Tree Register Allocation

Hongbo Rong
Microsoft Corporation
hongbor@microsoft.com

ABSTRACT

This paper presents tree register allocation, which maps the lifetimes of the variables in a program into a set of trees, colors each tree in a greedy style, which is optimal when there is no spilling, and connects dataflow between and within the trees afterward. This approach generalizes and subsumes as special cases SSA-based, linear scan, and local register allocation. It keeps their simplicity and low throughput cost, and exposes a wide solution space beyond them. Its flexibility enables control flow structure and/or profile information to be better reflected in the trees.

This approach has been prototyped in the Phoenix production compiler framework. Preliminary experiments suggest this is a promising direction with great potential. Register allocation based on two special kinds of trees, extended basic blocks and the maximal spanning tree, are found to be competitive alternatives to SSA-based register allocation, and they all tend to generate better code than linear scan.

Categories and Subject Descriptors

D.3.4 [**PROGRAMMING LANGUAGES**]: Processors—*Compilers, Optimization*

General Terms

Algorithms, Experimentation, Languages, Theory

Keywords

Register Allocation, Chordal Graph

1. INTRODUCTION

Register allocation maps the *lifetimes* of the variables in a given program to the physical registers of the underlying architecture. *Local register allocation* focuses on a *basic block*, while *global register allocation* handles an entire *procedure*. Both local and global register allocation have been extensively studied[2, 3, 4, 5, 6, 7, 8, 12, 13, 14, 16, 19, 20, 21, 23, 29, 30]. They are generally considered among the most important optimizations in a compiler.

Global register allocation has long been simplified as a problem of coloring an *interference graph* [3, 6]. An interference graph is an *intersection graph of the lifetimes*, where a node represents a lifetime, and an undirected edge is connected between any two nodes whose corresponding lifetimes are simultaneously live at some program point. Allocating k registers to the lifetimes is then reduced to a coloring problem: Given k colors, can the interference graph be colored such that no two adjacent nodes have the same color?

In general, this graph coloring problem is NP-complete, as the interference graph can be any undirected graph [6]. However, when the interference graph is chordal, it can be colored in the minimum colors in time linear to the number of nodes and edges, with an algorithm known as *Maximum Cardinality Search (MCS)* [28], which is to be described in Section 3.2.

A graph is *chordal* if every cycle with more than three nodes has a *chord*, which is an edge that is not part of the cycle, but connects two nodes in the cycle [11, 19]. For example, Fig. 1(a) is a chordal graph, where there is one cycle *abcd*, which has a chord *bd*. Removing the chord makes the graph no longer chordal (Fig. 1(b)). We can see from Fig. 1(a) that in a chordal graph, a cycle is divided into triangles by the chord(s), and thus a chordal graph is also figuratively called *triangulated graph*.

Because a chordal graph can be colored optimally in linear time, chordal graph-based register allocation, especially the *Static Single Assignment (SSA)*-based register allocation, has recently attracted a lot of interest. It has been found that when a program is in SSA form [10], the interference graph must be chordal [4, 12, 19]. It should be pointed out here that linear scan [18, 20, 22, 23, 29] and the classical local register allocation [8], are based on chordal graph as well: in both cases, every lifetime must be a straight line, for which the interference graph is an *interval graph*, a sub-class of chordal graph [1].

If we observe carefully, in all of linear scan, SSA-based, and local register allocation, *a lifetime has a shape of tree*: it spans a sub-tree of a tree that is composed of basic blocks. For instance, for the program represented by the control flow graph in Fig. 2(a), linear scan orders the basic blocks sequentially in Fig. 2(b), which can be regarded as a 1-ary tree. It can be seen that for the two variables a and b, their lifetimes are two straight lines, each spanning a subtree. Alternatively, SSA-based register allocation organizes the basic blocks as the dominator tree in Fig. 2(c), where

MICRO'09, December 12–16, 2009, New York, NY, USA.

variable a has one lifetime, while variable b has 3 disjoint lifetimes [1]; each of the four lifetimes spans a subtree of the dominator tree.

It is not a coincidence that in these chordal graph-based approaches, a lifetime spans a subtree of a tree. In fact, it is the fundamental requirement for them. According to Gavril [11],

Gavril's theorem *For a graph G, the following conditions are equivalent: (i) G is a chordal graph; (ii) G is the intersection graph of a family of subtrees in an undirected tree, where a node represents a subtree, and an undirected edge is connected between any two nodes when their corresponding subtrees overlap.*

In the context of register allocation, consider an interference graph. It is *the intersection graph of a family of lifetimes*. According to the above theorem, the interference graph is chordal, if and only if it is also *the intersection graph of a family of subtrees* in a tree. Therefore, it is chordal, if and only if every lifetime is a subtree. By "a lifetime is (or spans) a subtree", one can imagine that the lifetime flows uninterruptedly through some paths of a tree. The strict definition is to be given in Section 2. So Gavril's statements can be translated as follows:

Re-interpretation of Gavril's theorem *For an interference graph G, the following conditions are equivalent: (i) G is a chordal graph; (ii) every lifetime spans a subtree in a tree.*

Considering the context of register allocation, the tree should be composed of the basic blocks of the program.

Based on the above observation, this paper proposes a new global register allocation approach, namely *tree register allocation*. Its central idea is to "treefy" the lifetimes of the variables, so that the interference graph must be chordal and therefore the greedy MCS algorithm can be applied to allocate the minimum number of registers to the lifetimes. It has three steps:

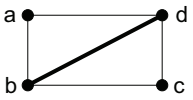
1. *Tree formation.* The lifetimes are mapped into a set of trees. The total number of the trees, and each tree, can be arbitrarily defined in theory. *The only requirement is that in each tree, a lifetime must span a subtree.*

2. *Coloring.* For each tree, the interference graph for the lifetimes in it must be chordal. Thus the lifetimes can be colored in an MCS-based algorithm.

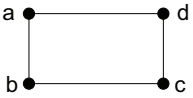
3. *Connecting dataflow.* If the lifetimes for the same variable are allocated different registers between or within the trees, the registers may need to be permuted on some control flow edges to maintain the dataflow of the variable.

This approach has the following advantages:

[1]The essence of SSA is that for a variable, each static definition of it is distinguished from the others. That is, each static definition has an individual lifetime. Fig. 2(c) has 3 definitions of variable b: two are explicit, the other one is not shown, but can be imagined to be at the start of block $B4$ as a ϕ instruction to merge the values of the two explicit definitions.



