *prematurely* splitting live ranges prior to assignment via parallel copy operations. This places additional pressure on the coalescer to eliminate as many of these extra copies as possible. Moreover, coalescing is the most costly task of register allocation [11, 20] and is NP-complete (within the size of the program) [6, 21]. Thus, one may want to avoid the size expansion implied by these live ranges splitting.

This paper proposes a new technique called *repairing* that deals with these kind of register constraints and does not require pre live ranges splitting. We emphasize that repairing is useful when certain variables *must* be assigned to a subset of registers, possibly a singleton, [23, 1, 20, 35]. Repairing *ignores* register constraints during allocation and *repairs* constraints that have been violated afterwards. Repairing is compatible with the elegant formalisms that have made graph coloring, linear scan, and Static Single Assignment

(SSA)-based decouple register allocation techniques appealing in the first place. Moreover, it saves the overhead of premature live range splitting; and lastly, repairing can be integrated into the iterated register coalescer (IRC) [20] graph-coloring register allocator through the introduction of *negative affinities* without substantially changing its architecture or implementation.

We also present how repairing can be applied to the recent *treescan* allocators in SSA-based register allocation. Those allocators use a *decoupled approach* [1]: Several theorems that relate SSA to liveness and interference [5, 13, 22] guarantee that the register pressure of the program *equals* its register demand. Thus, in SSA-based register allocation, one uses a spiller that decreases the register pressure to the number of available registers K . Then, a simple scan-like allocator that works on the dominance tree produces a register assignment in linear time *without* introducing further spilling [7, 10].

Repairing does *not* address register bank irregularities, such as aliasing [32] or register pairing; one method to handle these kind of constraints in the context of a decoupled register allocator is to split every variable at every point where it is live [28]; however, handling aliasing constraints without excessive splitting remains an open problem, which we do not attempt to address here. Repairing is concerned with constraints that are local to individual instructions and can be handled using copy instructions.

In summary, we make the following contributions:

- In Section 2, we present an extension to IRC. We use *negative affinities* to inform the allocator that constrained live ranges should avoid certain registers. Unlike classical coalescing, negative affinities signal a register *dislike*, rather than a preference as expressed by *positive* affinities. Using negative affinities, hints for register constraints can be modeled without significantly blowing up the size of the interference graph. We show how negative affinities can be modeled by interferences and positive affinities and can thus be incorporated into existing allocators without changing the allocator itself.
- In Sections 3 and 4, we show how repairing can be used in tree scan allocators. We show how the copies introduced by repairing can be avoided without spending the large compile-time penalty of coalescing. To this end, we present by several heuristics to *bias* the color choice in the coloring algorithm.
- In Section 5, we present the related work.
- In Section 6 we present an extensive experimental evaluation that shows the effectiveness of our techniques on the integer part of the Spec CINT2000 benchmark suite. Our graph-coloring allocator using repairing produces interference graphs that have 26% less nodes (33% less edges) compared to IRC with live-range splitting without increasing the run time of the compiled program. Our base line tree scan algorithm produces code of the same quality as our IRC implementation at a allocation time speedup of 6.97x. Activating our biasing techniques, we meet the run time performance of IRC while the allocation time speedup compared to IRC is still 3.15x.

Finally, Section 7 concludes the paper.

2. GRAPH COLORING WITH REPAIRING

Many compilers use an interference graph to guide register allocation; to conserve space, we assume that the reader is familiar with interference graphs and their related concepts such as liveness and affinities between move-related variables. In principle, any graph coloring register allocator can be modified to handle register constraints through the introduction of pre-colored vertices [20]. Any variable u that is preassigned to register R is merged with R 's pre-colored vertex. Unfortunately, preemptively merging vertices can cause additional spilling. To limit the lifetime of constrained variables and the variables that interfere with them, the allocator splits, prior to coloring, live ranges with parallel copies before each constrained instruction [24, 21], as illustrated in Figure 1 in SSA context; in general, this can reduce the amount of additional spilling, and in the case of a decoupled SSA-based register allocator, it can ensure that the coalescer performs no additional spilling.

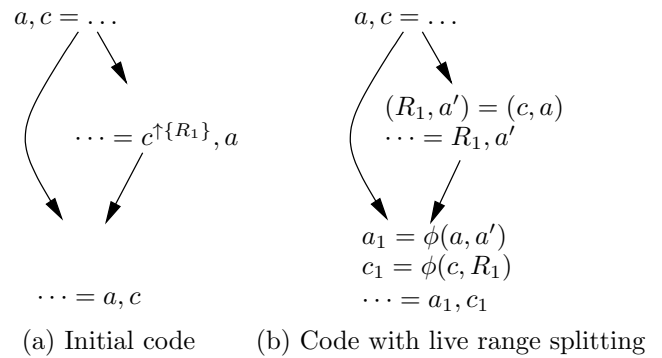


Figure 1: Effects of live range splitting. $(R_1, a') = (c, a)$ stands for a parallel copy where $R_1 = c$ and $a' = a$ are done in parallel. $c^{\uparrow\{R_1\}}$ indicates that c has to be in the related subset of registers, here $\{R_1\}$.

2.1 Model and restrictions

Register constraints have different variants. Commonly several registers are charged with a special meaning throughout the program such as the stack or frame pointer. Hence, they are usually not subject to register allocation and excluded from the set of available registers.

In this paper, we consider the most common constraints: An instruction dictates that an operand has to be in a specific subset of registers, a register class. Such constraints often occur in calling conventions of the ABI. Each argument to a function call has to be put into a dedicated register.

Figure 2 illustrates how this constraint is modeled using negative affinities. In Figure 2a, $b^{\uparrow\{R_1, R_3\}}$ states that the corresponding operand that uses b is constrained to a register class made of registers R_1 and R_3 . If, for some reason, b is assigned to R_2 then some shuffle code has to be inserted prior to (and after) the constrained instruction to copy b to (and respectively from) either R_1 or R_3 , as shown in Figure 2b.

