A Survey of Register Allocation Techniques

Jonathan Protzenko

July 29, 2009

Contents

PART I
GETTING STARTED

1 Introduction

Register Allocation is one of the most studied problems in the field of compiler optimization. Indeed, the legacy of past architectures (few registers) as well as the ever-lasting relative slowness of memory versus registers (several orders of magnitude) make such an optimization crucial for the good performance of a program.

Algorithms have been devised over the years to deal with that problem. We have gone from "blind" techniques to "informed" algorithms that try to take into account the different aspects of the problem. New advances in compiler science (SSA) have turned previously minor problems into subjects of uttermost importance. We now have a good understanding of the three sub-problems register allocation implies:

- *spilling*: given a variable, should that variable be kept in registers or spilled in memory?
- *coalescing*: given two virtual registers A and B, and given that the last use of A is to move it into B's virtual register, should we assign A and B the same virtual register?
- *coloring*: given an interference graph, how should I assign colors to nodes, such that do adjacent nodes do not share the same color?

These three subproblems have had different answers over the years. Taken separately, some of these problems can even be solved! However, taking every constraint into account to give the best allocation is still an open problem. In this survey, we will quickly detail past fundamental papers [Cha, BCT94]. Then, we will detail different state-of-the-art approaches used in different contexts.

We have tried to stick to most recent papers, or to papers that describe a method that is still the most widely used today. Indeed, many of the papers in the bibliography are less than 10 years old. Two papers, [PS99, GA96] are rather old. However, they represent the most widely used techniques in their respective fields. We cannot not include them.

We will assume in this survey paper that the reader is familiar with common facts, such as the link between graph coloring and register allocation, and has a rough understanding of SSA form. Our survey of "classical" papers [Cha, BCT94] will help the reader remember the main facts about register allocation. Rather than creating a huge section dedicated to definitions and concepts, we will introduce them as needed (like SSA).

a. Which register allocation?

Register allocation is such a huge problem that there are many different situations in which it is relevant. Imagine for instance that we are designing a component for inclusion into a larger electronic appliance. That component is going to perform computations, and will need

registers. How can I design it so that the number of registers I add in the circuit is as small as possible?

This is indeed a variant of register allocation, that is out of the scope of this article. What we will focus on is the three following situations, probably the most common ones where register allocation is needed, and certainly the most studied:

- classical register allocation in "modern" processors with more than 16 registers;
- specific register allocation for one family of widely used processors that surprisingly have as little as 8 registers (namely, the Pentium series, as the reader will have guessed);
- register allocation for JIT (Just In Time) compilation.

Before going further, it seems to us a good idea to give the main ideas behind JIT compilation, as this is still quite a novel concept.

JIT compilation stems from the following remark: imagine a mobile device (e.g. a mobile phone) running an application. The variety of available processors and devices is such that a vendor cannot reasonably compile its program for every target phone. As a consequence, the vendor distributes his program as a *bytecode* program (namely, J2ME[1] bytecode): code optimized for a virtual machine (stack-based for Java). That code can be *interpreted* by the virtual machine on each phone, but that is slow and gives poor performance. What can be done, instead, is translate bytecode into the actual machine code, on the mobile phone, so as to take into account the specifics of the architecture the program is to be run on.

The main problem with this approach is that it takes a huge time to compile everything in the first place from the bytecode to the real machine code (even more on a mobile phone!). Maybe the user will use only a tenth of the program, and compiling everything is a waste of time. For those reasons, we use JIT compilation: we compile code just before it is run, so as to minimize the amount of code compiled. *We only compile what is necessary.* We do not compile the whole program, only the pieces of code that are about to be run. In this context, the constraints are different: we want compilation to be ultra-fast, and produce "acceptable" code, not the best code possible.

These constraints have given birth to a rich flow of papers and clever algorithms to tackle this special case, that is becoming more and more important nowadays.

b. Which papers?

For the survey of past methods, we will review quickly two of the most fundamental papers: [Cha] and [BCT94]. For "classical" register allocation, [GA96] and [PP05]. Some insights one the "underlying complexity" will be given by [HG06] and [PP]. For "Pentium" register allocation, [AG01] is the fundamental paper with its counterparts [PM04] and [GH07]. For JIT register allocation, [CD06] and most importantly [PS99] are some key papers. Some results on NP-completeness will be provided by [BDR07a] and [BDR07b]. Finally, very novel techniques will be studied, namely [PP08].