(a) A chordal graph. The cycle *abcd* has a chord *bd*, which is highlighted in bold line.



(b) The graph becomes non-chordal after removing the chord.

Figure 1: Illustration of chordal graph.

1. Generality. This approach represents a family of chordal graph-based register allocation methods. It generalizes and subsumes as special cases the recent SSA-based, linear scan, and the classical local register allocation methods [2, 4, 8, 12, 20, 23, 29]. In this way, it exposes the deep connection between these seemingly different approaches, and shows they are of the same class.

   Intuitively, in SSA-based register allocation, the lifetimes are mapped to the dominator tree of the program. See Fig. 2(c). In linear scan (and local) register allocation, the lifetimes are mapped to a 1-ary tree. See Fig. 2(b). The 1-ary tree for local register allocation contains only one basic block.

2. A wide solution space. As explained above, the existing SSA-based, linear scan and local register allocation ensure that the interference graph must be chordal, by "treefying" the lifetimes either over a dominator tree, or over a 1-ary tree.

   The paper clarifies that the only condition to get a chordal interference graph is to have every lifetime span a subtree of a tree. The tree is far from unique. The 1-ary tree and the dominator tree are only two special cases. We point out, theoretically and experimentally, that they are by no means the only choices, nor necessarily the best choices. This provides further research a broad horizon beyond the existing methods.

   Theoretically, in the re-interpretation of Gavril's theorem, the second condition only requires the lifetimes span subtrees of a tree, but does not say how exactly the tree is defined. For a tree with n nodes, there can be n^{n-2} number of distinct trees (Cayley's formula [26]). Experimentally, it is to be shown in Section 4 that a tree, including a dominator tree and a 1-ary tree, can have certain structural limitations that lead to inferior coloring in some cases.

3. Control flow structure and profile information can be better reflected in the tree(s).

   In general, it is a challenge how to construct the tree(s) to achieve the best possible allocation results. Linear scan arranges the basic blocks into a 1-ary tree in the

reverse postorder [20], which loses much of the control flow structure of the program. SSA-based register allocation organizes the basic blocks into the dominator tree of the program, which summarizes the control flow structure in terms of *dominance* relationship between the blocks and thus maintains the control flow structure to greater extent, but the dominator tree's edges do not necessarily correspond to the most frequently executed control flow edges in the program.

This paper has experimented a few ways to expose some useful heuristics. Experimentally, it shows that with the flexibility in constructing the tree(s), we can exceed linear scan by better reflecting the control flow structure in the tree(s), which introduces "control flow sensitivity" to register allocation to some extent; we can also go beyond the SSA-based register allocation by considering profile information in building a tree.

4. Simplicity and throughput advantage. This approach still keeps the simplicity and the throughput advantage of the previous chordal graph-based register allocation [4, 12, 19, 20, 23, 29]. The simplicity can be seen from these aspects: (1) A chordal graph can be colored in the minimum colors in linear time using the MCS algorithm. (2) This minimum number of colors is precisely the size of the largest clique in the graph, where a *clique* is a sub-graph in which any two nodes are connected by an edge. For this reason, spilling might be performed beforehand such that the size of any clique is not more than the number of available registers. After that, it is guaranteed that there will be no more spilling in coloring. In short, spilling and coloring can be separated.

   The approach also enjoys the advantage in compiler throughput: An interference graph does not need to be explicitly built in order to perform coloring. Instead, the coloring algorithm simply processes the lifetimes in temporal order along each path of a tree. This is in contrast with the general graph-coloring approach [6], where it is essential but expensive to construct and manipulate an interference graph, as the time it takes dominates the overall time of allocation [9].

A prototype of this approach has been implemented inside the Phoenix production compiler framework. To show the flexibility, generality, and potential benefits of the approach, the experiments will demonstrate several kinds of trees, each representing a different register allocation method. They are by no means the only viable choices, though. These trees include basic block trees (BBs), extended basic block trees (EBBs), 1-ary tree, dominator tree, and maximal spanning tree. The 1-ary and dominator tree correspond to linear scan and SSA-based register allocation, respectively. Register allocation based on EBBs and the maximal spanning tree are newly proposed here.

Preliminary experiments on SPEC2006 and SPEC2000 have confirmed the feasibility of this approach, and suggest this is a new direction with great potential. They show that register allocation based on EBBs and the maximal spanning tree can be competitive alternatives to the current SSA-based register allocation; and all these three register allocation methods tend to generate better code than linear scan with similar compile time. Analysis of the results exposes some limitations of the current SSA-based and linear scan methods. It shows the importance of use profile information and control flow structure in register allocation, and due to the flexibility of tree building in this approach, they can be better encoded in the structures of the trees.

We assume that the instructions of the program have been generated and their execution order has been fixed. Thus there should be no confusion with code generation of expression trees, which decides the execution order of the instructions that minimizes register pressure [24, 25].

2. BASIC CONCEPTS

This section reviews the basic concepts on control flow graph, liveness, and tree. It then defines the concept of "a lifetime spans a subtree".

For a program, a *Control Flow Graph (CFG)* is a graph representation of all the paths that may be traversed by the program during execution. A node of the CFG represents a *basic block*, which consists of a sequence of instructions. The nodes are connected via *control flow edges*. Fig. 2(a) shows the control flow graph for a simple program.

The CFG has a unique entry node. A node A *dominates* another node B if every path from the entry to node B must go through node A. A *dominator tree* can be constructed from the CFG, where a parent dominates all its children. For example, in the CFG in Fig. 2(a), block $B1$ dominates all other blocks, because any path reaching any of the other blocks must pass $B1$. Consequently, the dominator tree is like that in Fig. 2(c).

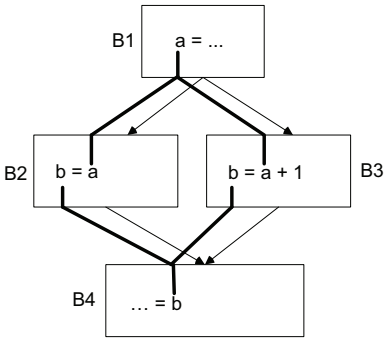
A *tree* is a hierarchical data structure, where any node has one and only one *parent* node, except the *root* node, which has no parent. Every node can have zero or more *children* nodes. The tree is an *n-ary tree* if every node has no more than n children. For instance, the trees in Fig. 2(b), 2(c), and 2(d) are 1, 3, 2-ary trees, respectively.