As shown in Figure 2c, an affinity of weight $-2w$ between b and R_2 indicates that assigning b to R_2 will require at least two repairing copies around the constrained instruction.

2.2 Strategies

We have integrated support for negative affinities into the IRC graph-coloring register allocator by George and Appel [20]. The original IRC implementation performs spilling

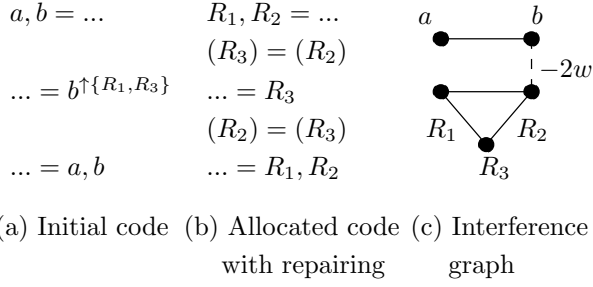


Figure 2: If a and b are respectively allocated to R_1 and R_2 some repairing code is inserted. The affinities in the interference graph model this cost.

and coalescing together; as our compiler uses a decoupled approach, and a different spilling algorithm, we have implemented a stripped-down version of IRC that does not perform spilling. Specifically, we omit the *potential spill*, *select*, and *actual spill* stages from their algorithm (see Figure 5 in [20]). Within this algorithm, we propose three different strategies to handle negative affinities.

The IRC algorithm and its stripped-down version iteratively transform the graph by merging (*coalescing*) some affinity related nodes. They also remove nodes of low degree that are not affinity related (*simplification*). Every simplified node is pushed onto a stack. This is the coalescing, simplification phase. When all nodes are simplified it pops nodes from the stack and assigns a color. This is the color phase.

The coalescing process uses an ordered (by decreasing weight) work-list of affinities (worklistMoves in [20]). For each affinity the algorithm checks by simple rules (namely Brigg’s & George’s) to see if the two nodes can be coalesced conservatively (i.e. will not constrain too much the coloring). If it can, it merges the nodes, otherwise, put the affinity in some other lists. Optimistically, a judicious choice of color still has the possibility to satisfy some or all of the non-coalesced affinities when it is later popped from the stack and assigned a color; this is called *biased coloring*, as discussed by Briggs et al. [12].

The three strategies for dealing with negative affinities are illustrated in Figure 3; each strategy is discussed in detail in the following three subsections.

Freeze Adding negative affinities in the graph and letting the IRC to cope with it is definitely a bad idea as, even if the weight of an affinity is negative, it will try to satisfy it, in other words merge the two corresponding nodes. The first technique we present avoids this behavior by freezing all negative affinities, therefore ignoring them during the coalescing phase. The biased coloring [12] approach of the color assignment phase is modified to take negative affinities into account.

Dummy Nodes The second technique consists in transforming a graph with negative affinities into an equivalent graph with only positive affinities. Every negative affinity (x, y) of weight w is replaced by a sequence of an interference edge (x, xy) , with a new vertex xy called a *dummy node*, which does not correspond to an actual variable in the program, and a positive-weighted affinity (xy, y) of weight w .

Conservative Interference The basic idea of this third technique is to *conservatively* replace a negative affinity edge

(x, y) with an interference edge, when doing so does not affect the colorability of the interference graph. Recall that the work-list is sorted using affinity weight. Our first modification consists in considering the absolute value of the weights in this sorting. Then when an affinity is popped from the work-list, this affinity might be a negative affinity. If positive, the code is unchanged: the conservative coalescing tests are performed and if successful the two corresponding nodes are merged. If negative, we run instead a conservative rule for checking if the negative affinity can be conservatively (regarding the graph colorability) replaced by an interference. If the test is successful the interference is actually added, the degrees of the corresponding nodes updated, and their position in the many work-lists handled by IRC updated also.

The rule can be stated as follow: let K be the number of available registers. Let (u, R) be a negative affinity between a regular vertex u and a pre-colored vertex R ; we can replace (u, R) with an interference edge if, afterwards, u has at most $K - 1$ neighbors of degree at least K .

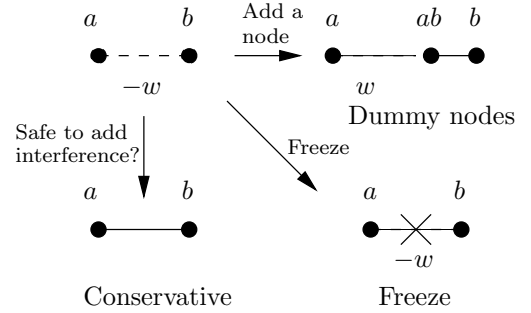


Figure 3: Strategies to deal with negative affinities.

2.3 Repairing Code

When coloring is over, repairing code has to be inserted for each negative affinities that have been satisfied. Repairing can be understood as an allocation problem restricted to a very small region around the constrained instruction. Consider the example of Figure 4 once variables a , and c have been assigned register R_1 , and R_2 . Since the allocation does not satisfy the constraint for the definition of c , all live ranges are split just before and after this instruction. Additionally any live-through variable used in a constrained operand, such as a in Figure 4, is duplicated. One of the copy (here a') is used (and constrained) in the use operand, the other one (here a'') is live-through and potentially constrained by the def operand. Moreover, for each variable whose constrained subset is a singleton, we directly replace this variable by the related register. We end up with a classical allocation problem where copies are affinities to be satisfied and interferences link nodes that cannot share the same register. The corresponding interference graph is represented in Figure 4c. We have represented affinities between interfering nodes, that could obviously not be satisfied. Here a' has to be allocated to R_2 leading to the final code with a copy $R_2 = R_1$ before the instruction and a swap of R_1 and R_2 after. In practice, the allocation problem being very local, the interference graph is not actually built. A greedy ad-hoc heuristic is designed instead.