[1] Java2 Mobile Edition

2 Quick history of register allocation

a. Chaitin

Although the link between graph coloring and register allocation seems to have been spotted in the 1960s, the first actual graph-coloring-based register allocator was designed by Chaitin [Cha] during his work at IBM on a PL/1 compiler. His first 1981 paper did not know what to do when the graph was not k-colorable[2] and was spilling randomly. His 1982 paper fixed that issue and introduced smarter heuristics to deal with spilling.

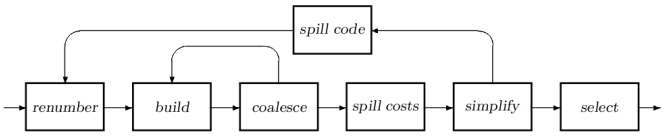

Figure 1: Chaitin's register allocation scheme

What Chaitin did was, basically, build the interference graph, coalesce nodes whenever possibe, and start *simplifying* the graph, that is, removing nodes one by one starting with the lowest degree first, until one node has too big a degree ($\geqslant k$ in our case): that corresponding live range is spilled and the whole process restarts until all nodes can be safely removed. The nodes are then put back (*select*) one by one and one color is assigned to each of them. We are guaranteed to find a color for each node because we spilled every node that has degree more than k, hence leaving a graph with only nodes of degre $\leqslant k-1$ that is trivially colorable, using the stack method.

However, Chaitin did not realize the terrible effects of his agressive coalescing: when node A's last use is to be moved into node B's virtual register, we can remove the `MOVE` instruction and save an instruction. With the arrival of SSA form, many temporaries are created (temporary variables). It would seem that coalescing all of them would save many instructions. However, this often creates huge live ranges that interfere with many more variables than the previous smaller live ranges. By aggressively coalescing, we break the colorability of the graph. However, we cannot avoid coalescing, because with SSA, `MOVE` instructions can represent up to 10% of the total instructions.

b. Briggs

Brigg's main contribution was to devise another graph coloring heuristic, that performed better at coloring the graph, and saved many spills.

He called this optimistic coalescing. In the simplify phase, when we remove nodes from the graph and put them one by one on a stack, instead of failing as soon as a node has degree $\geqslant k$, we just leave it transparent and hope to be able to color it later, when we pop it back.

[2] In the rest of this paper, we will use k as the number of available registers on the target architecture.

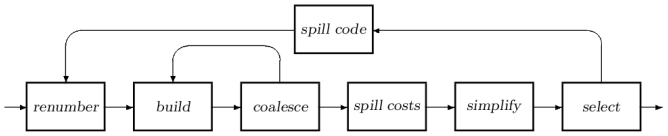

Figure 2: Brigg's register allocation scheme

Then, in the select phase, if one node can't actually be colored, it is spilled and the graph is rebuilt.

The main advantage of this method is that it can color more graphs, for instance, the diamond graph (that was not colorable with Chaitin's scheme).

Briggs also introduced a *conservative coalescing* method to coalesce only when not breaking the colorability of the graph. However, his method was too conservative, and he resorted to additional heuristics, with no theoretical justification, to coalesce more and avoid generating code with poor performance.

PART II
REFERENCE ALGORITHMS

What happened afterwards is that people realized that coloring is not the main challenge in register allocation. Indeed, it is important. But without a good coalescing strategy, however good is the heuristic used to color, there may be unsolvable cases. Indeed, a poor coalescing strategy will give poor results, and the coloring heuristic will not help.

1 Iterated Register Coalescing

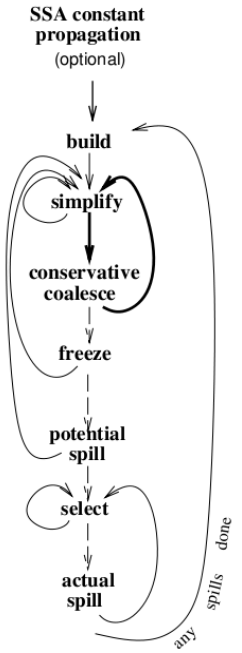

Figure 3: Iterated Register Coalescing

The first breakthrough was with Andrew Appel and Lal George's Iterated Register Coalescing [GA96]. By first removing non-`MOVE`, easily colorable nodes from the graph (*simplify*), they create more opportunities for coalescing. `MOVE` nodes are then coalesced *conservatively*, creating more opportunities for simplifying, and so on. When nothing can be done, we first give up on coalescing (*freeze*), and we chose to simplify `MOVE` nodes (they won't be coalesced but removed from the graph, if possible). When even this doesn't unlock the situation, we give up on registers, and we chose to potentially spill a node. We then keep doing all those iterations until all nodes have been removed from the graph. We then rebuild the graph in reverse order, and just like Briggs, we try to see if potential spills still can find an available color, or if they must be actually spilled. If an actual spill occurs, the IL[3] code is modified, the interference graph rebuilt, and the whole process starts over again.