An *Extended Basic Block (EBB)* is a maximal sub-graph of the CFG such that (1) it is a tree; and (2) except for the root node, any other node has one and only one incoming control flow edge in the CFG. Fig. 2(e) shows two EBBs for the CFG in Fig. 2(a). Each of them is a tree.

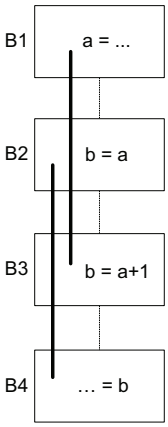
A *spanning tree* of the CFG is a tree composed of all the nodes and some of the edges of the CFG. If a control flow edge is associated with a weight, which is the frequency the edge is passed during the execution of the program, then the spanning tree has a weight equal to the sum of the weights of its edges. A *maximal spanning tree* has the maximal weight than any other spanning tree. An example spanning tree is shown in Fig. 2(d).

Often algorithms working on the CFG visit the nodes in some order. In a *reverse postorder*, also called *depth-first order* [20], the nodes in the CFG are ordered such that a node is before any of its control flow successors, except when a successor is reached via a back edge. Fig. 2(b) illustrates that linear scan register allocation arranges the basic blocks in the reverse postorder. In this order, every block is before any of its control flow successor. For example, $B1$ is before both of its control flow successors $B2$ and $B3$.

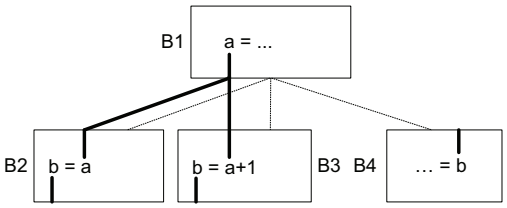
For register allocation, the lifetimes of the variables are identified through liveness analysis. A *program point* is the point immediately before or after an instruction in the program. A variable is *live* at a program point if the value of the variable at that point is to be used in future during the execution of the program. The *lifetime*, or *live range*, of the variable represents all the program points where the variable



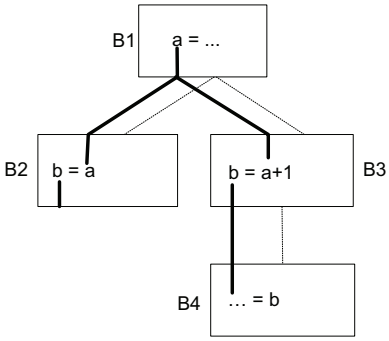
(a) Control flow graph



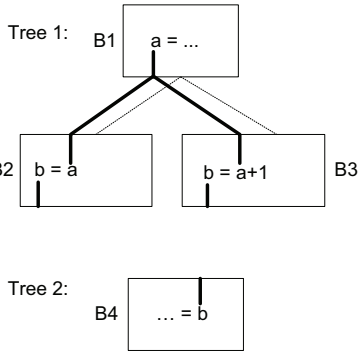
(b) A 1-ary tree (Linear scan)



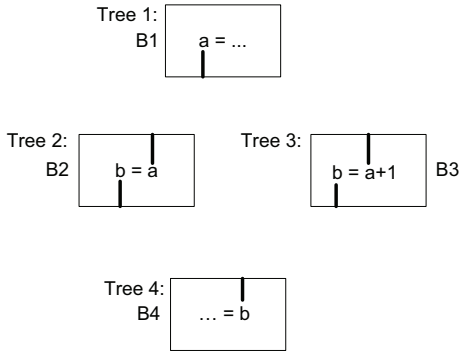
(c) The dominator tree (SSA-based register allocation)



(d) A spanning tree



(e) EBB trees



(f) BB trees

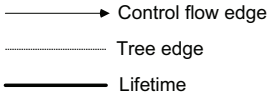


Figure 2: Illustration of a CFG and some trees built from it.

is live.

A lifetime spans a subtree in a tree of basic blocks, if (1) the lifetime is in the tree, i.e., any program point it includes is before or after an instruction in a basic block in the tree; and (2) the lifetime is connected. That is, any program point it includes is adjacent to another program point it includes. Two program points P_1 and P_2 are *adjacent* to each other if (a) P_1 is immediately before, and P_2 is immediately after, the same instruction in a basic block; or (b) P_1 is immediately after the last instruction of a basic block B, and P_2 is immediately before the first instruction of a child block of B in the tree.

From Fig. 2(b) to 2(f), all the lifetimes over the trees are shown in bold lines. In any tree, a lifetime spans a subtree.

3. SOLUTION

The solution is simple. It has three steps: tree formation, coloring, and connecting dataflow. The overall algorithm is shown in Fig. 3(a). We describe the solution in more detail below.

3.1 Tree Formation

In this step, the lifetimes are treefied. The basic blocks, along with the lifetimes in them, are distributed into a set of trees. Theoretically, the number of the trees can be arbitrary. The parent-child relationship within a tree can be arbitrary as well: any block can be the parent of another block. So a tree edge does not necessarily have any correspondence with a control flow edge.

In one extreme, every block can be a tree. For example, see Fig. 2(f). In another extreme, all the blocks form a tree. For example, see Fig. 2(b), 2(c) and 2(d). The decision will affect not the correctness of the approach, but the code quality.

With the distribution of the basic blocks, a variable may have more than one lifetime, each spanning a subtree. They are treated as separate lifetimes. For example, in Fig. 2(c), variable b has 3 lifetimes, each spanning a different subtree; in Fig. 2(d), variable b has 2 lifetimes.

It should be pointed out that when a variable has more than one lifetime, it is always feasible to conservatively extend the lifetimes such that they connect into a single lifetime. Again, this will affect not correctness but code quality. The classical linear scan method does this extension [20].

3.2 Coloring

In this step, the lifetimes in each tree are colored using an MCS-based algorithm. Each tree is processed in the same way independently.

The algorithm is shown in Fig. 3(b). It traverses a tree path by path. For the lifetimes along each path, it scans them in the order of start time. The algorithm scans each lifetime exactly once. So it has time complexity of $O(l)$, where l is the total number of lifetimes in the tree. The algorithm minimizes the number of registers needed, when there is no spilling. Spilling and other optimizations are to be considered in Section 3.7.