After repairing, like for every approaches that use the parallel copies [1, 28, 10], we sequentialize them using swap, moves and xor operations [21].

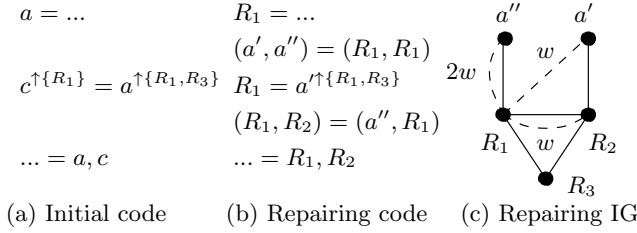


Figure 4: a, c have been allocated to R_1, R_2 . Some parallel copies are introduced to repair the inconsistency. The new local variables a' and a'' have to be allocated. The corresponding interference graph.

3. TREESCAN

In the general graph-coloring setting, the minimum number of registers needed for a coloring of the graph might very well exceed the maximum register pressure of the program. Recent results on SSA-based register allocation show [5, 13, 22] that if the program is in SSA form, its register demand equals its maximum register pressure. This allows for decoupling spilling and register assignment: once the maximum register pressure in the program is lowered to the number of available registers, a *treescan* algorithm manages to assign registers without causing further spills.

To this end, the *treescan* algorithm traverses the dominator tree in pre-order, while processing the definitions and uses of variables in a manner similar to linear scan register allocation [29]. However, in contrast to the original linear-scan algorithm, *treescan* does not overapproximate the live ranges of variables by intervals but uses precise liveness information.

Spilling techniques [9, 21] for SSA programs are not in the scope of this paper; We assume that spilling has already been performed and the register pressure is nowhere larger than the number of registers.

3.1 The Basic Algorithm

The control flow graph (CFG) is processed in reverse post order (in general dominance-preserving order works). Each basic block is traversed from top to bottom. We maintain a bitset of occupied registers that is set to the registers used by the variables that are live-in at the block. However, we do not need to compute liveness for this: in SSA, if a variable is live-in at a block that variable is also live-out of the block's immediate dominator (which we already processed). Hence, we can just copy the occupancy set at the end of the block's immediate dominator and test which of those variables are live-in. When we encounter the definition of a variable, we assign it the next free color. When we encounter a *death point* (the variable dies after this program point) of a variable, we release its color. Instead of performing classical liveness analysis to find the death points, we make use of the fast liveness check by Boissinot et al. [4]. The pseudo code of this algorithm is given in Figure 5.

ϕ -functions, which are conditional parallel copy operations, require special treatment. The use of each operand is

processed at the end of the basic block preceding the block containing the ϕ -function. The variable defined by the ϕ -function is processed at the start of the basic block containing it. After assigning colors to these variables, it may be necessary to insert shuffle code to realize these parallel copy operations (details can be found in [21]).

for block in region.RPO.blocks:

```

# initialize used colors from immediate dominator
block.allocatedVariables = block.idom.allocatedVariables
# remove dead variables
for v in block.allocatedVariables if v is not block.livein:
    block.allocatedVariables -= v
# initialize available colors
availableColors = allocatableColors
- block.allocatedVariables.colors
for inst in block.insts:
    # release last uses colors
    for u in inst.uses if u is not inst.liveout:
        availableColors += u.color
        block.allocatedVariables -= u
    # assign definitions
    for d in inst.defs:
        d.color = availableColors.first
        availableColors -= d.color
        block.allocatedVariables += d
    # release dead definitions colors
    for d in inst.defs if d is not inst.liveout:
        availableColors += d.color
        block.allocatedVariables -= d

```

Figure 5: Basic *treescan* algorithm.

3.2 Repairing

Next, we describe how the *treescan* algorithm can be extended to obey register constraints by repairing. The principle is similar to what we discussed in Section 2.3. However, since *treescan* works directly on the code and not on an abstraction like the interference graph, we can perform repairing more intelligently within a basic block.

Each variable is assigned one *global* color. This is the color that the variable has *across* basic blocks. Local to a basic block, we permit the variable to change its color to fulfill potential constraints. However, we always make sure that at the end of each block, every variable is in its global register.

Repairing at a constrained instruction When we encounter a constraint-imposing instruction during the *treescan*, we check, if all operands are in the appropriate registers to satisfy the constraints. If this is not the case, a parallel copy is inserted in front of the instruction that shuffles the registers appropriately. Behind the instruction we do *not* immediately restore the variables back to their global colors. We delay the restoring parallel copy for several reasons (see below). Thus, inside a block, a variable can be in a register other than its global one. In more detail, the basic algorithm as presented above needs to be modified in the following two aspects.

Selecting a color for a variable Repairing affects the color choice in several ways. Consider the situation that we select the global color of some variable v . This happens exactly at the definition of that variable. The color to choose must be different from the global colors used by interfering

variables. Hence, the global color for v has to be taken from the free global colors. However, it might be that a free global color is *locally* in use at v 's definition. This happens because of repairing: Another variable took that color to fulfill a certain constraint. In that case, we can *temporarily* take another available local color and add v to the restoring parallel copy later in the block. Besides these restrictions, it is beneficial to apply color biasing techniques as detailed in Section 4.

Restoring the global colors To enforce that every variable is in its global color at the end of the block, we insert a parallel copy. This parallel copy can be placed between the last constrained instruction and the end of the block. Its position is best chosen such that the number of registers to shuffle is minimized. This is the latest point in the block where a variable that deviated from its global color died.

