A Polynomial Spilling Heuristic: Layered Allocation

Boubacar Diouf, Albert Cohen, Fabrice Rastello

A Polynomial Spilling Heuristic: Layered Allocation

Boubacar Diouf
INRIA and École Normale Supérieure de Paris
Boubacar.Diouf@inria.fr

Albert Cohen
INRIA and École Normale Supérieure de Paris
Albert.Cohen@inria.fr

Fabrice Rastello
INRIA and École Normale Supérieure de Lyon
Fabrice.Rastello@ens-lyon.fr

Abstract

Register allocation is one of the most important, and one of the oldest compiler optimizations. It aims to map temporary variables to machine registers, and defaults to explicit load/store from memory when necessary. The latter option is referred to as spilling.

This paper addresses the minimization of the spill code overhead, one of the difficult problems in register allocation. We devised a heuristic, polynomial approach called *layered*. It is rooted in the recent advances in decoupled register allocation. As opposed to conventional incremental spilling, our method incrementally allocates clusters of variables. We demonstrate its quasi-optimiality on standard benchmarks and on two architectures.

Categories and Subject Descriptors D.3.4 [*Programming languages*]: Processors — Compilers, Optimization

General Terms Compilers, Algorithms, Performance

Keywords Register allocation, compilers

1. Introduction

Register allocation is an important compiler optimization. Its goal is to map temporary variables in a program to either machine registers or memory locations. Register allocation is subdivided into two sub-problems: first, the *allocation* selects the set of variables that will reside in registers at each point of the program; then, the *assignment* or *coloring* picks a specific register where a variable will reside. Usually, all the variables of code cannot reside in registers. Variables not held in registers should reside in memory, these variables are called *spilled variables*. The *spilling* problem [6, 15] decides which variables should be stored in memory to make the assignment possible; it aims at minimizing load/store overhead. The *coalescing* [5] and *alienation* (when repairing is enabled [11]) problems aims at minimizing the overhead of moves between registers. Spilling and coalescing are correlated problems and are classically solved in the same framework. Live-range splitting (adding register-to-register moves) to reduce register pressure is sometimes considered in such a framework [12], but it is very hard to control the interplay between spilling and splitting or coalescing.

Building on the properties of the static single assignment form (SSA), it is now possible to decouple the allocation from the assignment. Indeed, the interference graph of a program in SSA form is a chordal graph [20]. Since coloring a chordal graph is easy, it follows that the assignment problem is also easy. Finding a valid coloring whenever it exists can thus be solved optimally with a greedy, linear algorithm on chordal graphs, called *tree-scan* [11]. It follows the spirit of the linear-scan [22], applied to the dominance tree instead [24]. Thus, performing register allocation under SSA has led to new approaches where the remaining difficult problems, spilling and coalescing, are treated separately. This *decoupled* approach is advocated by Fabri [14], Appel and George [2], and Hack [4, 6, 8, 20].

Existing spilling heuristics rely on a sufficient condition to guarantee register assignment, and incrementally spill until the condition holds. For programs under SSA, the condition is necessary and sufficient: MAXLIVE, the maximal number of variables simultaneously live at a program point, has to be lower than or equal to R. Existing spilling rely on incremental spilling decisions to satisfy this condition, but these decisions tend to be overly local and suboptimal. Indeed, incremental spilling is NP-complete [6], and heuristics based upon it trade too much their optimality for polynomiality.

In contrast to incremental spilling, we propose to adopt the symmetric approach: incremental allocation. Intuition for it emerges from two observations allowing for more global spilling decisions:

1. Register allocation is pseudo-polynomial in the number of registers [6], suggesting a heuristic that solves (optimally) roughly R over *step* allocation problems on *step* registers each. The final allocation being the layered of the stepwise allocations.

CGO'13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

2. Stepwise optimality does not guarantee an overall optimal allocation, but we will show that it comes very close to optimal, even with *step* $= 1$. Intuition for this comes from recent work by Diouf et al. [13], observing that allocation decisions tend to be a monotonic function of the number of registers.

This paper proposes a new graph-based allocation heuristic, based on the maximum clique cover to define the profitability of spilling variables. It exploits the pseudo-polynomial complexity in the number of registers of the allocation problem under SSA—as opposed to the symmetric, spilling problem which remains strongly NP-complete. It addresses the spill-everywhere problem in a decoupled context and also presents an extension to non-decoupled approaches. It introduces *layered allocation* a new strategy that incrementally allocates variables instead of incrementally spilling variables. The evaluation performed on standard benchmarks shows that this new approach is quasi-optimal.

The outline of the paper is as follows. Section 2 details the rationale for our new approach. Section 3 surveys the important concepts and results upon which our approach is built. Section 4 presents our layered allocator for SSA programs. Section 5 adapts this scheme into a non-optimal heuristic for general, non-chordal interference graphs. Section 6 evaluates the algorithm. Section 7 discusses related work and Section 8 concludes the paper.

2. Our Approach to the Spilling Problem

We propose here a graph-based approach to the spilling problem under SSA. We first give the motivation of a graph-based approach and explain why we think that the spill everywhere problem is a relevant one. We then detail the two key motivations of our approach, i.e. the pseudo-polynomiality of the allocation problem and the quasi-optimality of the stepwise allocation scheme.

2.1 A Graph-Based Approach to Spilling

Apart from allowing the design of more efficient coalescing heuristics [7], the main advantage of the decoupled approach concerns the spilling problem: checking if the register pressure, MAXLIVE, is low enough is much simpler than checking the colorability of a general graph. This observation has led several researchers to design *program-based* heuristics to lower register pressure, opposing the new decoupled approach to the "old" (interference-) *graph-based* spilling heuristics. The goal of this section is to compare both approaches by making the distinction between two aspects of the spilling decision: usefulness and profitability.

First, a spilling decision should be *useful*, in other words help the assignment problem. Let us illustrate this point, using the example of Figure 1 with 3 registers. Here, spilling variable a_2 is useless in helping the assignment problem anyhow: while the upper part of the interference graph is not 3-colorable ($\{d, e, f, g\}$ require 4 registers), the lower part is.