Iterated Register Coalescing is a well-known algorithm, and is very popular in compilers that are linked in some way to research. SML compilers[4][5], Erlang compilers, and even a certified C compiler [Ler06] use that algorithm. It is somehow the *de facto* algorithm for people designing a new compiler. Every new register allocation scheme is compared against this algorithm, and we will

[3] Intermediate Language

[4] ML is a functional programming language that is based on the semantics of lambda calculus and that is very popular among theoretical computer scientists

[5] Appel has written a series of books: compiler implementation in {ML,Java,C} which are very popular in compiler classes and has also created a (failed) continuation-passing-style compiler for ML

do so in this report.

It is to be noted that an improvement has been proposed in 2004 [SRH04] that allows the algorithm to take into account register aliasing. That makes the allocator even more general and can lead to subtle improvements in some situations.

2 Linear scan

Another extremely popular algorithm, also used in many real world compilers such as LLVM [LA04] [6], is linear scan[7].

Linear scan [PS99] is a fast algorithm *not using graph coloring*. It is simple to implement and is linear: register allocation is performed as we compute the live ranges of the variables. This has contributed to its popularity, and it is the choice allocator when either manpower lacks to implement a more complicated allocator, or speed is required, such as in JIT environments. Its inclusion in many traditional compilers has convinced us to put it in this section. Linear scan produces code that is comparable in quality to other allocators, such as Appel-George's (although iterated register coalescing is always better, this linear scan method performs roughly 10% inferior as a general measurement), but is much faster at compile time.

We shall here call a live interval an interval associated to a variable so that the variable's live range is included in that interval. In short, a live interval is an "extended" live range. Here are the key steps of the linear scan algorithm.

- number the instructions so that if $i > j$, then instruction i is always executed after instruction j;
- run through the instructions in increasing order (1 is the `START` instruction and N is the `END` instruction);
- keep a list of *active* live intervals, that is to say, intervals that overlap the current point in the program, sorted by increasing end position;
- every time we enter a new live interval, we run through *active* intervals and we remove those who have expired (that no longer overlap the current point) – since the list is sorted according to the point where each interval ends, all the intervals we visit except the last one will be removed;
- at each point of the program, thanks to the *active* list, we know how many variables might be live (live intervals are larger than live ranges), and we know which registers are used: the variables in the *active* list are those using registers;
- every time we enter a new live interval, we know if we can allocate a new register to the current variable;
- if not, the heuristic is to spill the active register that ends the further away from the current point ($\mathcal{O}(1)$ with a good list data structure), hence removing it from the *active* list.

This seems very simple: it is! It must be mentioned that the description of the algorithm takes less than one page in the original paper, which is quite a record.

a. Complexity

If V is the number of variables, the only operation we need to consider is the following one. Let us say that we are currently in variable A's live interval, and that we enter variable B's live interval. Obviously, to keep the algorithm consistent, A's live interval must enter the *active* list. This is a very simple list insertion. Moreover, remember the we can have no more than R active intervals, where R is the number of available registers (because every active variable is in a register). Then this is a bounded quantity! $\mathcal{O}(\log R)$ with a binary tree or $\mathcal{O}(R)$. The total complexity of the algorithm is thus $\mathcal{O}(V \times \log R)$. And given that R is a constant of the problem, the complexity is $\mathcal{O}(V)$, that is to say, linear.

b. Comparison with previous algorithms

This algorithm is undoubtedly much faster *at compile-time* than graph-coloring based approaches, for the very simple reason that a graph can be, at worst, $\mathcal{O}(V^2)$ in size. Benchmarks designed to challenge register allocators clearly give the advantage to linear scan concerning compile time.

At run-time, linear scan gives worse results than graph coloring. The difference is not significant, however, with 10% less as an average and 20% in the worst case.

c. Refinements

Another method based on linear scan, but more complicated, is second-chance binpacking. It mitigates the worst-case results of linear scan, but we believe it is of less importance as it loses both the fast speed at compile-time and the elegant simplicity of the original algorithm.

Another refinement the authors explain is that the live range analysis can be replaced with an approximate version (which is why we used "live intervals" in the description of the algorithm) to be even faster at compile-time, but gives poor results.