4. BIASED COLORING

To this point, we presented how an SSA-based treescan allocator uses repairing to accommodate register constraints. Repairing introduces shuffle code to move the contents of variables to the needed registers. Usually, it is the task of coalescing to remove this shuffle code by finding register assignments that make them superfluous. However, coalescing is a hard problem (it is already NP-hard for SSA programs without constraints [6]). Coalescing algorithms that give good results (eliminate many copies) are complex and too slow (see Section 6) in the Just-In-Time (JIT) context. Hence, in this section, we present several heuristics to *bias* the color choice of the treescan algorithm to give move-related variables the same color in the first place. As our experimental evaluation shows, these heuristics suffice to waive the coalescing pass completely.

Hereafter, we quickly review the adoption of Mössenböck and Wimmer's register hints for treescan and present new biasing approaches that we use in our treescan allocator.

Register hints This technique can be considered as a copy propagation during the scan process. When assigning a color to the result of a move or parallel copy, if the color of the argument is available, the algorithm takes it. We also apply this technique for ϕ -functions results. In a ϕ -function, we have to choose among multiple source variables: one for each incoming edge. We select the color with the highest execution frequency (either determined by static analysis or profile information) over all already allocated sources.

Aggressive pre-coalescing Before coloring, aggressive coalescing is performed *without* applying the results to the program. An aggressive coalescing puts move-related variables into *equivalence classes*. In a classical graph-coloring allocator, the live ranges of the variables in a class are fused. We do not do so but use the classes to bias the coloring: each equivalence class has a color. The color of a class is initially unset and set as soon as a variable of the class is assigned a register. When assigning a color to a variable, treescan checks if the color of the class is available, if so, it takes it. If not, it picks a different color (based on the other heuristics presented here) and updates the class' color. Boissinot et al. [3] and Budimlić et al. [14] show how the SSA properties can be exploited to perform aggressive coalescing on SSA *without* building an interference graph.

Caller saved registers This technique tries to put variables that are live across a call site into registers that are

saved by the *callee*. Thus, it tries to avoid *caller-saved* registers for these variables. The fast liveness check method used by the original treescan algorithm does not help here, because we want to know at the definition of the variable whether that variable is live across a call. In that case we would need to test every call site dominated by the variable's definition. Thus, when using the caller-saved heuristics, we resort to a classic liveness analysis. If aggressive pre-coalescing is used as well, the across-a-call information is also propagated to the equivalence classes.

Round robin assignment The usual choice for a fresh register is to take the first available color, usually in the order of the bit set that tracks the registers in use. However, this paradigm usually leads to an unequal distribution of the colors used. Freed registers are immediately reused by the variable defined next. Hence, some registers are more frequently used than other ones. This can decrease the chance that a move-related variable can reside in the same register. Consider the example in Figure 6. The result of the ϕ -function c is colored before its operand a . The variable d interferes with a . Hence, assigning the register of the class $\{a, b, c\}$ to d is bad because a cannot get it anymore. With the classic allocation strategy this might easily be the case. However, using round robin, the register of c will only be reused after K definitions, where K is the number of available registers. This increases the chances that c 's register is available for a . Round robin assignment also has a positive effect on post-allocation scheduling because it decreases the locality of false dependencies. Thus, a post-allocation scheduler might have more freedom to reorder the instructions while keeping the register allocation.

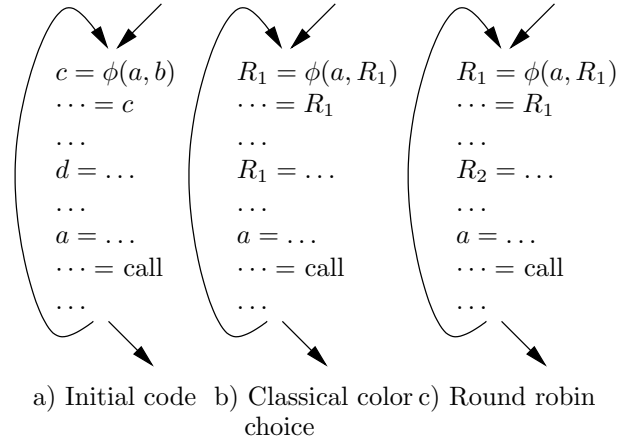


Figure 6: Benefits of round robin on the color choice. Classical color choice reuses c 's color for d and blocks the usage of that color for a . Round robin increases the chances that c 's color will be available at a 's definition.

Move related To further increase the chance for move-related variables to get "their" color (the one of their equivalence class), we divide the register file into two parts: The first part is reserved for move-related variables and is only used by non-move-related variables if registers of the second part are exhausted. Inside the move-related part, we use round robin to assign registers. The division ratio is a parameter; in our experiments, we divided the register file into two equally-sized halves.

5. RELATED WORK

Graph coloring and register constraints Chaitin et al. graph coloring [16] showed that for every undirected graph there is a program that has this graph as its interference graph. In this situation, there is no interest in properties of the graph’s structure. Thus, register constraints were represented as interferences.

More recently, it was shown that the interference graphs of SSA-form programs are chordal, which allows for optimal coloring in polynomial time [5, 13, 22]. However, if one takes register constraints into account, coloring is no longer polynomial. Thus, early SSA-based allocators [22] used premature live range splitting in front of constrained instructions as well. Moreover, Odaira et al. [25] show that live range splitting incur an overhead of 20% on average in the compile time of IRC.

Scan approaches The idea of linear-scan register allocation goes back to Traub et al. [33] and Poletto and Sarkar [29]. Allocation is done with a linear scan over the assembly code. Poletto and Sarkar do not take control flow into account and over-approximate the live range of a variable by an interval on the linearized assembly code. Thus, variables might occupy a register where they are not live and might provoke unnecessary spill code. This method is simple and fast, but gives worse results than standard graph-coloring approaches. Traub et al. perform liveness analysis before and allow for holes in the intervals to avoid the over-approximation of live ranges.