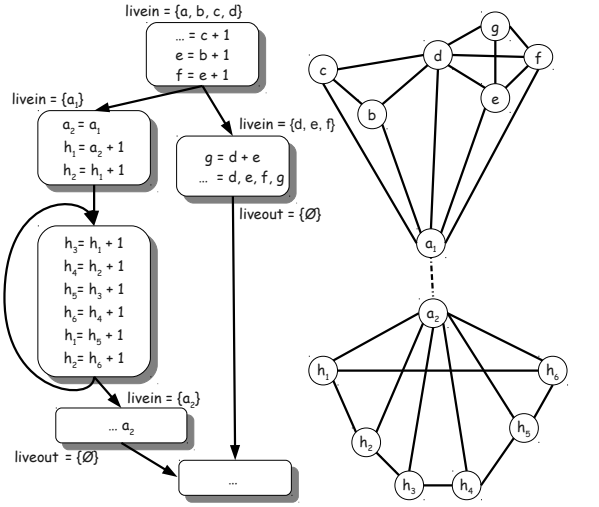

Figure 1. Program-based vs. graph-based spilling.

Taking only useful decisions when manipulating a program is straightforward: a simple heuristic consists in considering program points one after another, and when the register pressure at the current point is too high, incrementally spilling some variables to lower it. Here, every program point of the left-hand part of the control flow graph has register pressure less than 3, so none of the variables there have to be spilled. Checking this directly on the graph, is a priori quite hard, and degree guided heuristics are used in practice: it would wrongly spill a_2 because of its high degree. Actually, a very elegant graph-based approach, proposed by Pereira and Palsberg [21] exists. It uses a maximal clique cover, polynomial for chordal graphs. For a program under SSA, spilling a variable is useful [20] if it is part of a clique of size greater than the number of available registers; for non-SSA graphs this characterization shows to be reasonably accurate. This is especially interesting as pure SSA form might not always be possible in practice. In other words, a graph-based approach that uses the notion of (almost [1]) maximum clique cover to drive the spilling of variable would still work in this more general context.

Among a set of "useful variables", which one is the most *profitable* to be spilled first? To illustrate this point, let us consider again the example of Figure 1, but suppose a_1 and a_2 have been coalesced into a variable named a. At the entry of the region, at least one of the four variables that are simultaneously live here should be spilled. It is clearly more profitable to spill a or d which live-ranges include the live-ranges of b and c. On a single basic-block under SSA, the furthest first strategy of Belady [3, 6] is clearly optimal but the existing generalization [8] to a general control flow graph does not work well: it would wrongly consider a (that tra-

[1] if not chordal

verses a loop) to be much more profitable than d. On the other hand, a graph is very well suited for computing and updating profitability, an inductive function of the spilling cost and the probability it avoids the spilling of other variables. Here d belongs to two maximal cliques of size greater than 3, while a belongs to only one. In other words graphs enable more naturally the incremental update of profitability along the successive removal (spilling) of nodes (variables) from the graph.

The last motivation toward designing a graph-based approach is because, as we will see below, a graph allows to tackle the problem from different points e.g. by starting spilling the most profitable variable. Equivalently, a program-based approach would start by program points inside inner loops which leads to a sophisticated implementation [19].

2.2 Why Spill Everywhere?

The spilling problem can be considered at different granularity levels: the highest, so called *spill everywhere*, corresponds to considering the live range of each variable entirely. A spilled variable will then lead to a store after the definition and a load before each use. Of course, in practice, if the variable can stay in a register between two consecutive uses, a load is saved. The finest granularity, so called load-store optimization, corresponds to optimize each load and store separately. The latter, also known as paging with write back, is NP-complete [15] on a basic block, even under SSA form. The spill-everywhere problem is much simpler, applicable to just-in-time compilation, and many instances are polynomial under SSA form [6]. The algorithms we propose can be applied to both spill everywhere and load-store optimization problems. We focus here on the former for its simplicity, because our past experience summarized in the 4 following points tends to confirm the practical effectiveness of the spill everywhere problem:

1. The complexity of the load-store optimization problem comes from the asymmetry between loads and stores. Also, most SSA variables have only one or two uses in practice, and the cost of the store favors spilling the entire live range instead of two sub-ranges of different variables.
2. The queuing mechanism present in most architectures behave like a small, extremely fast cache. But it is highly sensitive to the number of simultaneously spilled variables.
3. In the other extreme situation where stores have no cost, a variable can be considered to be either in memory or in register but not in both. Such a formulation [2] is strictly equivalent to a spill everywhere formulation where live ranges are split at every use.
4. Last, a solution to the spill-everywhere problem gives to a load-store optimization problem the global view lengthily discussed so-far that existing heuristics lack. In other words a spill-everywhere solution can play the role of an oracle.

2.3 Allocation Instead of Spilling

After giving the reasons that support our work on the spill everywhere problem, let us stress the difference we want to make here between spilling and allocation. Spilling aims at finding which variable to evict from registers while allocation aims at finding which variable to keep in registers. Of course, one is the dual of the other, so conceptually spilling and allocation are the same. Now suppose you have a set of variables and you want to evict (spill) a minimum amount of them such that MAXLIVE is lowered by just one. As shown in [6] this problem is NP-complete even for the simplest SSA program instance. On the other-hand, consider you have already a set of allocated variables and you aim at allocating a maximum number of additional ones such that at every program point the register pressure LIVE is increased by at most 1. Then as outlined before, this problem is, under SSA, polynomial with a complexity of $\mathcal{O}(\Omega n)$. Ω being the maximum simultaneously live variables that remains to be allocated; n being the size of the program. Hence, in a way allocation is simpler than spilling. Our approach pushes this distinction further: conceptually, every variable is initially in memory, and we evaluate the *gain* of allocating a given one instead of considering every variables to be initially in a virtually unbounded register file and evaluate the *cost* of evicting it. As we will see in this paper, this allows to be much more accurate concerning the modeling of gain/cost that accounts for ABI and register constraints.

2.4 Stepwise Allocation is Close to Optimal

In a recent paper, Diouf et al. [13] studied the question of the *spill sets inclusion*, which is the question to know whether or not the variables spilled on an optimal allocation with R registers ($R > 0$) is included in the set of variables spilled on an optimal allocation with $R - 1$ registers. Diouf et al. showed that, theoretically, the answer to the question of spill sets inclusion is no, but they experimentally validated that when varying the number of registers from R_{min}, the minimum number of registers to enable code generation, to the number of registers allowing to allocate all the variables, the inclusion property holds for 99.83% of the SPEC JVM98's methods. From the spill sets inclusion, it is straightforward to see that the variables allocated on an optimal allocation with $R - 1$ registers are included in the set of variables allocated when R registers are available. Thus, the spill sets inclusion proves, empirically, that the stepwise allocation is close to optimal.