Live ranges can be splitted (so as to take advantage of "holes" in live ranges, and avoid stupid spilling of variables in loops), but that adds complexity to an algorithm whose main advantage is its simplicity.

d. Our opinion

Linear scan suffers from many defaults: always slower than iterated register coalescing, it doesn't even try to coalesce variables! As a consequence, we believe that such an algorithm is a very good candidate for embedded systems and JIT compilation, but a poor candidate for compiling desktop programs.

Moreover, most programs are only compiled very infrequently: the goal of a program is to be run, after all. Since the cost of a graph-coloring register allocator is still

[6] A highly modular compiler that can be easily modified to test new compiler optimizations. Different register allocators are available by default, and the default one is linear scan. Many authors implement their register allocation algorithms in LLVM, although the learning curve is quite steep.

[7] Although in a slightly modified version

affordable, we believe that linear scan should only be considered as a last resort option.

3 Other strategies, complexity

Much research has focused on the best coalescing strategies instead. As we said, it turned out that the heart of the problem lies in coalescing. For instance, optimistic register coalescing [PM04] claims better results than iterated register coalescing. However, finding the optimal coalescing given one of these strategies is NP-complete [BDR07a]. As a consequence, research has slightly shifted and is trying to find a way using polynomial algorithms on a special class of graphs, as we will see in the next section.

PART III
LATEST DEVELOPMENTS

Some very recent developments have arised in the field. We will present first what has been understood recently on the complexity of register allocation, as well as new techniques based on chordal graphs. We will then describe a radically different approach.

1 SSA, complexity and chordal graphs

Chaitin showed how to describe exactly the register allocation problem as a graph coloring problem. He also showed, that given any graph, a program can be created with this exact interference graph. Hence, if we could solve register allocation, we would be able to color any graph (polynomial reduction). **Register allocation is NP-complete.**

a. An introduction to SSA

As we discussed before, programs are subject to a number of optimizations, all of which are performed under the SSA form. SSA is the standard representation for programs now, and every decent compiler uses it. In SSA, each variable is assigned only once (each use has only one def). For variables that depend on the program flow (a variable that has been changed in an `if then else` for instance), after the control block, a so-called ϕ-node is inserted that allows to pick either the variable from the `if` or from the `else`, depending on the result of the test. This intermediate (abstract) representation is then removed before register allocation (using `MOVE` instructions)[8].

b. A special case

One interesting fact is that the interference graphs *with the SSA form* are always *chordal* [HG06]. This class of graphs has many interesting properties. For instance, the NP-complete problems maximum-clique, maximum independent set, minimum covering by cliques are for that particular class of graphs tractable in polynomial time. Similarly, **coloring a chordal graph is possible in linear time**.

A graph is said to be chordal if every cycle of more than three nodes has an edge *not belonging to the cycle* that links together two vertices of the cycle. Such an edge is a chord.

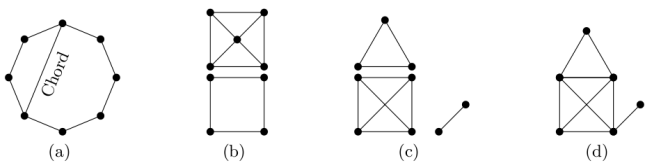

Figure 4: Chordal graphs: (a) A chord in a cycle (b) Two non-chordal graphs (c) Some cliques (d) A chordal graph built from those cliques

[8] hence requiring coalescing afterwards, since all those ϕ-nodes represent new temporaries, with new virtual registers and different live ranges

Hack's article proves (using dominators) that every SSA graph is chordal. We will not give any details of this proof, for it would require us to dive into many details about compiler techniques (lattices, dominators, concrete details of SSA) that would take too long to be explained. He also explains that, using dominators again, there is no need to build the interference graph to find a coloring of the nodes.

Another interesting feature is that if the SSA graph is k-colored, the article shows how to remove the SSA so that the program still uses no more than k registers.

This looks like a solution to our problem of register allocation! However, the article does not discuss the problem of spilling at all. So if the graph is not k-colorable (with a fixed k equal to the number of registers on the target machine), there is no solution to the spill problem: having a $k' > k$-coloring does not tell us the best choice for simplifying the graph. Let us take an example to see that.

If you consider an uneven cycle with n nodes, it will need three colors to be colored. What if we have $\frac{n-1}{2}$ red nodes, $\frac{n-1}{2}$ blue nodes, and one green node? We could chose to spill all the red nodes? That would be a poor decision, indeed. This simple example illustrates the following point: the heart of the problem is not in coloring but in splitting live ranges (see [BDGR]).

c. Not so useful...