Mössenböck and Pfeiffer [24] proposed a modification of the original linear scan to work on SSA. Unlike our *treescan*, they do not take advantage of SSA properties to allow for an optimal register assignment. Like Traub et al., their live ranges have holes.

Mössenböck and Wimmer [35] further improved linear scan. In particular, they improved spill code placement and added on demand live range splitting to avoid spilling in some context. In 2007, Sarkar and Barik [31] extended the linear-scan. They explicitly split at basic block boundaries to avoid spilling and to handle register constraints at the cost of shuffle code. In our setting, the program is in SSA. Thus, introducing live range splits in addition to ϕ -functions will *not* save any further spills [22]. In 2009, Rong [30] proposed the tree register allocation, which generalizes linear scan approaches. However, this algorithm needs global liveness information, in particular for the handling of pre-colored constraints. In 2010, Wimmer and Franz [34] pointed out the interest of relying on SSA to deal with liveness in linear-scan. Finally, the same year, Braun et al. [10] proposed a preference guided register allocator. Like our *treescan*, it works on SSA. But unlike our approach, it processes the program using a linear ordering of the basic blocks. This ordering is defined by a complete cover of the program by traces. Moreover, it has to insert shuffle code at join point if all predecessors have not been proceeded, using ϕ -functions. Regarding coloring, it uses a new bias technique, the preference sets, that gives the liked and disliked colors for each variables. Overall, the preference guided allocator is more complex than our approach.

Coalescings In graph-coloring register allocation, many different coalescing techniques have been developed. They fall into three categories: aggressive, conservative, and optimistic coalescing. *Aggressive coalescing* removes as many

copies as possible, regardless of the colorability of the interference graph [15]. While it removes many copies, it may also increase the register demand of the program which potentially causes spilling. Since we never want to trade a spill for a copy, aggressive coalescing has to be used with caution. *Conservative coalescing* uses conservative tests [12, 20, 12, 20, 5, 8] that ensure that the chromatic number of the graph is not increased, before a copy is coalesced. *Optimistic coalescing* uses aggressive coalescing and de-coalescing if the k -colorability was violated [26, 27].

Biased coloring tries to remove copies by giving the source and the target of the move the same color in the first place. Chow and Hennessy rely on copy propagation to remove moves [17] in the priority-based allocator. Briggs et al. integrate biased coloring into graph-coloring allocation [12]. Mössenböck and Wimmer [35] use “register hints” in their linear-scan allocator to propagate copy information to the definition points of the variables. They gave also a technique based on register next use distances to assign caller-saved registers to local temporaries.

6. EXPERIMENTS

The algorithms described earlier in the paper were implemented in the back end of a production compiler developed by our industrial partner, STMicroelectronics for their commercial media processor based on the Lx architecture [18]. This static C compiler uses Open Source Version of the SGI Pro64 Compiler [19] (OPEN64) as the code generator, Linear Assembly Optimizer (LAO) as the register allocator, and OPEN64 for post-allocation optimization and assembly code emission. LAO can be used both in a static and dynamic compilation context. While the funding of those developments was motivated by dynamic compilation constraints, the industrial partner does not provide us with access to the dynamic compilation tool-chain. The target processor is 4-issue VLIW with 32 general-purpose registers, 8 of which are callee-save. Compared to IA32, for example, the [18] has relatively few register constraints. That being said, our results show significant improvements compared to allocators that do not effectively handle register constraints; consequently, the disparity is likely to be even greater in our favor for target architectures with more constraints.

Our experiments use a decoupled register allocation approach. The spilling algorithms used is described in [21]; the purpose of the experiments is to compare coalescers.

Our experiments use a subset of the Spec CINT2000 benchmarks; our compiler cannot handle *eon*, which is written in C++, and *gcc*, which requires a frame pointer that our compiler does not support.

6.1 Graph Coloring and Repairing

These experiments establish the efficacy of our approach to repairing on five different coalescing configurations:

- IRC: The IRC algorithm without live range splitting, but no repairing; this algorithm is not guaranteed to find a k -coloring of the interference graph, so it is allowed to spill, when necessary.
- Split: The IRC algorithm with live range splitting, but no repairing.
- Freeze, Conservative, and Dummy Nodes: The IRC algorithm without live range splitting but with repairing implemented as described in Section 2.2.

Figure 7 reports the normalized execution time of each benchmark, while Figure 8 reports the normalized number of vertices and number of edges of the interference graph for each benchmark. These numbers are geometric means over all functions of the benchmarks, because of the large number of functions that were compiled overall. The baseline approaches are IRC and Split. Between the two, Split produces better quality code (Figure 7), but with a noticeable increase in the size of the interference graph (Figure 8); in its favor, Split is the only existing technique that can deal with register constraints in a decoupled register allocation context. Our goal is to identify a coalescer that achieves the code quality of Split but without increasing the size of the interference graph.

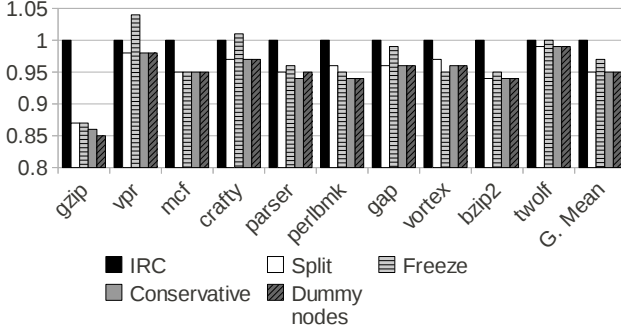


Figure 7: Execution time of generated code, normalized to IRC. Lower is better. In all but a handful of cases, the different repairing strategies reduced the execution time.

