N° d'ordre : ...

N° attribué par la bibliothèque : ...

THÈSE d'HABILITATION à DIRIGER des RECHERCHES

en vue d'obtenir le grade de

HABILITATION à DIRIGER des RECHERCHES

École Normale Supérieure de Lyon

spécialité : Informatique

Laboratoire de l'Informatique du Parallélisme

présentée et soutenue publiquement le 07/12/12

par Monsieur Fabrice RASTELLO

On Sparse Intermediate Representations: Some Structural Properties and Applications to Just In Time Compilation

---

Après avis de : Monsieur Erik ALTMAN, Membre/Rapporteur
Monsieur Scott MAHLKE, Rapporteur
Monsieur Vivek SARKAR, Membre/Rapporteur

Devant la Commission d'examen formée de :
Monsieur Erik ALTMAN, Membre/Rapporteur
Monsieur Philippe CLAUSS, Membre
Monsieur Alain DARTE, Membre
Monsieur Xavier LEROY, Membre
Monsieur Scott MAHLKE, Membre/Rapporteur
Monsieur Keshav PINGALI, Membre
Monsieur Vivek SARKAR, Membre/Rapporteur

Contents

1 Introduction

1.1 Foreword

Compilation is an old activity, in particular back-end code optimizations. Fortunately for me, I am not old enough to talk about the birth of compilation, so I will talk instead about why and how my researches lead me to revisit some of the oldest compiler optimization problems such as register allocation or liveness analysis. Lucky I am, the size of a thesis is too constrained to let me report all the results we accumulated since I defended my PhD. As a consequence this thesis is a selection, and the goal of the following introduction is to motivate the choice, provide the research context, and give a brief summary of my whole contribution.

In terms of layout, this introduction which goal is to tease the reader's curiosity and motivate him to go beyond this first chapter, falls into four main sections.

1. I will first draw the overall context that makes me believe that in a world where parallelism is ubiquitous, and when fundings for GP-GPU have left the place to exa-scale computing, without mentioning Amdahl's low, compiler optimizations for mono-processors still require efforts.
2. I will then summaries my contributions during the last 10 years, providing cross-references to the corresponding chapters developed in this dissertation.
3. I will then present and motivate my selection which goal is roughly to convince the reader that the following sentence I might have pronounced 10 years ago is not necessarily true (if not completely stupid): "Because of time constraints, JIT compilation summarizes to implementing the most naive greedy heuristics".
4. Last, I will adventure myself on the development of perspective ideas. I will first restrict to simple extensions of the materials developed in this dissertation, then enlarge to my current research project made up of trace analysis and loop transformations i.e. out of (the scope of) static single assignment form...

In terms of content, this dissertation can be read with two different objectives.

1. The first follows the viewpoint of a practitioner: this thesis contains all the required steps to implement a tree scan coalescing that generalizes the original linear scan of Poletto and Sarkar [113]. The objective of not trading simplicity with quality is reached thanks to the use of static-single assignment form. It allows to fully revisit liveness analysis and propose a new technique that we call liveness check; it allows to revisit the ultimate notion of interferences of Chaitin [42] and propose a lightweight and precise interference

check; it finally gives the foundations for revisiting scan-like register allocators [63; 52; 143] providing a clean handling of shuffle code.

2. The second objective follows the viewpoint of a graph theoretician. The chosen examples should illustrate, in the context of just-in-time compilation, the importance of data-structures for designing fast and reliable compiler optimizations. In particular, I take this opportunity to put the light on the powerful theory of graph labeling that I believe is currently under-exploited by the compiler community.

Each chapter corresponds to an already, or on the way to be, submitted paper. I did minor changes to make it fit the format of the thesis. I removed some redundancies, added a few cross-references and link between the chapters but no more. If there is a common thread running through the consecutive chapters, each of them is self-content. However, the chosen format implies that parts of the work including the writing belong to the colleagues or students that have been/are participating to it. Those are (in alphabetical order) Benoit Boissinot(former PhD student) for Chapters 2, 3, 4, and 5; Florian Brandner (former psot-doc) for Chapter 2; Philip Brisk for Chapter 4; Quentin Colombet (PhD student) for Chapter 4; Alain Darte for Chapters 2, and 3; Benoit Dupont-de-Dinechin for Chapters 2, and 3; Francois de Ferrière (Colleague from STMicroelectronics) for Chapter 3; Christophe Guillon (Colleague from STMicroelectronics) for Chapter 3; Sebastian Hack (former post-doc) for Chapters 2, and 4; Fernando Perreira for Chapter 5; Andre Tavares (former student) for Chapter 5.