3. Background

We now summarizes some definitions and results on graphs and chordal graphs upon which our approach is based.

In the rest of this paper, we assume that an estimated spill cost has been computed for each variable. A spill cost

represents the access frequency of a variable, it is high when the variable is frequently accessed and low when it is not. We denote R the number of available registers.

Programs are usually represented as graphs, within graph coloring frameworks, and live sets within linear scan frameworks. Thus the spilling problem is naturally solved over these two representations. Our approach is compatible for both representations, but in the rest of this section we will focus on the graph representation.

3.1 Graphs and Weighted Graphs

A *graph* $G = (V, E)$ consists of two sets, V the set of vertices or nodes, and E the set of edges. Every edge (v_1, v_2) of E has two end points $v_1 \in V$ and $v_2 \in V$. We say that v_1 and v_2 are *adjacent(s)* or are *neighbor(s)* if $(v_1, v_2) \in E$. The number of neighbors of a vertex v is called the *degree* of v. Here, We only consider *undirected* graphs, i.e., we do not make difference between the edges (v_1, v_2) and (v_2, v_1).

A sequence of vertices $[v_0, v_1, v_2, \ldots, v_l, v_0]$ is called a *cycle* of length $l + 1$ if $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq l$ and $(v_l, v_0) \in E$. A subset $A \subseteq V$ is called a *clique* of G if every two distinct vertices of A are adjacent. A clique A is *maximal* if it is not properly contained in any other clique of G. A clique is *maximum* if there is no clique of G of larger cardinality. A vertex v of a graph G is *simplicial* if its neighbors form a clique in G. In contrast to a clique, a *stable* or an *independent set* is a subset $S \subseteq V$ that does not contain two vertices that are adjacent.

Assuming each vertex v of $G = (V, E)$ is associated with a non-negative number $w(v)$, the weight of a subset $S \subset V$ is expressed as:

$$w(S) = \sum_{v \in S} w(v)$$

The graph G associated with the function w is called a weighted graph and denoted G_w. For example, on Figure 2 each vertex labelled with its corresponding variable is weighted by a number written close to it (e.g. vertex with f has a weight of 6).

From a graph representation of a program, the allocation problem becomes equivalent to the problem of finding a colorable sub-graph of maximum weight. This problem is NP-complete on arbitrary graphs (as coloring is NP-complete).

3.2 Chordal Graphs

A code is in static single assignment (SSA) form when every scalar variable has only one textual definition in the program code. Most compilers use a particular SSA form, the strict SSA form, with the additional so-called dominance property: given a variable, any path from the entry of a program to one of its uses goes through its (unique) definition. One of the useful properties of such a form is the live ranges of the variables (delimited by the definition and the uses of a variable) can be viewed as sub-trees of this dominance tree. The intersection graph of these sub-trees of the dominance tree represents the interference graph. An important result of graph theory states that the intersection graph of a family of sub-trees of a tree is a chordal graph [17]. It follows that the interference graph of a program in SSA form is chordal.

A graph G is *chordal*, if every cycle of length four or more has a chord, a chord being an edge joining two vertices of the cycle, that are not consecutive. The graph given in Figure 1 shows a non-chordal graph (circuit $[h_1, \ldots, h_6, h_1]$ has no chord) while Figure 2 shows a chordal graph, for instance the cycle $[c, d, f, e, c]$ has a chord which is (d, e).

Algorithm 1 MAXIMUMWEIGHTEDSTABLE

Require: $[v_1, \ldots, v_n]$: list of vertices indexed using a PEO
Require: *w*: a map associating to each vertex its weight
Require: *adj*: a map associating to each vertex the list of its neighbors
Var: *marked_red*: a (LIFO) list keeping track of vertices marked red
Var: *marked_blue*: a list keeping track of vertices marked blue
1: **for** $i = 1 \rightarrow n$ **do**
2: **if** $w(v_i) > 0$ **then**
3: add v_i to *marked_red*
4: **for all** $v_j \in adj(v_i)$ for $j > i$ **do**
5: $w(v_j) \leftarrow w(v_j) - w(v_i)$
6: **end for**
7: **end if**
8: **end for**
9: **while** *marked_red* $\neq \perp$ **do**
10: $v \leftarrow$ pop the first element of *marked_red*
11: Add v to *marked_blue*
12: remove all the vertices of *adj(v)* from *marked_red*
13: **end while**
14: **return** *marked_blue*

An interesting property, shown by Frank in [16] and used below in this paper, is that computing the maximum weighted stable of a chordal graph can be done in $\mathcal{O}(|E|+|V|)$. The algorithm uses the notion of perfect elimination order. An ordering $v_1, v_2, \ldots, v_n$ of the vertices of a graph G is a *perfect elimination order* (PEO) if each v_i is a simplicial vertex in $G_{\{v_i, v_{i+1}, \ldots, v_n\}}$, the graph remaining from G when all the vertices preceding v_i in the ordering have been removed. It is well known that a graph is chordal if and only if it has a perfect elimination order (PEO) [17]. For instance $[a, f, d, e, b, g, c]$ is a PEO of the chordal graph given in Figure 2.

Algorithm 1 computes the maximum weighted stable of a weighted graph G_w. It supposes the n vertices of G_w to be indexed along a perfect elimination order; the adjacency list of each vertex is provided through the map *adj*; finally the weight is given by the function w. As it traverses the list of vertices along the PEO (lines 1 to 8), it overwrites the weight of the upcoming neighbors ($w(v_j)$ for $v_j \in adj(v_i)$, $j > i$) by reducing it by the current vertex weight ($w(v_i)$ with v_i the current vertex). Any vertex that becomes negatively weighted is bypassed. A list *marked_red*, is used to keep track of vertices that stay positively weighted, and is then traversed along the reverse order. This traversal (lines 9 to 13) allows to greedily fill up the *marked_blue* list of non-interfering nodes (stable) of maximum weight.