3.3 Connecting Dataflow

As said in Section 3.1, after tree formation, a variable may have more than one lifetime, each being colored separately. If they happen to be assigned different registers, but there is dataflow between them along some control flow edges in the CFG, some transfer instructions may need to be inserted along such edges so that each lifetime gets the correct dataflow input.

In general, the process is as follows: examine each control flow edge. Gather any variable that is live out of the predecessor basic block of the edge, and is live into the successor basic block of the edge, with two different registers. For all such variables, the registers assigned to them in the predecessor block should be permuted, along the edge, so that the values are transferred to the successor block correctly. This is a well-understood common procedure in the literature. The reader is referred to [23] or [12] for more details.

3.4 Illustration

Let us illustrate the approach with the example program in Fig. 4(a). There are 5 variables, whose lifetimes are shown in bold lines. Every variable has one and only one lifetime. An interference graph can be constructed straightforward, as shown in Fig. 4(b). This graph has a cycle with 5 nodes, without any chord. So it is not a chordal graph.

Now let us apply tree register allocation for the program. Assume we have chosen to build the maximal spanning tree. Suppose the left branch of basic block $B1$ in Fig. 4(a) is less frequently executed than the right branch. Then the maximal spanning tree is shown in Fig. 4(c).

We have required that every lifetime must span a subtree of the tree. Therefore, variable c has two lifetimes, one is in the left part of the tree, the other in the right. Let us call them "c left" and "c right", respectively. Then the interference graph looks like that in Fig. 4(d). There is no cycle now. So the graph becomes chordal.

Now let us color the lifetimes. Assume we have two registers available, r_1 and r_2. The coloring process scans the two paths in the tree, one by one. See Fig. 4(e). First, scan the left path, which is composed of blocks $B1$ and $B2$. In the first block $B1$, the lifetime of variable a appears, and we assign it register r_1. Then we process the next block $B2$. The lifetime of variable b appears, which takes the next free register r_2. Then the lifetime of "c left" appears. At that time, variable a is dead and its assigned register r_1 is freed. This register can then be assigned to "c left". The left path is finished.

In the same way, we scan the right path, and assign registers to the lifetimes appearing along that path. Note that we do not have to process B_1, which has already been processed above. Instead, we start directly from the next block in the path, $B3$.

After the coloring process ends, all lifetimes have been colored, as shown by the annotations besides the lifetimes in Fig. 4(e). Note that "c left" and "c right" have been assigned two different registers: r_1 and r_2, respectively.

Fig. 4(f) shows the final program with the register allocation results. Variable c lives from block $B2$ to block $B4$. However, it is assigned register r_1 in block $B2$, but another register r_2 in block $B4$. To connect the dataflow, a new basic block $B5$ is inserted between the two blocks, and an instruction

$$c(r_2) = c(r_1)$$

is placed in the block.

3.5 Relation with Other Methods

In linear scan [20], all the basic blocks are ordered sequentially. This sequence can be viewed as a 1-ary tree, where

```
ALLOCATION(program, registers):
1:  Construct trees for program
2:  for each tree t do
3:      Active ← {}
4:      R      ← registers
5:      COLOR(ROOT(t), Active, R)
6:  Connect dataflow
```

Notation:
program: a given program
registers: the set of registers
ROOT(t): the root basic block of tree t

(a) Tree register allocation

```
COLOR(Block, Active, R):
1:  for each lifetime l in increasing order of start time,
2:          l ∉ Active and l starts from Block, do
3:      for each lifetime l' ∈ Active do
4:          if l' ends before the start of l then
5:              Active ← Active - {l'}
6:              R      ← R ∪ {REGISTER(l')}
7:      REGISTER(l) ← Selct a register r from R
8:      Active      ← Active ∪ {l}
9:      R           ← R − {r}
10: for each child C of Block do
11:     COLOR(C, Active, R)
```

Notation: REGISTER(x): the register assigned to lifetime x

(b) The MCS-based algorithm for coloring a tree

Figure 3: Algorithms

the first block is the root, and any block is the child of the block before it. A variable has only one lifetime, which is conservatively extended such that it is a straight line. In a later extension [23], a variable can have more than one lifetime, each of them being a straight line. Linear scan is a globalization of the classical local register allocation [8], where all the lifetimes are within a single basic block.

In SSA-based register allocation [4, 12], all the blocks form a dominator tree. Any lifetime naturally spans a subtree of it: when a program is in SSA-form, a definition of a variable must dominate all its uses. In other words, a value of the variable naturally flows from the definition to all the uses along some paths of the dominator tree.

Clearly, in linear scan, local, or SSA-based register allocation, the input to the coloring process is a tree, and any lifetime spans a subtree of it. Their coloring processes are applications of MCS as well. In this sense, they are all special cases of tree register allocation.

3.6 Tree Enumeration and Solution Space

Given the freedom to construct the trees arbitrarily, in terms of their total number and individual structures, the number of possible outcomes is exponential. Even when considering only one tree with n nodes, there can be n^{n-2} number of distinct trees (Cayley's formula [26]).

In practice, it might not be meaningful to construct trees randomly. Since the coloring algorithm scans a path of a tree top-down, it is natural that there is dataflow from a block to its child in the tree. That avoids breaking the lifetime of a variable artificially into multiple lifetimes, which after coloring, may require additional dataflow transfer instructions to keep them connected. As examples, SSA-based register allocation uses dominator tree, and linear scan uses a 1-ary tree out of the reverse postorder of the CFG. Both kinds of trees preserve the dataflow relationship between the blocks, to different extents.

Even under the above practical constraints, the solution space is still big. This can be seen from the difficulties of finding an optimal 1-ary tree, one of the simplest cases. The problem is to decide the order of the blocks in the 1-ary tree. There are many number of possible orders. Each order may have different impact upon the code quality and it is hard to find the best one:

First, each order may introduce different register pressure if, in the way of the classical linear scan [20], a variable is allowed to have only one lifetime such that it covers the whole range between the first and the last program point where the variable is live. It is difficult to find the optimal order with the least register pressure. To see this, let us abstract each basic block as a single instruction

$$(w_1, w_2, ...) = f(u_1, u_2, ...)$$

where $w_1, w_2, ...$ are the variables that are written in the block, $u_1, u_2, ...$ are the variables that are used (read) in the block, and f is an arbitrary dummy operation. Then ordering the blocks is to order the abstract instructions. Is there an order of the abstract instructions so that the number of lifetimes overlapped is no more than the number of the available registers? This problem is known to be NP-complete [24]. Consequently, for linear scan register allocation, it is difficult to find out an optimal order of the blocks [22]. The reverse postorder used in the literature [20] is not necessarily optimal.