[PP] gives another insight and proves that (and this is the title of his article) "Register allocation after classical SSA elimination is NP-complete".

It is possible to transform a standard program in SSA form, polynomially. We could then run the coloring algorithm on the SSA form, and then transform back the program into the "normal" form ($\mathcal{O}(n^3)$). That would enable us to "plug" that coloring method into previous algorithms to get better results (and take into account spilling as we've done before). However, if a coloring is valid with the SSA form, the standard transformation into regular form breaks the coloring, and there is no way (that is the proof in the article) to make use of the SSA form to get a polynomial coloring.

The intuitive reason is that SSA programs have shorter live ranges that make coloring possible, but with classical SSA elimination, the problem is no longer polynomially tractable because the interferences are more complicated (longer live ranges).

We could use the special transformation from [HG06] but there is no guarantee that we know what to do with the resulting graph, and once again, that offers no solution for the problem of spilling and coalescing.

d. A new method

The situation seemed to be locked (that was in 2005). However, building on the previous efforts, Pereira came up with a new register allocation technique that builds on the previous discoveries of tractability of chordal graphs [PP05] and makes use of those techniques. He claims to outperform iterated register coalescing. Indeed, this is a very new method, for it does not use the classic graph-coloring register allocator scheme. Let us see how it works.

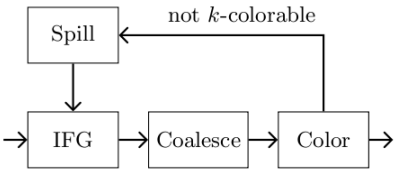

Figure 5: A traditional graph-based register allocator

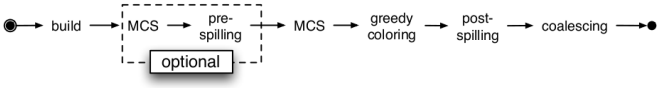

Figure 6: A chordal graph-based register allocator

Pereira's method is based on the following observation: a high percentage of functions have an interference graph which is chordal (95,5% in the Java standard library). What if we could design an algorithm that takes advantage of the natural properties of chordal graphs to perform optimally when the function has a chordal graph, and have good performance otherwise?

The algorithm uses a procedure called Maximal Cardinality Search (MCS) that, when combined with the greedy coloring described in the article, gives an optimal coloring for chordal graphs (still in linear time). For other graph, is still gives good performance as shown in the benchmarks of the article.

The next question is: what to do with spilling (described in the schema as "post-spilling")? What is remarkable is that, given k (the number of available registers, that is to say, colors), there exists a polynomial algorithm to find out the maximal k-colorable subgraph of a chordal graph! However, two problems arise:

- the complexity is $\mathcal{O}(V^k)$ where V is the number of live ranges...!
- our "real" graphs are only partially chordal.

The author then discusses different heuristics (the polynomial algorithm is unusable in practice...who would want a $\mathcal{O}(V^3 2)$ algorithm?):

- spill all nodes having the least used color;
- spill all nodes whose color has been introduced last.

It turns out, rather surprisingly, that the second heuristic is the more powerful of the two. When the greedy algorithm runs, it introduces new colors if it runs out of available colors. The colors that were introduced last, if removed, give the best results.

It is to be remarked (see figure 6) that this is a *one pass process* and not iterative, as in previous register allocators. Moreover, the graph is colored first to see which colors to

remove, which is once again a difference with previous algorithms.

Finally, we coalesce. The decision is made, wisely, to coalesce in a final phase, so as not to break the chordality of the graph. The algorithm is a greedy one and, most importantly, *informed* one: for each node, the algorithm knows *for sure* if a color will be available for it after coalescing because *the coloring is already done*. The algorithms proceeds in a greedy fashion and reviews all the nodes.

Finally, pre-spilling is another phase that takes advantage of the remarkable properties of the chordal graphs. Since optimal coloring is possible for those graphs, the number of colors needed is the size of the maximum clique in the graph. There exists a polynomial algorithm to selectively remove nodes so as to bring down the size of the maximum clique to the available number of colors. Once again, not all graphs may be chordal, so a post-verification (the "post-spilling" phase) is needed to faithfully ensure the graph is properly colored. Moreover, as we said, this is in no way guaranteed to be a good idea for non-chordal graphs. However, experiments have shown that this strategy performs well for non-chordal graphs as well.

e. Conclusion, forecasts

This algorithm exhibits better performance that Appel and George's iterated register coalescing. Moreover, it is faster at the compile time and its implementation is simpler.