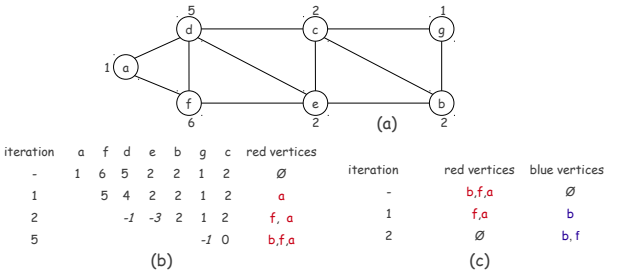

Figure 2. Maximum weighted stable with Algorithm 1.

Figure 2(b) and Figure 2(c) depicts the general steps of Algorithm 1 when applied to the graph given in Figure 2(a). Figure 2(b) shows how the set of vertices marked red is constructed. The column *iteration* presents the iterations of the first for-loop of Algorithm 1 that marks red vertices. The second column keeps track of the updated values of w. The vertices are ordered according to the given perfect elimination order. The last column shows how the set of marked red evolves. The first row shows, before the beginning of the loop, the values of w for each vertex and the set of vertices marked red which is empty. At the first iteration, the weight of a is 1, thus a is marked red and the weights of its neighbors d and f are decreased by 1. At the second iteration, the weight of f is 5. Thus, f is marked red and the weights of its neighbors a, d and e are decreased by 5. As the weight of d and e become negative, the next iterations will skip them. Finally, we obtain the set of vertices marked red which are composed of b, f, a.

Figure 2(c) runs the while-loop of Algorithm 1 that builds the blue list. At the first iteration, the vertex b (the latest inserted) is popped and is inserted in *makred_blue*. At the second iteration the vertex f is popped and inserted in *makred_blue*. The vertex a is adjacent to f and cannot be added to the set of vertices marked blue. Thus, a is removed from the list of vertices marked red. We then end up with a set of vertices marked blue composed of f and b of overall maximum weight 8.

4. Layered-Optimal Register Allocation

This section restricts to the spilling problem for SSA programs. General graphs will be handled in Section 5.

Based on the two observations explained on Section 2, we present here our solution which solves the spill minimization problem for R registers by stacking optimal allocations (layers) of simpler sub-problems on few registers. Each of this simpler problem is considered to have no more than *step* available registers. Stepwise optimality does not guarantee an overall optimal allocation, but we will show that it comes very close to optimal, even with $step = 1$.

Algorithm 2 implements layered-optimal allocation. It takes as input *candidates*, the list of variables that are candidates to register allocation. It then returns as result *alloc_list*, the list of variables that have been allocated with R registers. Algorithm 2 calls OPTIMALALLOCATION which returns the *optimal allocation set* with *step* registers minimizing the spill cost among the variables that have not yet been allocated (currently in *candidates*). This set is added to *alloc_list* and removed from *candidates*. In its last step, Algorithm 2 finds the set of variables that minimizes the spill cost among the variables remaining in *candidates*.

Algorithm 2 LAYEREDOPTIMALALLOCATION

Var: *candidates*: the list of vertices that are candidate to an allocation
Var: *alloc_list*: the list of so far allocated variables
1: *count* $\leftarrow 0$
2: **while** *candidates* $\neq \perp \wedge$ *count* $< R$ **do**
3: *step* $\leftarrow \min(R - count, step)$
4: *result* $\leftarrow$ OPTIMALALLOCATION(*candidates*,*step*)
5: add every vertex of *result* to *alloc_list* and remove it from *candidates*
6: *count* $\leftarrow$ *count* $+$ *step*
7: **end while**
8: **return** *alloc_list*

When the interference graph is chordal, for a *step* of one, OPTIMALALLOCATION reduces to finding the maximum weighted stable set, and can be implemented with Algorithm 1. For $step \geq 2$, OPTIMALALLOCATION can be implemented through dynamic programming [6]. In the following, we restrict ourselves to a step of one. The complexity of the layered allocator is then $\mathcal{O}(R(|V| + |E|))$.

Algorithm 2 is a solid basis for an incremental allocation, but it can be improved in two ways: biasing the cost of the variables, and iterating further on the set of allocated variables until no more variables can be allocated.

4.1 Biasing the Weights

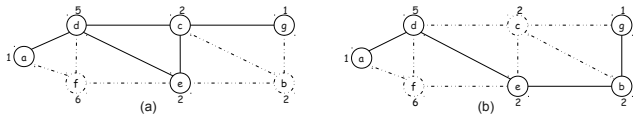

Figure 3. The benefit of biasing the weight.

Before we explain how we bias weights of vertices, let us first have a look on Figure 3. We assume two registers, a *step* of one, and we are looking for the set of variables to allocate for the weighted graph G_w given in Figure 2(a). G_w has two stable sets $\{b, f\}$ and $\{c, f\}$, represented using dashed lines in Figure 3, that have a maximum weight 8. If $\{b, f\}$ is chosen, at the next step, Algorithm 2 will look for the maximum weighted stable set on the graph remaining when b and f are removed from G_w. This graph is represented by the plain nodes and edges in Figure 3(a). The maximum weighted stable set for this graph is $\{d, g\}$ with a weight of 6. This leads to spilling variables a, c and e with a spill cost of 4. In contrast, if we choose c and f, at the next step the maximum weighted stable of the graph, shown in black nodes and edges in Figure 3(b), will be $\{b, d\}$ of weight 7. This leads to a spill cost of 3.

This example shows that the choice among different maximum weighted stable sets has an impact on the next iterations of Algorithm 2. As opposed to the "old" incremental spilling heuristic that spills variables one by one, the layering approach allocates variables layers by layers. Thanks to that, it stresses less the importance of profitability but does not eliminate it. Our proposal is to consider the maximum weighted stable set that removes the most interferences in the graph on non-allocated variables as the most profitable one. To achieve this, the weight of vertices considered to be made of positive integers, is biased with its degree using the following new weight function w':

$$w'(v) = w(v) \times |V| + d^o(v)$$

where $|V|$ is the number of vertices of the graph and $d^o(v)$ the degree of v. For two vertices u and v, as $d^o(u) \leq |V|-1$, the two following properties will always be verified:

$$\begin{cases} \text{if } w(u) < w(v) \text{ then } w'(u) < w'(v) \\ \text{if } w(u) = w(v) \text{ then } w'(u) \leq w'(v) \text{ iff } d^o(u) \leq d^o(v) \end{cases}$$

4.2 Iterating to Fixed Point

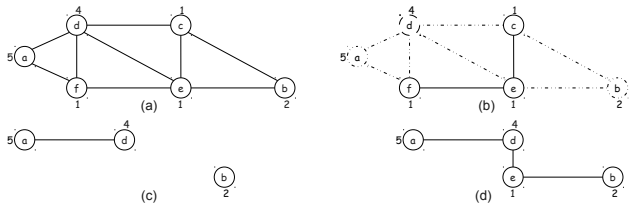

Figure 4. The benefit of iterating until a fixed point.