1.2 Research context

We first give some elements that explain why the development of embedded systems makes compilation come back as a research topic. We then detail the code optimizations that we are interested in, both for aggressive and just-in-time compilation.

1.2.1 Embedded Systems and the Revival of Compilation & Code Optimizations

Compilation for embedded processors is either aggressive or just in time. *Aggressive compilation* consists in allowing more time to implement costly solutions (so, looking for complete, even expensive, studies is mandatory): the compiled program is loaded in permanent memory and the compilation time is not so significant. For embedded systems, code size and energy consumption usually have a critical impact on the cost and the quality of the final product, hence the application is cross-compiled, i.e., compiled on a powerful platform distinct from the target processor. *Just-In-Time (JIT) compilation*, on the other hand, corresponds to compiling bytecode on demand on the target processor. The code can be uploaded or sold separately on a flash memory. Compilation is performed at load time or even dynamically during execution. The heuristics, constrained by time and limited resources, cannot be too aggressive: they must be fast enough. In this context, our goal is to contribute to the understanding of combinatorial problems that arise in compilation for embedded processors (e.g., in opcode selection, SSA conversion, register allocation, or in code placement in the instruction cache) so as to to derive both aggressive heuristics and JIT techniques.

Applications for embedded computing systems generate complex programs and need more and more processing power. This evolution is driven, among others, by the increasing impact of digital television, the first instances of UMTS networks, and the increasing size of digital supports, like recordable DVD, and even Internet applications. Furthermore, standards are evolving very rapidly (see for instance the successive versions of MPEG). As a consequence, the industry has rediscovered the interest of programmable structures, whose flexibility more than compensates for their larger size and power consumption. The appliance provider has a choice between hard-wired structures (ASIC), special-purpose processors (ASIP), or (quasi) general-purpose processors (DSP for multimedia applications). Our cooperation with STMicroelectronics lead us to investigate the last solution, as implemented in the ST100 (DSP processor) and the ST200 (VLIW DSP processor) family for example. Compilation and, in particular, back-end code optimizations find a second life in the context of such embedded computing systems.

At the heart of this progress is the concept of *virtualization*, which is the key for more portability, more simplicity, more reliability, and of course more security. This concept, implemented through binary translation, just-in-time compilation, etc., consists in hiding the architecture-dependent features as far as possible during the compilation process. It has been used for quite a long time for servers such as HotSpot, a bit more recently for workstations, and it is quite recent for embedded computing for reasons we now explain.

As previously mentioned, the definition of "*embedded systems*" is rather imprecise. However, one can at least agree on the following features:

- Even for processors that are programmable (as opposed to hardware accelerators), processors have some architectural specificities, and are very diverse;
- Many processors (but not all of them) have limited resources, in particular in terms of memory;
- For some processors, power consumption is an issue;
- In some cases, aggressive compilation (through cross-compilation) is possible, and even highly desirable for important functions.

This diversity is one of the reason why virtualization, which starts to be more mature, is becoming more and more common in programmable embedded systems, in particular through CIL (a standardization of MSIL). This implies a late compilation of programs, through just-in-time, including dynamic compilation. Some people even think that dynamic compilation, which can have more information because performed at run-time, can outperform the performances of "ahead-of-time" compilation.

Performing code generation (and some higher-level optimizations) in a late phase is potentially advantageous, as it can exploit architectural specificities and run-time program information such as constants and aliasing, but it is more constrained in terms of time and available resources. Indeed, the processor that performs the late compilation phase is, *a priori*, less powerful (in terms of memory for example) than a processor used for cross-compilation. The challenge is thus to spread the compilation process in time by deferring some optimizations ("*deferred compilation*") and by propagating some information for those whose computation is expensive ("split compilation"). Classically, a compiler has to deal with different intermediate representations (IR) where high-level information (i.e., more target-independent) coexist with low-level information. The *split compilation* has to solve a similar problem where, this time, the compactness of the information representation, and thus its relevance, is also an important criterion. Indeed, the IR is evolving not only from

a target-independent description to a target-dependent one, but also from a situation where the compilation time is almost unlimited (cross-compilation) to one where any type of resource is limited. This is also a reason why static single assignment (SSA) is becoming specific to embedded compilation. Indeed, SSA is a sparse (i.e., compact) representation of some of the information (such as liveness). In other words, if time constraints are common to all JIT compilers (not only for embedded computing), the benefit of using SSA is also in terms of its good ratio pertinence/storage of information. It also enables to simplify algorithms, which is also important for increasing the reliability of the compiler.