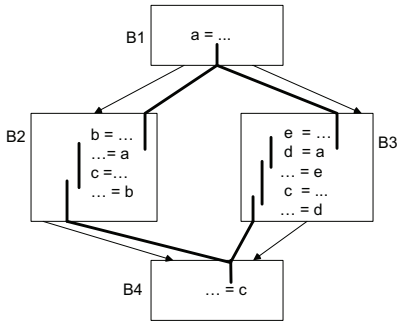
Second, if a variable is allowed to have multiple lifetimes, they may be assigned different colors, depending on the order of the basic blocks. Consequently, in connecting the dataflow, the number of dataflow transfer instructions needed is also dependent on the order. Intuitively, it is difficult to find the tree with the least dataflow transfer without enumerating all possible orders.

The above difficulties for linear scan suggest a big solution space (Otherwise, it can be easily searched by enumeration). In general, the same difficulties exist for any kind of tree. They present future research a vast solution space to explore.

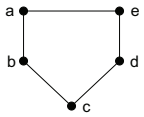
3.7 Spilling and Other Optimizations

So far, only the aspects fundamental to this approach have been shown. For practical implementation, spilling is important, and various other optimizations can help improve the code quality.

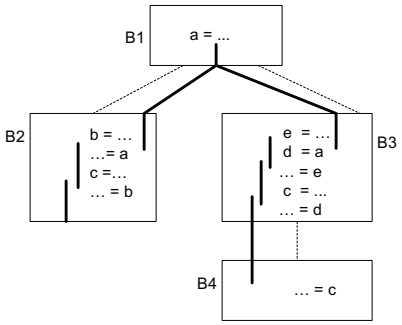
Spilling can be performed before coloring, and is often referred to as *pre-spilling* in this scenario. Some lifetimes



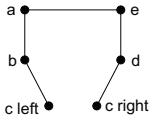
(a) Control flow graph



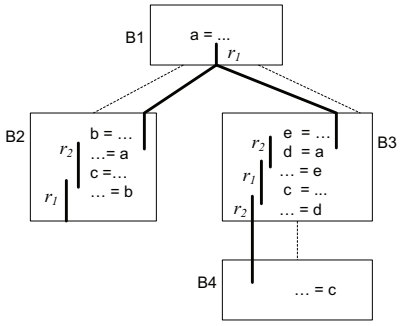
(b) Interference graph built from the CFG



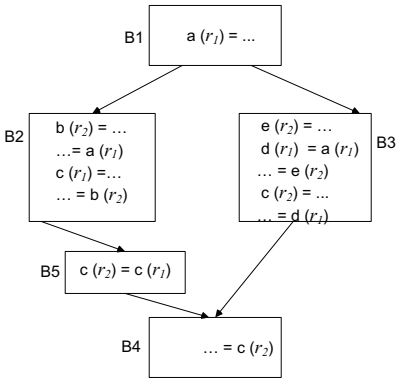
(c) The maximal spanning tree



(d) Interference graph built from the tree



(e) Coloring



(f) Final CFG with registers allocated and dataflow transfer inserted

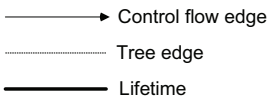


Figure 4: Illustration of tree register allocation

can be completely, or partially, spilled so that the maximum number of lifetimes live simultaneously in a tree does not exceed the total number of available registers. Subsequently, coloring is guaranteed to succeed without spilling.

This is a useful property common to all chordal graph-based approaches. Therefore, the pre-spilling heuristics developed for linear scan and SSA-based register allocation [12, 19, 22, 30] might be applied to general trees straightforward. The methods developed for local register allocation can also be inspiring: as coloring is so simple for it, spilling (or strictly speaking, pre-spilling), has been the major focus in previous studies, which result in many useful results [2, 8, 13, 14, 17].

Spilling can also be performed during coloring [20, 22]. When there is no free register available, a lifetime is chosen to be spilled, either completely, or partially by splitting it into multiple lifetimes.

Other optimizations, including coalescing, preferencing, rematerialization, etc., are orthogonal to this approach and should be applicable as well.

Note that these optimizations are feasible for any kind of tree, and may improve register allocation with some information that does not exist in the specific tree. For example, pre-spilling can use profile information to minimize loads/stores, and can be applied to any tree, including the dominator tree, which itself reflects control flow but no profile information.

Depending on the specific tree, there can also be specific optimizations for it. For example, for a 1-ary tree, second-chance bin-packing [29] improves the original linear scan [20] by considering the "holes" of a lifetime, the parts of the lifetime that do not have useful values for the corresponding variable, so that the register allocated to the lifetime can be re-used for other lifetimes fit in the holes.

Exception handling (EH) is an important practical issue. How to handle it in register allocation is to be introduced in Section 4.1.

4. EXPERIMENTS

The proposed approach has been prototyped in the Phoenix compiler framework. Five configurations have been tested to demonstrate the generality and potential benefits:

1. 1-ary tree. In this special case, the approach degenerates into linear scan register allocation. More exactly, it is the extended linear scan method named as ELS0 in the literature [23], since a variable is allowed to have more than one lifetime in the implementation. The tree is constructed with a reverse postorder traversal of the CFG.

2. Dominator tree. The approach degenerates into SSA-based register allocation in this case. SSA is not actually built, though: the essence of SSA is that each static definition of a variable is treated differently. This is naturally achieved by forming an individual lifetime, corresponding to one static definition, over the dominator tree. In this way, we not only save compile time in SSA construction and destruction, but also avoid the possibly complex problem of translating out of SSA [27] following register allocation [2].

3. Maximal spanning tree. In this case, a maximal spanning tree of the CFG is constructed using the profile counts for the control flow edges. The counts are statically estimated and propagated by the compiler.

4. Extended basic block trees (EBBs). Each extended basic block is an individual tree.

5. Basic block trees (BBs). Every basic block is a tree.

Tree formation is the only difference between the above configurations, to ensure their performance is compared under the same condition. An illustration of these trees has been shown in Fig. 2. These configurations of the tree register allocator have also been compared with the default priority-order register allocator in the compiler.