We compare IRC and Split against three approaches to handle negative affinities:

Dummy nodes is the naive approach to extend graph coloring to deal with negative affinities. It represents negative affinities using dummy nodes; as shown in Figure 7, this approach produces good quality code, but the dummy nodes that are added significantly increase the size of the interference graph; although Dummy Nodes is not shown in Figure 7, its interference graphs are larger than Split in virtually all instances. For this reason, we do not consider Dummy Nodes to be a realistic approach.

Freeze only considers negative affinities during the biased coloring phase as the end. As shown in Figure 7, the quality of code generated by Freeze is inferior to that of Split, Dummy Nodes, or Conservative. In terms of interference graph size, Freeze is comparable to IRC (Figure 8); the difference in size (due to a small number of negative-weighted affinity edges) is negligible, and is not shown in Figure 8.

Conservative converts negative affinities into interference edges using criteria similar to conservative coalescing. The quality of code generated by Conservative is comparable to Split, Dummy Nodes, and Freeze (Figure 7), while the size of the interference graph is comparable to that of IRC, and is not shown in Figure 8. Among the three negative-affinity-based coalescers considered here, Conservative is the only one to achieve the code quality of Split with an interference graph size comparable to IRC.

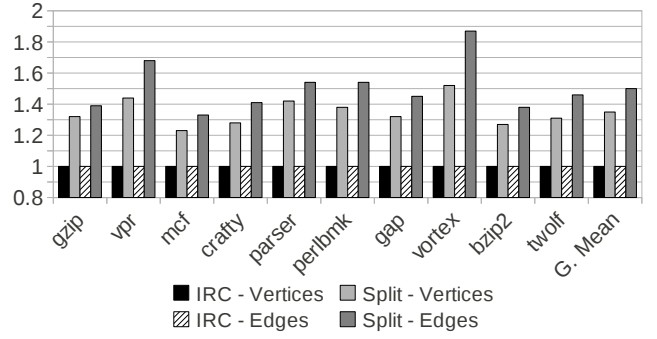


Figure 8: The normalized number of vertices and interference edges in the interference graph for each benchmark. IRC, Freeze, and Conservative have the same interference graph sizes, while Split’s interference graphs are noticeably larger. Dummy nodes is not a realistic solution, so we do not report its interference graph sizes, which would be large.

6.2 Treescan

This section evaluates the allocation time, i.e. the compile time dedicated to register allocation, and the number of copy operations that execute dynamically when coalescing is performed by the Treescan algorithm with different biased color assignment techniques, as discussed in Section 4.

6.2.1 Allocation Time

Figure 9 reports the normalized compile time of the different color assignment approaches. The compile times reported include all memory allocation/deallocation, structure initialization/destruction, and liveness analysis; however, they do not include the time required to translate out of SSA, which is not part of the coloring process. The reported runtime for each benchmark is the sum taken over all functions.

As expected, the introduction of Register Hints to bias the color assignment process during the treescan incurs a small average overhead of 1%, while Round Robin color assignment incurs an average overhead of 5%. Pre-coalescing comes at a higher price, 12% overhead for the Web strategy [14] and 31% for Aggressive coalescing [3]; Move Related coalescing costs an additional 6%. The most expensive technique, however, is Caller, whose overhead is 80%; this overhead is due to the data flow analysis required to compute liveness information and a traversal of the CFG to identify variables that are live across calls. Lastly, as a comparison point, we also report the allocation time of the treescan with a Split strategy, where all live ranges are split prior to constrained instructions [21]; the overhead of this technique is 76%, comparable to Caller.

On average, the baseline treescan runs 6.97 times faster than IRC, which respectively represents 5% and 38% of the whole compile time; in contrast, even the slowest-running variant of treescan has a runtime of less than 2 times than of the baseline version. For a JIT compiler, it is quite clear that treescan runs much more efficiently than register allocation based on graph coloring; next, we look at the quality of the code generated by the coalescers.

Note that by adding the techniques overhead, you get the

allocation time of the related composed method. For instance, caller plus web have composed overhead of $(1.8 - 1) + (1.11 - 1) = 0.91$ on average. Thus, this composed method is 1.91 times slower than the baseline.

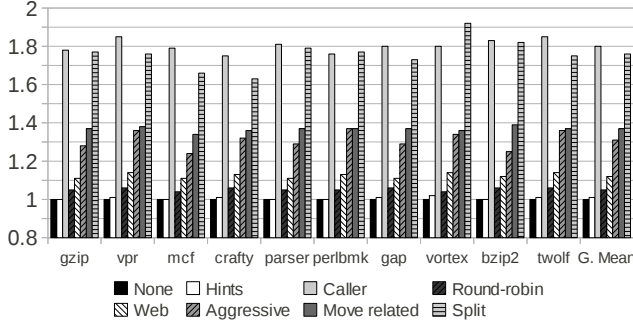


Figure 9: Treescan allocation time with biased coalescing. Lower is better.

6.2.2 Number of Dynamically Executed Copies

Figure 10 reports the number of dynamically executed copy operations that result from different combinations of color assignment enhancements to the treescan algorithm. We profiled each application to find the number of times each basic block executed [2]; for each copy operation occurring in basic block b , we use the profiling weight assigned to b to estimate the number of times the copy executes. This metric is architecture agnostic, as it ignores, for example, the possibility to hide the copies by scheduling them in parallel with one another or with other operations, or to schedule them in a branch delay slot. In Figure 10, we normalized the number of dynamically executed moves to the treescan implementation using register hints (H), as we consider this to be the most realistic baseline implementation for treescan, due to its low runtime overhead. Due to the large number of tested configurations, we only report the geometric means over all benchmarks.