One of the goal of this thesis is to illustrate the new challenges related to the multiple trade-offs that compiler designers face in this new context. The computation and representation of structural properties of both SSA (and some of its extensions) and Control Flow graphs will compose the heart of the dissertation.

1.2.2 SSA form and Work Methodology

The *static single assignment (SSA) form* is an intermediate representation in which multiplexers (called ϕ functions) are used to merge values at a "join" point in the control graph. The SSA form is becoming more and more popular in compilers due to its properties that speed up optimizations and make them easier to implement. Unfortunately, the SSA form is not machine code and ϕ functions have to be replaced, at the end of the process, by register-to-register copy instructions on control flow edges. Naive methods for destructing SSA generate many useless copies (live-range splitting) and additional goto instructions (edge splitting). Our past experience on SSA destruction were a bit frustrating. Although our strategy improved previous translation schemes, the final execution times were sometimes disappointing. The reason is that a too aggressive coalescing (removal of register copies) can degrade the next compilation phase, register allocation, by increasing spilling (load/store insertions). However, thanks to Alain Darte who pointed out in 2002 that the interference graph of variables in SSA is chordal (i.e. every circuit of length more than 4 has at least a chord), we quickly understood that SSA could be useful for register allocation as coloring a chordal graph is polynomial. This remark was the starting point of a deep and long research on register allocation, including the absorption of the literature on this very old problem. Before claiming anything new, we needed to study this problem on all aspects, to check that what we thought we could improve on one side was not destroyed due to another unexpected aspect.

In this context, our goal was more generally to contribute to the understanding of combinatorial problems that arise in compilation for embedded processors (e.g., in opcode selection, SSA conversion, register allocation, code placement in the instruction cache) to derive both aggressive heuristics and JIT techniques. A first specificity of our work is that we always aim at adding a theoretical value on the problems we address (using graph theory, NP-completeness), even for problems that can appear "old" (such as register allocation). The second specificity is that, thanks to the collaboration with STMicroelectronics (and more recently with Kalray), we can implement and test our techniques directly within an industrial compiler. After clarifying, debunking, understanding the key issues that make the addressed problem hard, we first develop potentially-costly solutions (e.g., using integer linear programing) for aggressive compilation. This process allows us to confront the theory with the practice and provides a basis for designing and evaluating JIT solutions.

1.3 Scientific achievements

The most visible result concerns the revisiting of register allocation in the light of SSA. Except the work on the parallelization of saturated reductions [62] and the one on instruction cache optimization [71], all other contributions can be articulated around the design of a faster, cleaner, more robust, and optionally more efficient register allocation. To this end, our research in this topic can be structured along four main axes.

1. the study of some fundamental properties of SSA form and its semantic; the revisiting of the notion of SSI [6; 128] (static single information property, a generalization of SSA to support both forward and backward data-flow analyses) and the debunking of the corresponding existing form; the design of some construction and destruction algorithms (along with some clarifications and debunking) for those forms; the developments of new algorithms for liveness computation.
2. a deep study of the complexity of spilling (under SSA) and the design of an "optimal" algorithm and some heuristics.
3. a deep study of the complexity of the coalescing problem (under SSA) and the design of a graph coloring based heuristic that handles both aliasing and register constraints; the design of an effective heuristic in the context of JIT compilation.
4. the development of a formalism to eliminate/move shuffle code and get rid of ϕ-functions on an allocated code.