To illustrate the second improvement, let us again consider an example which weighted graph G_w is depicted in Figure 4(a), and let us assume two registers and a *step* of one. G_w has four maximal cliques $\{a, d, f\}$, $\{b, c, e\}$, $\{c, d, e\}$, and $\{d, e, f\}$. Figure 4(b) shows in dashed lines the set $\{a, b, d\}$ of allocated variables returned by Algorithm 2. The sub-graph of allocated variable is reported in Figure 4(c). Algorithm 2 ends up with this set of allocated vertices whether or not the weights are biased. Let us recall that a coloring with R colors is possible on a chordal graph if the maximum clique of the graph does not have more than R vertices. If we focus on the vertex f, we notice that in the graph G_w, f belongs to a maximal clique composed of a, d and f, which have 2 vertices already allocated. Thus vertex f cannot be added to the graph of allocated vertices without adding a clique of size 3 and then making a coloring with 2 colors impossible. Unlike f, the vertices c and e are not contained in a maximal clique that has 2 vertices already allocated. It follows that either c or e can be added to the graph of allocated vertices: by adding e, the resulting graph, shown in Figure 4(d) remains 2-colorable.

Algorithm 3 FIXEDPOINTLAYERED (FPL)

Require: *candidates*: the list of vertices that are candidate to an allocation
Var: *alloc_list*: the list of variables allocated so far
Var: *allowed_cliques*: the list of cliques that do not have more than R allocated vertices, it is initialized to the list of maximal cliques
Var: *alloc_per_clique*: a map associating to each maximal clique the number of its allocated vertices

```
1: count ← 0
2: while candidates do
3:     result ← OPTIMALALLOCATION(candidates, 1)
4:     if result = ∅ then
5:         break
6:     end if
7:     add every vertex of result to alloc_list
8:     remove every vertex of result from candidates
9:     count ← count + 1
10:    if count ≥ R then
11:        UPDATE(candidates, result, allowed_cliques, alloc_per_clique)
12:    end if
13: end while
14: return alloc_list
```

Algorithm 4 UPDATE

Require: *candidates*: the list of vertices that are candidate to an allocation
Require: *result*: a list of newly allocated variables
Var: *allowed_cliques*: the list of cliques that do not have more than R allocated vertices
Var: *alloc_per_clique*: a map associating to each maximal clique the number of its allocated vertices

```
1: for all clique ∈ allowed_cliques do
2:     if clique ⋂ result ≠ ∅ then
3:         increment alloc_per_clique(clique) by one
4:         if alloc_per_clique(clique) ≥ R then
5:             remove all the vertices of clique from candidates
6:             remove clique from allowed_cliques
7:         end if
8:     end if
9: end for
```

Figure 4 shows that even after allocated R maximal layers of thickness 1, more variables can still be allocated. Finding such variables can be done through the cost of maintaining a set of not already saturated maximal cliques. This task is performed by Algorithm 4 that Algorithm 3 calls once the first R layers have been stacked. Algorithm 4 increments, for each freshly allocated vertex, the number of allocated vertices of each clique which contains it. If a clique is saturated i.e. has *R* of its vertices already allocated, it is removed from *allowed_cliques* (the list of none saturated cliques), and all its composing vertices are removed from *candidates*. On the example of Figure 4(b) after the first two iterations, $\{a, d, f\}$ is saturated (so f is removed from *candidates*), $\{b, c, e\}$, $\{c, d, e\}$, and $\{d, e, f\}$ are not. *candidates* is then restricted to vertices $\{c, e\}$. If e is allocated it saturates all remaining allowed clique, the loop ends.

4.3 Chads

When a variable is spilled, it does not completely disappear from the interference graph. It is replaced by a set of short-lived variables (called chads in [6]) which must be taken

into account. For instance on RISC architectures, memory can only be accessed through load and store instructions: before using a variable say v spilled at address a, the value of v must be loaded from address a into a register. The extra instructions inserted to reload spilled variables form the *spill code*.

Some approaches—like the JikesRVM implementation of the linear scan—spill locally an allocated variable when there is no free register to assign a reloaded variable. On CISC architectures like the x86, we also can take advantage of complex addressing modes to get operands directly from memory (at most one such operand on x86). On the other hand, graph coloring heuristics iteratively rebuild interferences after spill. Symmetrically, interferences can be updated after each new allocation.

5. Layered-Heuristic Allocator

Although the spill minimization problem is only pseudo-polynomial on SSA programs, the method also applies to general programs. The layered approach remains applicable, but Algorithm 1 can not be used anymore as the graphs are not chordal. The layered-heuristic algorithm that we present in this section, uses almost the same general scheme, i.e. stacks layers of allocated variables. The first difference is that, obviously, each layer is not guaranteed to be maximal: stables are built using a greedy heuristic. The second difference, is that instead of stacking layers as they are built, a set of layers is first built, as if the number of registers were infinite; then layers are sorted using decreasing weight so as to allocate the first R of largest weight. The solution advocated for building each (sub-)maximal weighted stable is extremely simple: the set grows iteratively using the non-interfering vertex of maximum weight.

Algorithm 5 SUBMAXIMUMWEIGHTEDSTABLESLIST

Require: *candidates*: a list of vertices, ordered by decreasing weight
Require: *adj*: a map associating to each vertex the list of its neighbors
Var: *stable_list*: a list of stables
1: **while** *candidates* $\neq \emptyset$ **do**
2: *stable* $\leftarrow \perp$
3: // add all the vertices of candidates to potentials
4: *potentials* $\leftarrow$ *candidates*
5: **while** *potentials* $\neq \emptyset$ **do**
6: remove from *potentials* its first vertex, called *v*
7: add *v* to *stable*
8: remove all the vertices of *adj*(*v*) from *potentials*
9: **end while**
10: add *stable* to *stable_list*
11: remove all the vertices of *stable* from *candidates*
12: **end while**
13: **return** *stable_list*

Algorithm 5 computes *stable_list* a set of sub-maximum weighted stables that cover the graph. It starts from *candidates*, a list of vertices of a graph sorted by decreasing weight. A new stable is built at each iteration of the outer while-loop. In order to compute *stable*, the new stable, all the variables still in *candidates* are added to *potentials*. The list *potentials* keeps tracks, at each round of the inner while-loop, the vertices that do not interfere with the vertices already into *stable*. Every time a vertex *v* is added to *stable*, all the neighbors of *v* are removed from *potentials*. At the end of the inner while-loop, the computed *stable* is added to the *stable_list*, and the next round of the outer while-loop starts. Algorithm 5 ends when every variable is in a stable.