This is due mainly to the fact (which is quite a break with previous strategies) that **the coloring is done first**. This allows for more "well-thought" decisions in the subsequent phases and more informed strategies. Optimality is achieved in terms of coloring and spill for many methods, and only highly complicated methods with a lot of temporaries do not exhibit chordal graphs, and are not optimally colored.

There is still room for improvement: the coalescing phase does not take advantage of the properties of chordal graphs. Moreover, this was only tested on the Java standard library which might not be very representative of programs in general.

However, this is a very promising area, and the energy that has been devoted in it shows that even better results can be expected in the next few years.

2 Using puzzle-solving

The same author, F. Pereira, has recently created a new abstraction for register allocation, based on puzzle solving. Not only is it a total break with previous graph-based approaches, but it is also very competitive with Appel-George's iterated coalescing and uses many ideas from recent papers. It works as follows.

a. The idea

For each type of machine, he creates an associated puzzle board, depending on the registers available, their size, and maybe aliasing[9] Then, he transforms the program to what he calls the "elementary representation". Finally, for each instruction in the program, a new puzzle board is instanciated and puzzle pieces are created according to the variables whose live range end at that instruction, start at that instruction, or span that instruction. Register-free spilling is then equivalent to solving the puzzle, which can be done in linear time.

b. Spilling and coalescing

Spilling (NP-complete in this scheme) and coalescing are handled by heuristics inspired by previous papers. The algorithm takes advantage of its prior knowledge of the "real" program to better handle the "elementary" program.

c. Performance

There is a clear tie between puzzle solving and iterated register coalescing. The algorithm is definitely competitive with the standard Appel-George method, and further improvements are to be expected as this is a very recent development. This is a topic that is to be followed closely in the next years, for it may lead to further improvements.

It is to be noted that as of now, some architecture with complicated aliasing of registers cannot be solved by this method. The author gives the example of SparcV8. This is a serious limitation that one has to take into account when considering an implementation.

[9] Registers are said to be aliased when they are not independent from each other. For instance, on x86, the `eax` register is 32-bit wide, but the `ax` register is not a 16-bit register of its own: it's the lower 16 bits of the `eax` register. Similarly, `ah` and `al` are aliased registers since they are nothing but the higher and lower part of `ax`.

PART IV
USING INTEGER LINEAR PROGRAMMING

The Pentium series, also called x86 series, has many well-known features:

- accumulation of over 30 years of instructions (plus extra instruction sets MMX, MMX2, SSE1 to SSE5...);
- retro-compatibility, including old 8-bit and 16-bit registers;
- none of the modern features found in "modern" processors: register rotation, explicit parallelism, branch prediction...
- and so much more!

All those reasons justify the Pentium series' incredible success to these days. One notable feature is that all the Pentiums only have eight 32-bit registers. But two registers are always used: one for the base address of the procedure, the other one for the stack pointer (`esp`). As a consequence, only six (!) registers are available. It turned out that iterated register coalescing performed poorly in this context, which led to other studies.

Starting in 2000, efforts have been made to deal better with such specific case. These efforts are closely linked to Integer Linear Programming, as we will see.

1 Separating spilling and coalescing

Appel and George, while compiling SML programs, realized that their register allocator was performing poorly in the specific context of the Pentium series. They first tried to split their live ranges into many more smaller live ranges (in the SSA spirit, but not according to SSA rules) so the "relevant" parts of variables are kept in registers while the variable, when "unused", can be spilled. This also gave poor results[10]. As a consequence, they devised a new approach to register coalescing [AG01] to fix the performance problem encountered with Pentiums.

There has been (and there still is) a large part of research on register allocation focused on *integer linear programming*. The idea is to use the well-known, optimized solvers for ILP to solve register allocation as an ILP problem. However, this approach is still too slow [GW96]. What Appel and George did was split the main problem into two subproblems: spilling on the one hand, coalescing and coloring on the other hand.

For the spilling problem, they use 0/1 ILP (that is to say, an integer linear programming problem where variables can only take values 0 and 1). For each program, they create an associated ILP problem that is then given to a general solver. This approach is extremely rewarding, for even in a big program, the answer to the question "which variables should be kept in registers and which variables should be spilled?" can be found in a few minutes. This speed is in no way comparable to linear scan for instance, but we must keep in mind that the answer to the question is optimal! Moreover, empirical evidence tends to show that the complexity is close to linear ($\mathcal{O}(n^{1.3})$).