1.3.1 Complexity Results

We first revisited the initial NP-completeness proof given, in 1981, by Chaitin et al. to show that it was used wrongly to justify heuristics for problems that it does not cover. Our study shows that, somehow surprisingly, the NP-completeness of register allocation is not due to the coloring phase, as may be suggested by a misinterpretation of the reduction of Chaitin et al. from graph k-coloring. If live-range splitting is taken into account, deciding if k registers are enough or if some spilling is necessary is not as hard as one might think. In fact, the NP-completeness of register allocation is due to three factors: special edges (called critical or abnormal) in the control flow graph, the optimization of spilling costs (if k registers are not enough) and of coalescing costs, i.e., which live-ranges should be fused while keeping the graph k-colorable. These results have been presented at WDDD'06 [26] (the "debunking" workshop) and LCPC'06 [27]. We also made a complete study of spill-everywhere spilling problems (LCTES'07 [29]) and of coalescing problems (CGO'07 [28]). In both cases, we analyzed the complexity in terms of graph structures, especially for chordal graphs and k-greedy graphs, those used in register allocation based on graph coloring. These studies are the basis to develop new register allocation schemes based on two phases, a first spilling phase, followed by a coloring/coalescing/splitting phase with no additional spills.

1.3.2 Properties

Liveness analysis in SSA (Chapter 2) One important source of error of prior implementations of SSA destruction is a bad understanding of the liveness of ϕ-related variables. We proposed at CGO'08 [22] a method to check the liveness of a variable

at a given program point. Our method is specialized to SSA and survives all program transformations other than changes of the control-flow graph structure. As a bonus, it is less memory consuming and, depending on the client, usually faster. *This so called liveness check technique allows to avoid the memory and time costly liveness sets when designing register allocation heuristics in the context of JIT compilation.* However, the SSA properties we identified for this liveness check allowed us to revisit the problem of computing liveness sets, too. By exploiting the dominance property of (strict) SSA form and the concept of loop-nesting forest, we designed a two-phase data-flow algorithm. Compared to traditional iterative data-flow approaches, which perform updates until a fixed point is reached, our algorithm, presented at APLAS'11 [19], is twice faster on average than the fastest algorithm (Cooper). *The overall technique is reported in Chapter 2.*

Interferences – Revisiting SSI (Chapters 3 and 5) As previously mentioned, one of the important properties of SSA is that graph coloring under SSA is polynomial because the corresponding interference graph is chordal. Even more interesting is the fact that this properties also allows to avoid the construct of an interference graph through the use of an interference check instead. Basically, two variables would be considered to interfere if the live-range of one intersects the definition point of the other. Although it suffices for correctness, this is a fairly restrictive definition of interference, based on static considerations. The ultimate notion of interference, that is obviously undecidable because of a reduction to the Halting problem, should decide for two distinct variables whether there exists an execution for which they simultaneously hold two different values. SSA form again, allows to improve the quality of the interference check in a very elegant way. *The overall technique for a good quality interference check that is currently used in our register allocator is reported in Chapter 3.*

What first attracted our curiosity to SSI was the claim that the interference graph in SSI is an interval graph. Our debunking paper [20] clarifies a number of mistakes on SSI and provides a proof (much harder than the initial proof, which was completely wrong) for this interval graph property. We also revisited SSI for its theoretical ability to perform both forward and backward sparse data flow analysis. *This work, reported in Chapter 5, aims at organizing the zoo of existing program representations (SSA, SSI, e-SSA, SSU, etc.) that exploit live-range splitting (e.g., with ϕ and σ functions) to enforce a static single information property (i.e., valid on a whole live-range).* While being more general and most of the time sparser than the existing sparse graphs or program representations, it covers a wide range of data flow problems, and is compatible with many propagation/interpretation engines.

1.3.3 Coalescing

Conservative Coalescing (Chapter 4) The effectiveness of the decoupled approach depends on the ability to cope efficiently, during the coloring phase (coalescing), with the shuffle code (register-to-register copies, edge splitting) introduced by the repairing phase (ϕ-functions replacement, register constraints handling, etc.). Our first results [28] were devoted to the complexity of coalescing problems (aggressive, conservative, incremental, and optimistic), discussing also on the structure of the interference graph (arbitrary, chordal, or k-colorable in a greedy fashion). This study was extremely useful to point out where the complexity comes from. In [30] – a more practical paper – we improved the de-coalescing phase of an optimistic