After the stables have been computed, Algorithm 6 decides which stables should be allocated to registers. The R stables of maximum weight (the weight of stable being the sum of the weighted of its vertices) are allocated.

Algorithm 6 LAYEREDHEURISTICALLOCATION

Require: *candidates*: a list of vertices that are candidate to an allocation
Require: R: the number of available registers
stable_list $\leftarrow$ SUBMAXIMUMWEIGHTEDSTABLESLIST(*candidates*)
sort *stable_list* by decreasing weight
if $|$*stable_list*$| > R$ **then**
spill each variable in the last ($|$*stable_list*$| - R$) stables from *stable_list*
end if

The worst-case complexity of the layered-heuristic allocation is $\mathcal{O}(\Delta \times (|V| + |E|))$ were Δ is the maximum degree of the interference graph. Indeed the algorithm iterates at most $\Delta+1$ times, the built of a single stable set visits every neighbor of a node only once. As the experiments will show, while non restricted to SSA form programs, this extremely simple heuristic turns out to be quite efficient in practice providing weights are biased as explained in Section 4.1. On the other hand, one should outline that the fixed point improvement described in Section 4.2 cannot be applied for this method.

6. Experimental Evaluation

Our approach is very well suited to SSA programs. We also present excellent results on arbitrary interference graphs from non-SSA programs.

6.1 Chordal Graphs: SSA Programs

Methodology We evaluated our approach on chordal interference graphs extracted from the Open64 compiler using the MinIR intermediate representation [18]. We consider two different target processors: the ST231 VLIW processor and the ARM Cortex A8 (ARMv7). For the former, we generated the interference graphs for the *SPEC CPU 2000int*, the *lao-kernels* (an internal suite from STMicroelectronics) and the *eembc* benchmarks. We only used the *lao-kernels* for the ARMv7 instruction set.

For each of the considered benchmarks, we computed the spill costs based on the basic blocks's frequency and on the number of accesses to the variables within the basic blocks. We studied the impact of the register count, ranging from 1 to 32. For each instance of the register allocation problem and for each configuration, we compared the following algorithms:

GC The Chaitin-Briggs, optimistic graph coloring [9].

Optimal An optimal ILP-based allocator.

NL The naïve layered-optimal allocation method implemented without the two improvements presented in Section 4.

FPL The layered-optimal allocation method with the fixed point improvement presented in Section 4.2.

BL The layered-optimal allocation method with the biased weights presented in Section 4.1.

BFPL The layered-optimal allocation method including both fixed points and biased weights improvements.

Results and discussion The results obtained from the evaluation of chordal graphs generated from SPEC CPU 2000int, lao-kernels and eembc benchmarks are very similar.

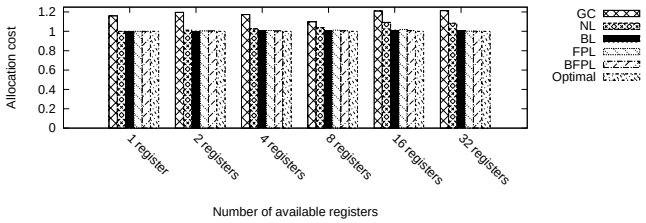

Figure 5. Allocation cost for SPEC CPU 2000int on ST231.

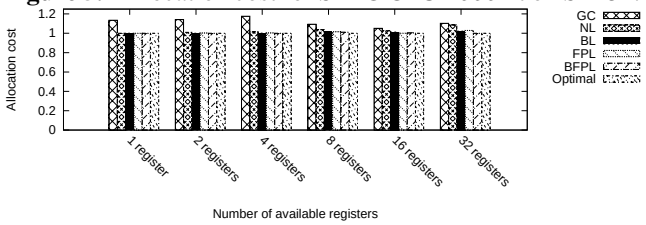

Figure 6. Allocation cost for EEMBC on ST231.

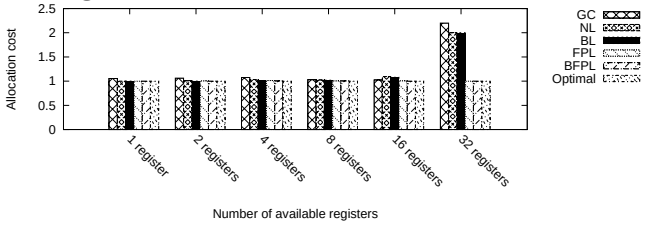

Figure 7. Allocation cost for lao-kernels on ARMv7.

Figure 5 presents the average of the allocation cost of all the application of the SPEC CPU 2000int. For the sake of exposition, we reported here the results for configuration with a register count of 1, 2, 4, 8, 16 and 32 registers. For all the configuration, BL, FPL, BFPL are close to optimal on average and are better than GC. On configuration with register counts up to 8, BL is also quasi-optimal, but for configuration with 16 and 32 registers, we notice a performance degradation. This is reinforced by Figure 6 and on Figure 7 we also notice a performance degradation, when the register count is 32, of the FPL approach; it suggests that the biased improvement is very helpful on the lao-kernels benchmark suite, which is made of small benchmarks and thus can be more impacted by a bad allocation choice.

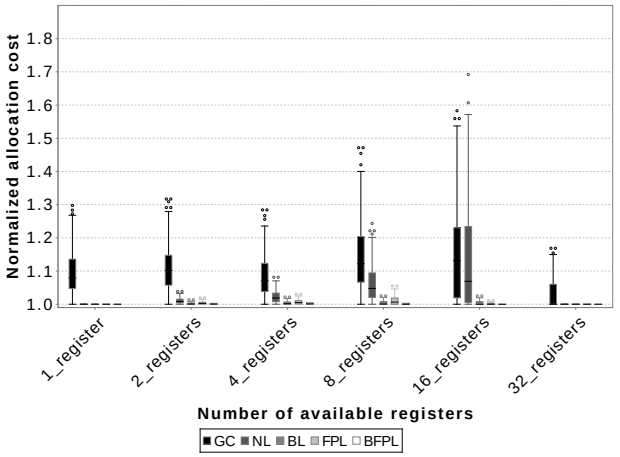

Figure 8. Cost distribution over SPEC CPU 2000int on ST231.