The impact of the caller heuristic (HC) in isolation is minimal: the compiler inserts less repairing code, but fewer copies are coalesced. In many cases, two move-related variables cannot be coalesced because one crosses a call and the other does not; as we will see, the caller heuristic becomes more effective when combined with better coalescers.

Round-robin (HR) increases the number of dynamically executed copies by 21% because it always chooses the first available register; thus, it has a tendency to assign variables whose lifetimes cross procedure calls to caller-saved registers. Moreover, round-robin does not have any information about future uses of variables, e.g., as operands of ϕ s; consequently, the likelihood of eliminating the copies that result during SSA destruction is quite low. When combining round-robin with caller (HCR) the negative impact is reduced from 21% to 16%.

The techniques that employ pre-coalescing (HW, HA) perform quite well; web and aggressive strategies respectively reduce the number of dynamically executed copies by 13% and 14%. When combined with round-robin (HAR) and move-related (HAM), the negative impacts observed for the round-robin strategy, as described above, manifest them-

selves, but in a more limited fashion, as the pre-coalescer gives a better guide for assigning registers.

Combining the pre-coalescer and caller heuristic (HAC) is beneficial, because variables that are move-related to others that cross procedure call boundaries are biased using callee-saved registers; compared to H and HAC reduces the number of dynamically executed copies by 20%. Augmenting HAC with the round-robin strategy (HARC) achieves an additional 2% improvement. Similarly, combining the move-related heuristic with the caller heuristic and a pre-coalescer (HACM) achieves a 23% improvement.

Lastly, we wish to establish that pre-splitting is not necessary when using repairing; the best result achieved with pre-splitting (HARCS) reduces the number of dynamically executed copies by 24%, which is just a 1% improvement over HARC; in our opinion, the impact of pre-splitting on allocation time does not justify a 1% reduction in the number of dynamically executed copies in our JIT compiler.

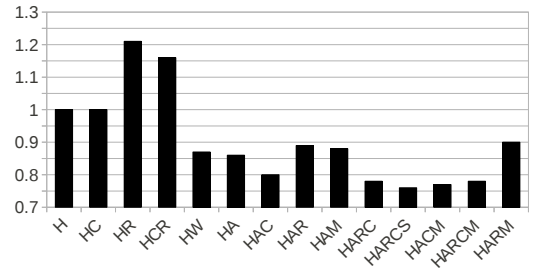


Figure 10: Geometric means of dynamic number of moves. Lower is better. Letters stand for the mix of the following configurations. H: Hints; R: Round-robin; C: Caller; M: Move related; A: Aggressive; W: Web; S: Split.

6.2.3 Run Time Performance

We compare the quality of the code generated by treescan using the different biasing techniques against the IRC algorithm. Figure 11 reports these results. The numbers can be compared to the results shown in Figure 7, since the base (IRC) is the same. Here too, because of the amount of configurations, we only give the geometric means.

First, all programs compiled with treescan are always as fast as their counterparts compiled with IRC. Although treescan is approximately seven times faster in allocation time than IRC.

The base treescan with register hints (H) generates code that is 1% faster than IRC. More surprisingly, with the additional caller heuristic (HC), code is as fast as the IRC. This is because of the VLIW processor we use for evaluation. When the caller heuristic is *not* active, repairing often occurs at call sites. However, at call sites there are usually enough free slots in the VLIW bundles to hide the repairing code. Hence, this repairing code comes for free. If the caller heuristic is *active*, the repairing move instructions occur at different places where they are no longer easy to hide because of saturated VLIW bundles.

Round-robin (HR) gives an additional percent of improvement. This benefit comes from the additional freedom for the post scheduler. On our machine, post scheduling is very

important because it places moves, spills, and reloads in unused slots of near bundles.

Using a pre-coalescing approach (HW, HA), treescan achieves 4% of improvement. This is almost as good as IRC with splitting shown in Figure 7. Combining these approaches with round robin (HAR) or caller heuristics (HAC), treescan gets an additional percent of improvements and is as good as the best graph coloring algorithms reported here. We achieve an additional percent by combining pre coalescing, the caller heuristic, and round robin (HARC, HACM).

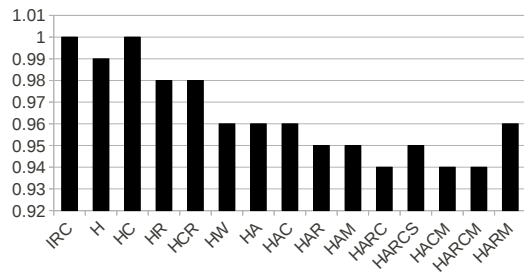


Figure 11: Geometric means of execution time of generated code. Lower is better. First, column, IRC, iterated register coalescing, other columns see caption of figure 10.

In summary, we draw the following conclusions: Register hints should be always used. Then, if there is a post-scheduling phase, round robin should be applied. Although it does not help coalescing, the post scheduler has more freedom and can hide more shuffle code in empty slots. On the other hand, it might increase the number of moves a little bit. Here, the choice has to be made dependent on the architecture. On our machine and our benchmarks, there were enough empty slots in the VLIW bundles to hide those additional moves. The benefit from relaxed post scheduling outweighed those extra copies.

Pre-coalescing has a non-negligible overhead but gives very good results and can improve other heuristics, too. This is the main source of treescan’s performance gain. The caller heuristic is quite expensive and gives bad results if used alone. It should be avoided, unless pre-coalescing is enabled. Together, they are more powerful in avoiding caller-saved registers for move-related variables that are live across calls. We show that splitting before coloring does not give any benefits in terms of run time. As it increases allocation time significantly, it should be avoided in the JIT context.