This section shows the preliminary results for all the benchmarks in SPEC2006 and SPEC2000 that can be built correctly so far. Each benchmark is built under -O2 -GL compiler options, which maximizes speed for the whole program, with each of the five kinds of trees, and the priority-order register allocator. The generated binaries are run on an Intel64 Core2 Quad 2.83Ghz machine with 2G memory. The observed results can be summarized as follows:

1. **Register allocation based on EBBs or the maximal spanning tree can be a competitive alternative to that based on the dominator tree**.

   The advantage of EBBs is in multiple trees: code from an inner and outer loop is naturally distributed to two separate trees, and thus the inner loop code can be allocated registers without the influence of the outer loop. In comparison, with the dominator tree, an outer loop basic block can dominate an inner loop basic block, and thus gets allocated earlier, which restricts the allocation of the inner loop later and may introduce spilling inside the inner loop [3].

   The maximal spanning tree wins in profile information: the dataflow along the most frequently executed control flow edges are guaranteed to be kept in the tree structure; therefore, dataflow transfer, if needed, will be inserted only on the less frequent control flow edges that are not in the tree. In contrast, the dominator tree is constructed purely based on the control flow structure. Profile information is not used in that process.

[2] According to Brisk [4], "the translation out of SSA form following the assignment of register and memory locations is an intricate and tricky process that has a complex interaction with the register and memory assignment phase of the allocator.... The translation out of SSA can insert a significant number of loads and stores, derived from the assignment of register and memory locations, and possibly even cause additional spilling. At present, this issue appears to be the largest impediment to the success of SSA-based register allocation." This paper and the implementation show that it is not necessary to build SSA at all in order to take advantage of the "SSA-based" approach. The key is to build an individual lifetime for each static definition of a variable. When the program is in SSA, it is convenient to do so, because different definitions of the same variable have already been renamed *based on the dominator tree*. When the program is not in SSA, the same thing can be done by performing the standard liveness analysis, and then building the lifetimes over the dominator tree. Thus SSA is not a necessary component. However, the dominator tree is indispensable: the coloring algorithm needs it to choose the lifetimes to color, no matter the program is in SSA or not. So it is the dominator tree, not the SSA form, that directly matters to the register allocator. In this sense, "SSA-based register allocation" is somewhat a misnomer. It might be more accurate to be called "dominator tree-based register allocation".

[3] This can be avoided if pre-spilling spills for the inner loop first. A comparison in this case is left as a future subject.

Consequently, in coloring the tree, profile information is not taken advantage of.

This observation might also indicate a new direction, which combines the advantages of multiple trees and profile information: construct multiple trees, each being the maximal spanning tree for a program region (e.g., a loop), where a region contains a set of basic blocks with similar execution frequencies.

2. **Register allocation based on EBBs, the maximal spanning tree, or the dominator tree, tends to generate better code than linear scan. The reason is that they introduce more "control flow sensitivity" to register allocation, to different extents**.

   The control flow structure of the original program is maintained, more or less, in the EBBs, spanning tree, or dominator tree. Correspondingly, the original dataflow along each control flow path is basically maintained in the tree(s) as well. Therefore, the lifetimes in the tree(s) are more natural; the basic blocks with close dataflow relationship are putting close together, which avoids unnecessary dataflow transfer between them afterward, and also reduces spilling.

3. **Compared with the priority-based register allocator, the tree register allocator so far has an expected performance gap of 14% (17%), but compiles 40% (31%) faster, for SPEC2006 (SPEC2000).** The performance gap is due to the lack of other optimizations and heuristics.

4.1 Details of the Implementation

The implementation has the following steps: (1) Modifying EH instructions, (2) Liveness analysis, (3) Tree formation, (4) Simple pre-spilling and preferencing, (5) Coloring, in which a variable may have more than one lifetime, each being treated independently. Spilling is implemented within coloring, where a lifetime can be partially spilled on demand: it is spilled between two points along a path in the tree, where the register pressure is over the available registers; before the first point, it is still in a register; after the second point, it is added back to the candidate list, and re-colored later; such spilling/recoloring of the lifetime along one path does not affect the processing of it along other paths. Spill code is not inserted at this step, but the spill points are remembered. (6) Inserting spill code, (7) Recovering the EH instructions, and finally (8) Connecting dataflow. To help debugging the correctness of the register allocator, a validator is also implemented, which compares the definition-use relationship before and after the allocation. It is similar to SARAC[15] but uses SSA to calculate reaching definitions. It runs before and after the above steps in debug mode.

Here we explain Step 1, 4, and 7 in more detail. All are related with EH.

An instruction that may cause exceptions is always the last instruction in a basic block, which has two outgoing control flow edges: one is to the fall-through block, the other to the exception handling block. Semantically, the variables defined by this instruction are live in the fall-through block, but not in the exception handling block. Take the following instruction as an example:

```
x = call foo, L1 (EH),   (1)
```

where `x` is the destination variable, `L1` is the entry label of the exception handling block. If any exception happens during the call to function `foo`, `x` is not defined yet, and thus it is not live in the exception handling block. To reflect this semantics without changing the standard liveness analysis procedure, the instruction is broken into two instructions:

```
t = call foo, L1 (EH)   (2)
x = t                   (3)
```

where the first instruction remains in the same basic block as that of the original call instruction, while the second instruction is placed at the beginning of the fall-through block.

After coloring is done, we can recover the EH instructions to their original shape (Step 7). For example, the two instructions (2) and (3) shown above are merged back into a single instruction (1), if both variable `x` and `t` have been assigned the same register.

EH also requires some lifetimes to be pre-spilled (Step 4), when they are live along a control flow edge to an exception handling block. Such an edge is controlled by the run time system when an exception happens from the hardware, and thus we cannot insert any dataflow transfer instruction on it. To be simple, all the lifetimes along such an edge are spilled at the end of the predecessor block of the edge.

Pre-spilling is also done in step 4 at the entry of any basic block, when there are more lifetimes live simultaneously than the available registers. Some lifetimes are partially spilled, in the order of their furthest uses in the block. Advanced pre-spilling heuristics considering profile information are yet to be implemented.