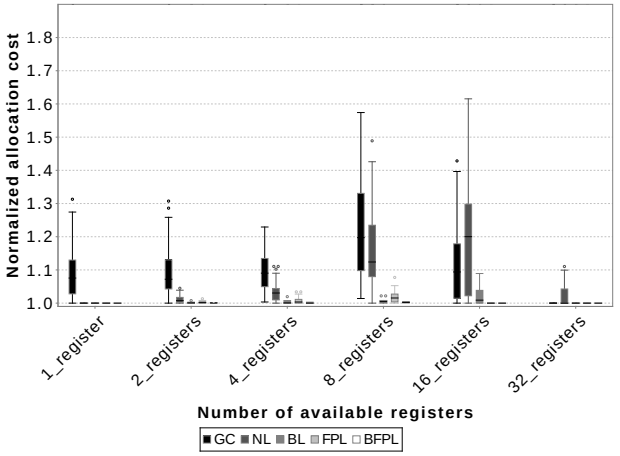

Figure 9. Cost distribution over EEMBC on ST231.

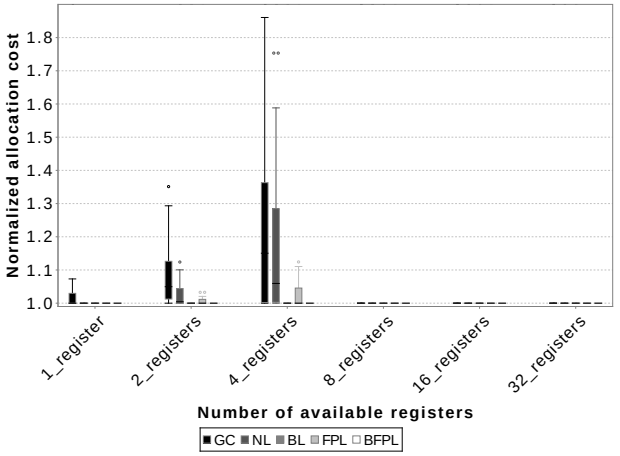

Figure 10. Cost distribution over lao-kernels on ARMv7.

Figure 8 studies the variability of allocation across individual interference graphs from the SPEC CPU 2000int, targeting the ST231. Each allocation result is normalized to the optimal allocation for the specific benchmark. This figure depicts the distribution of these normalized allocation

costs. GC, and to a lesser extent NL, show a high variability. This indicates that some benchmarks yield poor allocations for these allocators. On the contrary, BL, FPL and BFPL are consistently successful at computing close-to-optimal allocations. This is confirmed by Figures 9 and 10 on the other benchmark suites. Notice a slight variability for FPL and registers on the lao-kernels targeting the ARMv7 instruction set (Figure 10).

6.2 Extension to Non-Chordal Graphs

Methodology We evaluated our approach on general, non-SSA programs, studying the *SPEC JVM 98* benchmark suite (a benchmark set to measure the performance of Java virtual machines). We used the JikesRVM [1] just-in-time compiler; its intermediate representation is not in SSA, and the interference graphs are not chordal in general.

We considered different configurations of register count going from 2 to 16. For each instance of the register allocation problem and for each configuration, we compared the following algorithms:

LS The linear scan algorithm as implemented in JikesRVM.
BLS A variant of the linear scan relying on Belady's furthest-first strategy to make spilling decisions if their costs are close enough according to a chosen threshold.
GC The Chaitin-Briggs, optimistic graph coloring.
Optimal The globally optimal allocation implementing an ILP model proposed by Diouf et al. [13].
LH Our layered-heuristic algorithm.

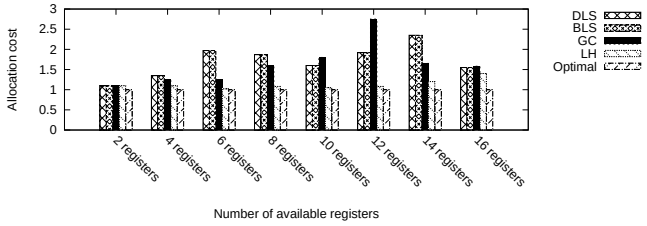

Figure 11. Evaluation of layered-heuristic allocation.

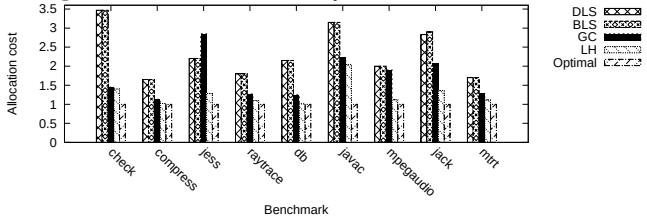

Figure 12. Layered-heuristic allocation for 6 registers.

Results and discussion Figure 11 shows the allocation costs for all SPEC JVM together, normalized over the cost of the optimal allocation's cost. Configurations with different register counts going from 2 to 16 registers. For almost all the register counts, the layered-heuristic allocator is close to optimal, except for the configurations with 14 and 16 registers. This can be explained by the accumulation of approximations in the incremental construction of maximal weighted stables, a consequence of the non-chordality of the interference graphs.

Figure 12 reports for each individual benchmark the normalized allocation costs when we have a register count of 6 registers. We see that here again, the layered-heuristic allocator performs close to optimal allocations, and outperforms all the other allocation heuristics. For `check`, `jess`, `javac`, and `jack`, the overhead can reach 60% of the optimal, but the cost is still better than the conventional heuristics.

7. Related Work

Register allocation algorithms often rely on spilling algorithms to perform spill minimization.

In static compilation the dominant approach to register allocation is the graph coloring in which the spilling and coloring (assignment) algorithms are interleaved. During the *simplify phase*, whenever all the remaining nodes have at least R degrees, a node needs to be spilled or pushed onto the stack (optimistic coloring) and removed from the graph. A natural intuition is to choose a node that has a low spilling cost and which interferes a lot. Many of the graph coloring variant are based on this intuition and use the quantity $cost(v)/deg(v)$ — a global information over the whole program — to choose the variables to spill [10].