approach and designed an advanced incremental conservative approach, which, contradicting the common belief, turned out to be simple to implement and close to optimal. A good context to stress its performances was to apply it in the context of register aliasing, which we address in [135] with the introduction of a "semi-elementary form", generalizing the "puzzle" approach of Pereira and Palsberg. With the democratization of SIMD instruction set architectures, handling register aliasing will become critical even though current compilers are not mature enough to fully expose it. The last step towards the design of a practical SSA-based "coloring" algorithm – generalization of linear scan – was the efficient handling of register constraints [48]. Thanks to the concept of post-repairing of violated register constraint, the spirit of decoupled register allocation can be kept, i.e., with spilling and coloring as simple as possible, without tricky patches to handle special cases of the instruction set architecture. The cost of repairing (as for register-to-register copies used to get rid of ϕ-functions) is encapsulated in the coloring objective function, through affinities and antipathies (negative weight affinities). *Chapter 4 applies this method to develop both a graph-based approach (extension of conservative coalescing to handle register antipathies) and a scan-based decoupled approach (new tree scan coalescing). In particular it presents a elegant and simple generalization of the iterated register coalescing scheme [68] to handle antipathies, through the process of conservative alienation that adds interferences between variables only when the colorability of the interference graph is preserved.*

Aggressive Coalescing (Chapter 3) Especially for the design of a fast heuristics, experiments have shown that it is usually profitable to use the result of an aggressive coalescing (as opposed to conservative, aggressive coalescing, that is easier to compute, coalesces variables regardless of the colorability of the obtained graph) to bias the computation of a conservative one. This is precisely the approach advocated in our tree scan coalescing described in Chapter 4. As outlined by Sreedhar in [130] aggressive coalescing under SSA and optimizing the insertion of copy instruction during SSA destruction are equivalent problems. The goal of SSA destruction is to get rid of ϕ-functions by replacing them by register-to-register copy instructions. Naive methods for destructing SSA, when correct, generate many useless copies (live-range splitting), but also relies on the ability of disambiguating indirect jumps for splitting edges. We addressed three issues: correctness (in the presence of register constraints [118]), code quality (elimination of useless copies, i.e., aggressive coalescing), algorithm efficiency (speed and memory footprint). Our most recent method [21], separates the issues of correctness and optimization, which makes it conceptually simpler and more robust than previous approaches that were often based on "patches". This correctness issue was, for a long time, a slowing factor to the development of SSA (e.g., bugs in GCC and Jikes). Also, by exploiting SSA properties (in particular with the liveness and interference *check* algorithms mentioned above), our algorithm outperforms the speed of the best algorithm so far (Sreedhar [130]) by 2x and reduces the memory footprint by 10x. *With the goal of designing an even faster heuristic, Chapter 3 revisits the dominator-tree based technique of Budimlic et al. in the same spirit than [21], i.e. tackling correctness, code quality, and efficiency.*

Parallel copies All decoupled approaches, and even SSA destruction, rely on shuffle code represented as parallel copies (involving registers but also memory slots) in

basic blocks and also, implicitly, on edges. To optimize such copies, we proposed a new back-end optimization called parallel copy motion [24]. The technique is to move copy instructions in a register-allocated code from a program point, possibly an edge, to another. The elegant formalism developed for this purpose allows to let a copy just "traverse" any instruction of a basic block, except those with conflicting register constraints. This contrasts with traditional schedulers that would have to preserve data dependences, and thus limit the possible movements of the copy. The second strength of the technique is that as it traverses instructions, a parallel copy might shrink or even be fully eliminated. In the same spirit than the recoloring technique advocated by Hack et al. in [73], the parallel copy motion plays the role of a incremental coalescing on an already colored code, and is thus well suited for just-in-time compilation as it can be interrupted at any time.

As the interplay of this optimization with the scheduler is high, we pushed this idea further to perform code motion (of copies) on register-allocated data dependence graphs. This technique [31] investigated by Florian Brandner (Post-doctoral student) et Quentin Colombet (PhD student) can eliminate useless copies and reorder instructions, while preserving a valid register assignment. It is a step forward the design of register-pressure aware schedulers.

1.3.3.1 Spilling

The fact that the interference graph in SSA is chordal enables the design of a decoupled allocator: the spilling phase that stores variables to memory to lower the register pressure can be done *before* the coloring phase that assigns the other variables to registers. This decoupling opened the door for new spilling strategies. We first made an exhaustive study on how SSA impacts the complexity of "spill everywhere" (i.e., the whole live-range of a spilled variable is in memory). Contrarily to our initial hopes, most problems remain NP-complete [29]. However, the fact they are polynomial for a fixed number of registers suggested spill-everywhere heuristics that incrementally solve, in "polynomial" time, the allocation problem with few registers, then "stack" the solutions. We applied this principle for split compilation [58]: an ahead-of-time stacking algorithm drives, through portable bytecode annotations, the decisions of a light online JIT algorithm that adapts the allocation to the right target. We are also currently designing a purely JIT "stacking" solution for spill everywhere: with layers of width one, each step can be done optimally in linear time. Experiments that show the quasi-optimality [c31] of our solution let us believe that the stacking approach is very promising for designing both lightweight and aggressive load-store optimization algorithms.