Simple preferencing is also implemented in step 4. Each lifetime is associated with one preferred physical register. For a copy instructions in the program,

```
x = y,                  (4)
```

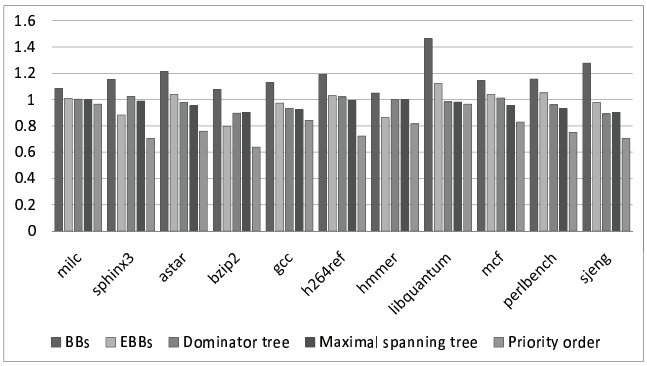
where `x` and `y` are two variables, if `x` is pre-colored with a register `r` (for example, due to calling convention), propagate this color to `y`. That is, the preferred register of the lifetime of `y` is set as register `r`. Then from the definitions of `y`, the same propagation continues.

In the subsequent coloring process, a lifetime will be assigned its preferred register, if it is free.

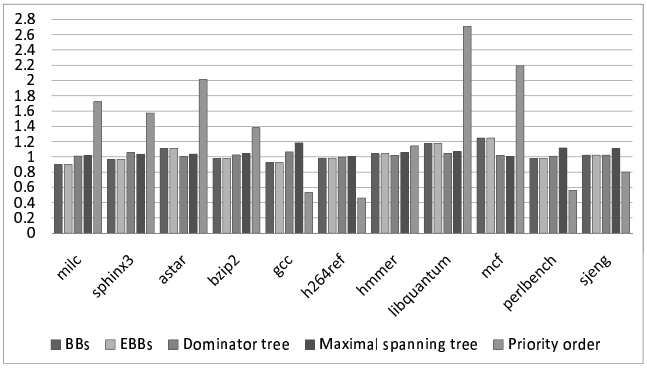
4.2 Preliminary Results

Fig. 5 shows the execution time of SPEC2006 and SPEC2000 benchmarks, and their compile time spent in register allocation under different configurations. Execution (compile) time is normalized by being divided by that of the 1-ary tree configuration, i.e., linear scan. The lower a bar is, the better the code quality (throughput).

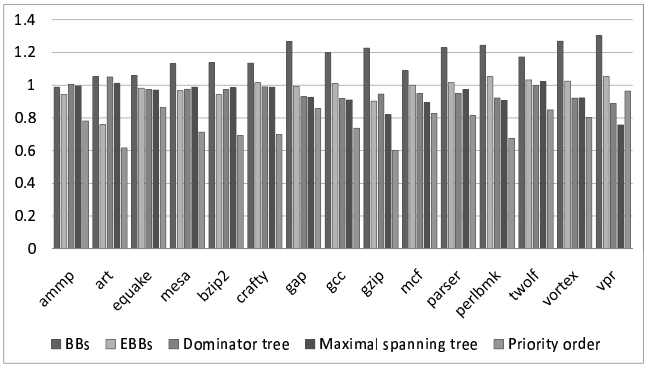
For SPEC2006, the BBs configuration, which can be regarded as a global register allocator naively extended from a local register allocator, is 18% slower than linear scan on average, because of the extraneous dataflow transfers between the basic blocks. This situation is improved by the EBBs configuration, in which case, there is no such transfer needed between any two blocks connected by a tree edge. With the EBBs, the execution time is 2% faster than linear scan on average. Then with the dominator tree configuration (SSA-based register allocation), the execution time is about 3% faster than linear scan on avarage. However, all these configurations do not make use of any profile information. In the compiler framework, there is static profile



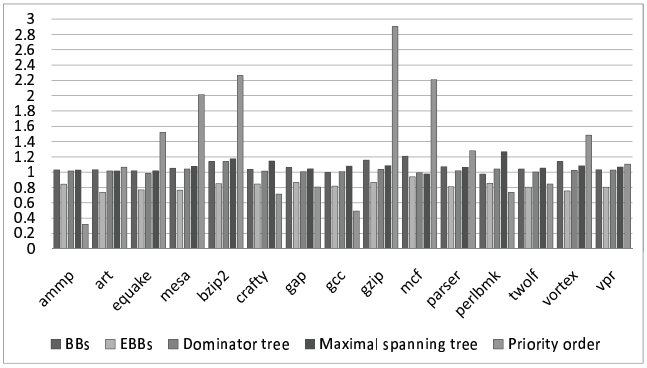
(a) Execution time of SPEC2006.



(b) Compile time in register allocation for SPEC2006.



(c) Execution time of SPEC2000.



(d) Compile time in register allocation for SPEC2000.

Figure 5: Execution and compile time for SPEC2006 and SPEC2000, normalized against those of linear scan.

information available in the register allocation phase. By making use of this information to build the maximal spanning tree, the execution time is about 5% faster than linear scan on average.

The SPEC2000 execution time is similar. The BBs, EBBs, dominator tree, and maximal spanning tree configurations are about -16%, 2%, 4%, 6% faster than linear scan, respectively.

It is noticeable that although the average performance of EBBs configuration is worse than the dominator tree, it is significantly better in several cases: the code is 10% $\sim$ 29% faster in SPEC2006 sphinx3, bzip2, hmmer, and SPEC2000 art. The win is due to the multiple trees for code with different execution frequencies: code in an inner loop is naturally in a different tree from code in an outer loop: The start basic block of a loop has more than one incoming edges; Such a block can not be a node in an EBB other than the root node, according to the definition of EBB; Therefore, whenever a loop start is encountered, a new EBB is built. Thus the inner loop code is allocated registers without being affected by the outer loop. In contrast, with the dominator tree, since an outer loop basic block can dominate an inner loop basic block, even though it executes less frequently, it can be allocated registers earlier, which restricts the allocation of the inner loop later and may introduce spilling inside the inner loop.

The maximal spanning tree generates as good code as the dominator tree in most cases, and is significantly better in a few cases: the code is 5% $\sim$ 13% faster in SPEC2006 mcf, SPEC2000 gzip, mcf, and vpr. By making use of the profile information, the most frequently executed control flow edges are guaranteed to be kept in the tree structure. Therefore, after coloring, if any dataflow transfer instructions are needed, they will be inserted only on the less frequent control flow edges that are not in the tree.