In just-in-time (JIT) compilation, (quasi-)linear complexity remains a driving force in the design of optimization algorithms. The linear scan which is one of the most used register allocation algorithm on JIT compilers has a worst case complexity[2] of $\mathcal{O}(n \times R)$, where n is the number of variables in the program and R is the number of available registers on the target architecture. The original spilling heuristic used in linear scan [22] is based on the Belady's furthest first algorithm [3]. This algorithm relies on *local information* to perform spilling: "At a point p where registers are not enough to hold all the live variables, spill the variables whose live ranges go farther in the future". Recent versions of linear scan use more elaborate algorithms based on variables' spill cost estimation and sharing some of the global spilling decisions of graph coloring [25].

The idea of improving the spill minimization in a decoupled approach has been explored by Proebsting and Fischer [23], and by Braun and Hack [8]. Braun and Hack generalized the Belady's furthest first algorithm — which works very well on straight-line code — to control-flow graphs. Their approach, while being applicable as a pre-spill phase in any compiler, is more adapted to SSA-based register allocation. They reported a reduction in the number of reload instructions by 54.5% compared to the linear scan and by 58.2% compared to the graph coloring. Pereira and Palsberg rely on maximal cliques to drive spilling decisions with similar goals on chordal graphs, and generalizability to general

[2] Notice that this complexity does not take into account the complexity of the liveness analysis.

graphs [21]. Like the latter approaches, layered allocation is fast and can be used in a non-decoupled context for general programs, in a decoupled context for SSA programs, and as a pre-spill phase in any compiler. Unlike Braun and Hack, we show that our layered algorithm performs close-to-optimal allocations.

8. Conclusion

Combining key observations in SSA-based, decoupled register allocation, we designed a new, polynomial approach to the spill-cost minimization problem: layered allocation. Our method contrasts with decades of work on register allocation by incrementally allocating clusters of variables to registers, while conventional heuristics incrementally spill variables. The criterion to form these clusters, rooted in the maximal clique problem (polynomial on chordal graphs), is also original. Our algorithm produces allocations that are very close to optimal on SSA programs, outperforming higher complexity heuristics such as the graph coloring methods. We also adapt our method to design an allocation heuristic for general, non-SSA programs.

These fundamental results pave the way to a simpler and very effective register allocation framework. Several steps remain to be taken to integrate it in a production compiler: studying the interactions with register coalescing and other downstream optimizations, studying load/store optimization variants (with transparent, fine-grain live range splitting), and reducing the number of incremental allocations to compete with the slightly faster linear scan allocators.

Acknowledgments

We would like to thank Taj Muhammad Khan for his comments and help proofreading the paper.

References

[1] B. Alpern and et al. The Jikes RVM project: Building an open source research community. *IBM Systems Journal*, 44(2):399–418, 2005.

[2] A. W. Appel and L. George. Optimal spilling for CISC machines with few registers. In *PLDI'01*, pages 243–253, Snowbird, Utah, USA, June 2001.

[3] L. A. Belady. A study of replacement algorithms for virtual storage computers. *9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1966.

[4] F. Bouchez, A. Darte, and F. Rastello. On the complexity of register coalescing. In *CGO'07*, 2007.

[5] F. Bouchez, A. Darte, and F. Rastello. On the complexity of register coalescing. CGO '07, pages 102–114, Washington, DC, USA, 2007. IEEE Computer Society.

[6] F. Bouchez, A. Darte, and F. Rastello. On the complexity of spill everywhere under ssa form. In *LCTES'07*, pages 103–112, 2007.

[7] F. Bouchez, A. Darte, and F. Rastello. Advanced conservative and optimistic register coalescing. In *CASES'08*, pages 147–156, 2008.

[8] M. Braun and S. Hack. Register spilling and live-range splitting for ssa-form programs. volume 5501 of *LNCS*, pages 174–189. Springer Berlin / Heidelberg, 2009.

[9] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[10] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–105, New York, NY, 1982. ACM.

[11] Q. Colombet, B. Boissinot, P. Brisk, S. Hack, and F. Rastello. Graph-coloring and treescan register allocation using repairing. In *CASES'11*, pages 45–54. IEEE Computer Society, Oct. 2011.

[12] K. D. Cooper and L. T. Simpson. Live range splitting in a graph coloring register allocator. In *CC'98*, pages 174–187. Springer-Verlag, 1998.

[13] B. Diouf, J. Cavazos, A. Cohen, and F. Rastello. Split register allocation: Linear complexity without the performance penalty. In *Intl. Conf. HiPEAC'10*, LNCS, Pisa, Italy, Jan. 2010. Springer-Verlag.

[14] J. Fabri. Automatic storage optimization. In *ACM Symp. on Compiler Construction*, pages 83–91, 1979.

[15] M. Farach-Colton and V. Liberatore. On local register allocation. *J. of Algorithms*, 37(1):37–65, 2000.

[16] A. Frank. Some polynomial algorithms for certain graphs and hypergraphs. *Proceedings of the Fifth British Combinatorial Conference*, pages 211–226, 1975.

[17] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 2004. ISBN 0444515305.

[18] J. L. Guen, C. Guillon, and F. Rastello. Minir: a minimalistic intermediate representation. In *Workshop on Intermediate Representations (WIR'11)*, Chamonix, France, 2011.

[19] S. Hack. *Register Allocation for Programs in SSA Form*. PhD thesis, Universität Karlsruhe, October 2007.

[20] S. Hack, D. Grund, and G. Goos. Register allocation for programs in SSA-form. In *CC'06*, pages 247–262, 2006.

[21] F. M. Q. Pereira and J. Palsberg. Register allocation via coloring of chordal graphs. In *APLAS*, pages 315–329, 2005.

[22] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999.

[23] T. A. Proebsting and C. N. Fischer. Probabilistic register allocation. PLDI '92, pages 300–310, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9.

[24] H. Rong. Tree register allocation. In *International Symposium on Microarchitecture (MICRO'09)*, pages 67–77. IEEE Computer Society, Dec. 2009.

[25] V. Sarkar and R. Barik. Extended linear scan: An alternate foundation for global register allocation. In *CC'07*, volume 4420 of *LNCS*, pages 141–155. Springer, 2007.