In parallel, to better understand spilling in its generality (not just spill everywhere), Quentin Colombet developed an integer linear programming formulation [49], more accurate and expressive than previous approaches, that exploits the decoupling between spilling and coalescing. The experimental comparison, in the STMicroelectronics compiler, of various heuristics to this "optimal" solution draws, among others, the following conclusions: a) significant savings can still be obtained in terms of static spill costs, cache misses, and dynamic instruction counts; b) rematerialization is extremely important and SSA can pay off here; c) SSA complicates the formulation of optimal spilling, because of memory coalescing of interfering variables; c) micro-architectural features are significant and thus should be accounted for in the model (but it is never the case). This deep study is still the first step for designing new aggressive and JIT spilling strategies.

1.3.4 Experimental environment

LAO Our aggressive optimization techniques are all implemented in stand-alone experimental tools (as for example for register coalescing algorithms) or within LAO, the back-end compiler of STMicroelectronics, or both. They concern SSA construction and destruction, instruction-cache optimizations, register allocation.

LAO is an open-source VLIW code generator used by STMicroelectronics to complement the SGI Pro64 (OPEN64) framework in several production compilers. In this configuration, the OPEN64 compiler generates the code up to register allocation. More important, the LAO code generator is also used by a just-in-time compiler for the Common Language Infrastructure (CLI) program representation. For this reason, it has been carefully profiled and tuned. Since recently, LAO accepts more front-ends (such as LLVM or GCC) thanks to the new interface that uses TireX described below.

Most of the time, our target processor is a commercial media-processing embedded VLIW architecture (ST200) from the Lx family [64] of processors issuing up to 4 instructions per cycle over 6 functional units consisting of 1 load-store unit, 1 branch unit, and 4 arithmetic units.

MiniR. TireX Most compilers define their own intermediate representation (IR) to be able to work on a program. Sometimes, they even use a different representation for each representation level, from source code parsing to the final object code generation. MinIR (Minimalist Intermediate Representation) is a new intermediate representation, designed to ease the interconnection of compilers, static analyzers, code generators, and other tools. In addition to the specification of MinIR, generic core tools have been developed to offer a basic toolkit and to help the connection of client tools. MinIR generators exist for several compilers, and different analyzers are developed as a testbed to rapidly prototype different static analyses over SSA code. This new common format enables the comparison of the code generator of several production compilers, and simplifies the connection of external tools to existing compilers.

MinIR has been extended into Tirex, a Textual Intermediate Representation for EXchanging target-level information between compiler optimizers and whole or parts of code generators (aka compiler back-end). The first motivation for this intermediate representation is to factor target-specific compiler optimizations into a single component, in case several compilers need to be maintained for a particular target (e.g., operating system compiler and application code compiler). Another motivation is to reduce the run-time cost of JIT compilation and of mixed mode execution, since the program to compile is already in a representation lowered to the level of the target processor. Besides the lowering at the target level, the extensions of MinIR include the program data stream and loop scoped information. Tirex is currently produced by the Open64/Path64 and the LLVM compilers, with a GCC producer under work. It is consumed by the LAO code generator.

Detailed information, generic core tools, and LLVM IR based generator for MinIR are available at `http://www.assembla.com/spaces/minir-dev/wiki`. Open64/Path64 emitter for Tirex and its LAO back-end are available at `https://compilation.ens-lyon.fr/`.