It is also noticeable that EBBs, the maximal spanning tree, and the dominator tree, tend to generate better code than linear scan. They all keep the control flow structure of the original program to more extents, and thus the original dataflow of the variables are naturally reflected in the lifetimes over the trees, which avoids breaking them artificially and inserting dataflow transfer later. In some sense, they have introduced better "control flow sensitivity" to register allocation.

So far, the performance of the tree register allocator is off the default priority-based register allocator in the compiler by 14% and 17% for SPEC2006 and SPEC2000 on average, but its compile time is 40% and 31% less.

The performance gap is expected. The priority-based register allocator has been fine tuned for several years. It is

sophisticated and has various heuristics for spilling, preferencing, splitting, rematerialization, address mode folding, coalescing, etc. In comparison, various heuristics are yet to be implemented for the very new tree register allocator.

There are several obvious limitations to the current implementation: (1) Preferencing: Each tree is allocated registers independently, while there are opportunities to propagate the results of a tree to the next tree to be colored, so that some dataflow transfers between the trees can be avoided. (2) Pre-spilling and spilling: no profile information has been used in pre-spilling or spilling. Stores are inserted wherever a lifetime is spilled, no matter the lifetime's value has been changed or not. Similar redundancy exists for reloads. (3) A full register is always allocated to a variable, even when a sub-register of it suffices.

These limitations are specific to the implementation, not to the theory. They are being worked on and will be removed in future. So far, the implementation has already shown the potential of the approach to be beyond linear scan and the dominator tree (SSA)-based register allocation. With various advanced heuristics implemented, it is reasonable to expect the gap to be closed in future.

5. CONCLUSION AND AN OPEN QUESTION

This paper has gone one step further beyond the recent studies in chordal graph-based register allocation. It shows that linear scan and SSA-based register allocation are the members of a large family. A very simple and flexible framework has been proposed. Initial experiments have demonstrated the potentials.

Section 3.7 explains that pre-spilling can be done before coloring but still on a tree. Consider another scheme, where spilling is performed even earlier: before tree formation. In this case, spilling is independent of any specific tree. A tree constructed afterward is guaranteed to be colored successfully without spilling. Then any trees, in terms of spilling, have no difference. The only difference between them would be the dataflow transfer instructions inserted to connect the dataflow. *How to find a tree that minimizes the number of dataflow transfer instructions, without enumerating all possible trees?* This is conceivably a difficult problem. There would not be a unique answer, and many useful results are expected.

6. ACKNOWLEDGMENTS

I am grateful to the valuable feedback from David Gillies, Weirong Zhu, Yuan Zhang, and the anonymous reviewers, and the support from Jim Radigan and Steve Kruy. I appreciate the comprehensive functionality and friendly API of Phoenix out of the great efforts of the team.

References

[1] Interval graph. http://en.wikipedia.org/wiki/Interval_graph, May 2009.

[2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems*, 5(2):78–101, 1966.

[3] P. Briggs. *Register allocation via graph coloring.* PhD thesis, Rice University, Houston, TX, USA, 1992.

[4] P. Brisk. *Advances in static single assignment form and register allocation.* PhD thesis, University of California at Los Angeles, Los Angeles, CA, USA, 2006.

[5] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. *SIGPLAN Not.*, 26(6):192–203, 1991.

[6] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[7] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, 1990.

[8] K. Cooper and L. Torczon. *Engineering A Compiler*, chapter 13.3.2 Bootom-Up Local Register Allocation. Morgan Kaufmann Publishers Inc., pages 626–629, San Francisco, 2004.

[9] K. D. Cooper, T. J. Harvey, and L. Torczon. How to build an interference graph. *Softw. Pract. Exper.*, 28(4):425–444, 1998.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[11] F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1):47–56, 1974.

[12] S. Hack, D. Grund, and G. Goos. Towards register allocation for programs in ssa-form. Technical Report RR2005-27, Universität Karlsruhe, September 2005.

[13] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index register allocation. *J. ACM*, 13(1):43–61, 1966.

[14] W. Hsu, C. N. Fisher, and J. R. Goodman. On the minimization of loads/stores in local register allocation. *IEEE Trans. Softw. Eng.*, 15(10):1252–1260, 1989.

[15] Y. Huang, B. R. Childers, and M. L. Soffa. Catching and identifying bugs in register allocation. In *SAS*, pages 281–300, 2006.

[16] D. R. Koes and S. C. Goldstein. A global progressive register allocator. *SIGPLAN Not.*, 41(6):204–215, 2006.

[17] V. Liberatore, M. Farach-Colton, and U. Kremer. Evaluation of algorithms for local register allocation. In *CC '99*, pages 137–152, London, UK, 1999. Springer-Verlag.

[18] H. Mössenböck and M. Pfeiffer. Linear scan register allocation in the context of ssa form and register constraints. In *CC '02*, pages 229–246, London, UK, 2002. Springer-Verlag.

[19] F. M. Q. Pereira and J. Palsberg. Register allocation via coloring of chordal graphs. *Lecture Notes in Computer Science*, 2005(3780):315–329, 2005.

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

[21] T. A. Proebsting and C. N. Fischer. Probabilistic register allocation. *SIGPLAN Not.*, 27(7):300–310, 1992.

[22] K. Sagonas and E. Stenman. Experimental evaluation and improvements to linear scan register allocation. *Software: Practice and Experience*, 33(11):1003–1034, 2003.

[23] V. Sarkar and R. Barik. Extended linear scan: An alternate foundation for global register allocation. *Lecture Notes in Computer Science*, 2007(4420):141–155, 2007.

[24] R. Sethi. Complete register allocation problems. In *STOC'73*, pages 182–195, 1973.

[25] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *J. ACM*, 17(4):715–728, 1970.

[26] P. W. Shor. A new proof of cayley's formula for counting labeled trees. *J. Comb. Theory Ser. A*, 71(1):154–158, 1995.

[27] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam. Translating out of static single assignment form. In *SAS '99*, pages 194–210, London, UK, 1999. Springer-Verlag.

[28] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.

[29] O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *PLDI'98*, pages 142–151, 1998.

[30] C. Wimmer and H. Mössenböck. Optimized interval splitting in a linear scan register allocator. In *VEE'05*, pages 132–141, 2005.