However, this leaves two very important questions to answer:

- which algorithm can we use for coalescing and coloring? (we only have solved spilling)
- if I have an optimal solution to both subproblems, will it be an optimal solution to the global problem?

The latter question admits no theoretical answer yet, but empirical evidence tends to convince us that the global solution will also be excellent.

The former question is more interesting, and is still an open problem: this is the coalescing challenge `http://www.cs.princeton.edu/~appel/coalesce/`.

2 Which coalescer?

The ultimate solution has not been found yet. The first solution was offered by Appel and George, and was called "Optimal register coalescing". It is described in their November 2000 version of their paper (that updates their August 2000 paper) [AG01]. Once again, using linear programming, they model coalescing and coloring as an integer linear programming problem. The ILP problem variables have two indices: when $x_{v,j}$ is equal to 1, that means that variable v will be stored in register j. This is indeed a very straightforward way to model the problem, and the ILP solver allows them to derive a solution to the problem, given the constraints of the underlying architecture. The coloring+coalescing found is not optimal, but in practice, it is good.

a. Optimistic coalescing

However, this solution is in no way feasible because is it too slow (many hours for small programs). Nevertheless, it serves to demonstrate a key point: by splitting the main problem into two subproblems, we do not "break" the feasibility of the main problem. It is still possible to find good solutions to each of the subproblems.

What happens is the first subproblem for which they give a practical solution, i.e. the spilling problem, generates a second subproblem, i.e. the coalescing and coloring problem, and this second problem admits good solutions. And given that by combining those two good solutions, we obtain still a good solution to the main problem, it would be very worthwile to find a good algorithm for optimal solutions to the second subproblem.

b. Cutting-plane algorithm

Two years ago, a solution to the optimal coalescing challenge was proposed [GH07]. This is a generic coalescer that can be used in any register allocator that works according to the scheme in figure 7 on the next page.

The colorer-coalescer gives *optimal results* and is faster than the "optimistic coalescer" proposed by Appel and

[10] because given the high interferences, no conservative coalescing was possible

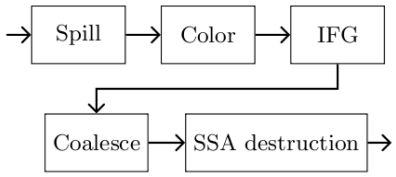

Figure 7: A SSA-based register allocator (it is not iterative!)

George. Once again, it works using integer linear programming. Because solving the whole problem to find the optimal solution would be too slow, they use a series of clever improvements to simplify the full problem while still retaining the key features that lead to an optimal solution when solved. They also tweak the ILP solver with hints on bounds for some variables, hence saving time.

⋆ Performance

This algorithm performs much better than Appel's "optimistic coalescer" and the average time for register allocation is around 5 seconds on the test case given for the optimal coalescing challenge. However, as it is the case for integer linear problems, in some cases, the register allocation takes too long. The authors suggest to halt the computation and resort to a more "standard" solver that is guaranteed not to take forever.

⋆ A few remarks

As far as we know, this algorithm has not been implemented yet in a real-world compiler. The possible reasons are: complexity, need to implement a "backup" coalescer in case the solver takes too much time... And the very idea of maybe not getting the result in time is unusual for a "real-world" software. It is possible that some further refinements may lead to even better performance so as to make it a choice candidate for commercial compilers.

PART V
JIT COMPILATION

We will end this survey with a quick discussion of which algorithms to consider when one wishes to implement a register allocator for JIT compilation.

1 Linear scan

The first obvious answer is to use linear scan. Indeed, this is a choice candidate for embedded environment. It is very resource-friendly, requires only one pass, and implies very little overhead. It is guaranteed to find a solution after one pass (and not an undetermined number of passes for other algorithms). For all those reasons, this is probably the most used algorithm in this context. However, graph coloring is also worth a look.

2 Adapting graph coloring

Graph coloring cannot be used "as is" for JIT environment. The very simple reason is that it is iterative and requires to rebuild the graph after each round of spill, and this is very time-consuming. ILP can clearly not be used: too slow, requiring too much power, this is not adapted to an embedded environment.

What [CD06] suggests to do is to adapt graph coloring. Instead of rebuilding the graph after each round of spill, we modify the graph in place to take into account the IL modifications. Chaitin dismissed this approach because of its complexity. Indeed, this is quite a difficult process. However, if we allow ourselves to be too conservative, hence introducing some imprecision in the graph, then this is feasible. The authors call it a "lossy allocator".