7. CONCLUSION

This paper has introduced repairing to handle register constraints during register coalescing. Repairing has been shown to be compatible with graph coloring-based coalescers and a new type of SSA-based coalescer called a treescan, that does not build an interference graph and improves significantly upon past linear scan allocators. Our evaluation has shown that a graph coloring coalescer that employs repairing can generate code whose quality is comparable to the most effective prior techniques that handle register constraints, but does so without enlarging any of the necessary auxiliary data structures. The treescan, moreover, runs more efficiently than the graph coloring-based coalescer with re-

pairing because it does not require an interference graph, while producing code of comparable quality. Consequently, we believe that a decoupled register allocation approach that uses two treescans, one for spilling, the other for coalescing, is the most reasonable implementation choice for JIT compilers.

8. REFERENCES

- [1] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’01)*, pages 243–253. ACM Press, 2001.
- [2] Thomas Ball and James R. Larus. Branch prediction for free. *SIGPLAN Notices*, 28(6):300–313, 1993.
- [3] Benoît Boissinot, Alain Darté, Benoît Dupont de Dinechin, Christophe Guillon, and Fabrice Rastello. Revisiting out-of-SSA translation for correctness, code quality, and efficiency. In *International Symposium on Code Generation and Optimization (CGO’09)*. IEEE Computer Society Press, March 2009.
- [4] Benoît Boissinot, Sebastian Hack, Daniel Grund, Benoît Dupont de Dinechin, and Fabrice Rastello. Fast liveness checking for SSA-form programs. In *CGO’08: proceedings of the sixth annual ieee/acm international symposium on code generation and optimization*, pages 35–44, New York, NY, USA, 2008. ACM.
- [5] Florent Bouchez. *A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases*. PhD thesis, ENS Lyon, France, apr 2009.
- [6] Florent Bouchez, Alain Darté, and Fabrice Rastello. On the complexity of register coalescing. In *CGO ’07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 102–114, Washington, DC, USA, mar 2007. IEEE Computer Society Press. Best paper award.
- [7] Florent Bouchez, Alain Darté, and Fabrice Rastello. On the complexity of spill everywhere under ssa form. *SIGPLAN Not.*, 42:103–112, June 2007.
- [8] Florent Bouchez, Alain Darté, and Fabrice Rastello. Advanced conservative and optimistic register coalescing. In *CASES’08: Proceedings of the 2008 international conference on Compilers, +Architectures and Synthesis for Embedded Systems*, pages 147–156, New York, NY, USA, 2008. ACM.
- [9] Matthias Braun and Sebastian Hack. Register Spilling and Live-Range Splitting for SSA-Form Programs. In *Compiler Construction 2009*, volume 5501 of *Lecture Notes In Computer Science*, pages 174–189. Springer, 2009.
- [10] Matthias Braun, Christoph Mallon, and Sebastian Hack. Preference-Guided Register Assignment. In *Compiler Construction 2010*, volume 6011 of *Lecture Notes In Computer Science*, pages 205–223. Springer, 2010.
- [11] Preston Briggs. *Register allocation via graph coloring*. PhD thesis, Rice university, April 1992.
- [12] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

- [13] Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*, June 2005.
- [14] Zoran Budimlić, Keith Cooper, Tim Harvey, Ken Kennedy, Tim Oberg, and Steve Reeves. Fast copy coalescing and live range identification. In *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation (PLDI'02)*, pages 25–32, Berlin, Germany, June 2002. ACM Press.
- [15] G. J. Chaitin. Register allocation & spilling via graph coloring. In *ACM SIGPLAN Symposium on Compiler Construction (CC'82)*, volume 17(6) of *SIGPLAN Notices*, pages 98–105, 1982.
- [16] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.
- [17] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(4):501–536, Oct. 1990.
- [18] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 203–213. ACM, June 2000.
- [19] G. Gao, J. Amaral, J. Dehnert, and R. Towle. The SGI Pro64 compiler infrastructure. In *Tutorial, International Conference on Parallel Architectures and Compilation Techniques*, 2000.
- [20] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.
- [21] Sebastian Hack. *Register Allocation for Programs in SSA Form*. PhD thesis, Universität Karlsruhe, October 2007.
- [22] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA form. In *International Conference on Compiler Construction (CC'06)*, volume 3923 of *LNCS*. Springer, 2006.
- [23] Allen Leung and Lal George. Static single assignment form for machine code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, pages 204–214. ACM Press, 1999.
- [24] Hanspeter Mössenböck and Michael Pfeiffer. Linear scan register allocation in the context of SSA form and register constraints. In *International Conference on Compiler Construction (CC'02)*, volume 2304 of *LNCS*, pages 229–246. Springer, 2002.
- [25] Rei Odaira, Takuya Nakaike, Tatsushi Inagaki, Hideaki Komatsu, and Toshio Nakatani. Coloring-based coalescing for graph coloring register allocation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 160–169, New York, NY, USA, 2010. ACM.
- [26] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT'98)*, pages 196–204. IEEE Press, 1998.
- [27] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 26(4), 2004.
- [28] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation by puzzle solving. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 216–226, New York, NY, USA, 2008. ACM.
- [29] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.
- [30] Hongbo Rong. Tree Register Allocation. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 67–77. ACM, 2009.
- [31] Vivek Sarkar and Rajkishore Barik. Extended linear scan: An alternate foundation for global register allocation. In *International Conference on Compiler Construction (CC'07)*, volume 4420 of *LNCS*, pages 141–155. Springer, 2007.
- [32] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288, New York, NY, USA, 2004. ACM.
- [33] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. *SIGPLAN Not.*, 33(5):142–151, 1998.
- [34] C. Wimmer and M. Franz. Linear scan register allocation on ssa form. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 170–179. ACM, 2010.
- [35] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *1st ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 132–141, 2005.