This algorithms gives very good results when taking into account the compile-time and the run-time. It outperforms linear scan in most programs, and obviously beats Briggs' algorithm. Indeed, it turns out that the most time-consuming step in Briggs' algorithm is (re-)building the graph. Naturally, this algorithm "tailored for runtime compilation" is designed to perform well.

However, this requires an added complexity that cannot compete with the incredible simplicity of linear scan. As such, it might be that this algorithm, in spite of its excellent performance, might not be chosen because of implementation details. However, the authors have provided a reference implementation for, again, LLVM, which will undoubtedly boost its adoption.

PART VI CONCLUSION

After reading the survey, we hope that the reader will know which algorithm to choose depending on his specific needs. We can quickly summarize the results of this survey as follows:

- if you have no time, just use linear scan;
- if you have more time, then:
  - if you are in a JIT context, use graph-coloring with the lossy allocator;
  - otherwise:
    - if performance is critical (especially on x86), use optimal spilling from Appel-George with optimal coalescing from Grund and Hack
    - if you like new things, use register allocation via coloring of chordal graphs, or even by puzzle-solving
    - if you just want a widely tested, robust-performanced, well-documented algorithm, use the classical iterated register coalescing from Appel and George.

About simulation

We have not implemented any algorithm in this survey. Implementing a register allocator takes several months, and requires to learn the internals of a compiler (let us say, LLVM for instance). Moreover, the authors do not always provide reference implementations for LLVM. Some implementations are outdated, require heavy patching of LLVM to work. However, the benchmarks are always provided in the papers. We believed it to be more interesting to review many different algorithms for many different contexts than to waste time implementing an algorithm, which would have been probably a failure.

PART VII BIBLIOGRAPHY

References

[AG01] A.W. Appel and L. George. Optimal spilling for CISC machines with few registers. *ACM SIGPLAN Notices*, 36(5):243–253, 2001.

[BCT94] P. Briggs, K.D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, 1994.

[BDGR] F. Bouchez, A. Darte, C. Guillon, and F. Rastello. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove. In *WDDD 2006, Fifth Annual Workshop on Duplicating, Deconstructing, and Debunking, part of ISCA*, volume 33.

[BDR07a] F. Bouchez, A. Darte, and F. Rastello. On the complexity of register coalescing. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 102–114. IEEE Computer Society Washington, DC, USA, 2007.

[BDR07b] F. Bouchez, A. Darte, and F. Rastello. On the complexity of spill everywhere under SSA form. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 103–112. ACM New York, NY, USA, 2007.

[CD06] K.D. Cooper and A. Dasgupta. Tailoring graph-coloring register allocation for runtime compilation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 39–49. IEEE Computer Society Washington, DC, USA, 2006.

[Cha] GJ Chaitin. REGISTER ALLOCATION & SPILLING VIA GRAPH COLORING.

[GA96] L. George and A.W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):300–324, 1996.

[GH07] D. Grund and S. Hack. A fast cutting-plane algorithm for optimal coalescing. *Lecture Notes in Computer Science*, 4420:111, 2007.

[GW96] D.W. Goodwin and K.D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software: Practice and Experience*, 26(8), 1996.

[HG06] S. Hack and G. Goos. Optimal register allocation for SSA-form programs in polynomial time. *Information Processing Letters*, 98(4):150–155, 2006.

VII. BIBLIOGRAPHY

[LA04] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society Washington, DC, USA, 2004.

[Ler06] X. Leroy. The Compcert certified compiler back-end–commented Coq development. *Available on-line at http://cristal.inria.fr/xleroy*, 2006.

[PM04] J. Park and S.M. Moon. Optimistic register coalescing. *ACM Transactions on Programming Languages and Systems*, 26(4):735–765, 2004.

[PP] F.M.Q. Pereira and J. Palsberg. Register allocation after classical SSA elimination is NP-complete. *def (a)*, 11:21.

[PP05] F.M.Q. Pereira and J. Palsberg. Register allocation via coloring of chordal graphs. *LECTURE NOTES IN COMPUTER SCIENCE*, 3780:315, 2005.

[PP08] F.M.Q. Pereira and J. Palsberg. Register allocation by puzzle solving. 2008.

[PS99] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.

[SRH04] M.D. Smith, N. Ramsey, and G. Holloway. A generalized algorithm for graph-coloring register allocation. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288. ACM New York, NY, USA, 2004.