1.4 Sparse Intermediate Representations and Data Structures for Just In Time Compilation

As already mentioned, I currently believe that the challenges associated to the design of fast, memory frugal, and still efficient compiler optimizations, is highly related to trade-offs between storing/recomputing/updating some relevant information. By information we include, for example aliasing, but also def-use chains, or even loop-nesting forest. In this thesis we will over-exploit the information of dominance. Dominance is an order relation between nodes of the control flow graph which transitive reduction is a tree (the dominator-tree). A tree-structure which is more general than a chain is quite interesting algorithmically, as many problems that are NP-complete even on a directed acyclic graph (DAG) become polynomial on a tree (such as spilling with few registers [29] or code selection). Indeed, dynamic programing or depth-first-search (DFS) traversal based approaches are made possible. The best illustration of this point is the tree-scan coloring described in Chapter 4 that generalizes with identical complexity and better code quality the well know linear-scan register allocation scheme.

SSA is appealing mostly because it provides def-use chain information almost for free and because it is, in its most common flavor (T-SSA for Transformed-SSA as opposed to Conventional SSA; See Chapter 3), easy to update. Let us also outline that SSA means that statically i.e. *textually*, each variable is defined only once. This provides referential transparency, which enables the association of value information to variables themselves. Global Value Numbering [123], or copy folding [34] that also exploits the dominance property (see below) to perform copy propagation linearly, are nice applications of this property. The same technique as for copy-folding is used in Chapter 3 to improve in one shot (as opposed to Chaitin in [42]) the accuracy of the interference.

Data flow analysis is a good example to illustrate the trade-off we are talking about. Any cross-compiler can afford multiple traversal of the control flow graph and even recompute multiple times the same information, once for each analysis/-transformation. This is usually not possible in a JIT context although path expression elimination schemes [125] have very good theoretical complexity. Several forward data flow analysis can directly take advantage of the referential transparency of SSA mentioned above, but not all of them. SSI form, as proposed by Ananian [6] in his thesis, is a step toward extending SSA so as to enable, with the same simplicity both forward and backward analysis. But this is to the cost of a substantial increase in the number of variables and instructions while many analysis such as the numerous ones that make use of value information on conditional branches cannot take advantage of it. As the number of variables directly impacts the memory consumption and the complexity of every analysis and transformation, while the simplicity objective might have been reached, the solution might show to be impractical in a JIT context. Chapter 5 addresses this problem by revisiting the SSI form.

Liveness analysis can be treated differently: as the lattice is restricted to boolean values, the problem can be expressed as manipulation of properties of the control flow graph (related to path expressions). Liveness is quite specific analysis as it is required by several back-end level transformations in addition to the fact that it usually has to be updated. The mechanism developed to make it as efficient as possible is quite sophisticated (but its implementation is quite simple). Similarly to path expression eliminations techniques, it exploits the loop nesting forest [116] (see Sec-

tion 2.3) information which corresponds to a recursive decomposition of the control flow graph into strongly connected components (SCC). But it also exploits the already mentioned tree-structure. One can even go further and exploit the fact that structured code have at most dimension two (see Section 2.3.2). This allows to encode in linear time (instead of the standard quadratic complexity) the reachability of the forward control-flow graph. When the code is not structured, one can conservatively over-approximate reachability (and hence liveness) by mapping the forward CFG to a DAG of dimension two. Mapping it to a DAG of dimension one, would precisely correspond to the over-approximation of liveness made by a linear scan register allocator. The quality of the over-approximation is significantly improved by going from dimension one to dimension two. Chapter 2 illustrates all these points by completely revisiting liveness analysis. Note that as we show in [20], original SSI form [128], allows to map live-ranges to sub-chains of a chain and obtain exact liveness with purely linear complexity also but, as already mentioned, this would be to the cost of a substantial increase in the number of variables.

The above mentioned underlying tree-structure is only obtained when considering the *strict flavor of SSA* which enforces that the unique definition of a variable dominates all its uses. Most existing codes are always strict (sometimes enforced by the programming language itself). This thesis will always consider the SSA form to be strict. The consequence is that live-ranges of variables are sub-trees of the dominator tree. As chordal graphs can be characterized as intersection graphs of sub-trees, this explains from another angle the linear time complexity for exhibiting a k-coloring when exists.

Layout This thesis presents only a partial view of the overall results accumulated during those last 10 years. The choice illustrates the challenges faced by JIT compilation in terms of trade-offs between storing/recomputing/updating relevant information. However, the overall objective of my work is composed of practical considerations and many discussions will be toward the cohabitation between a nice theory and its implementation in a production compiler. In particular one of the objectives was to provide a full and realistic framework for an SSA based decoupled allocator in a JIT context. As liveness analysis and SSA destruction are at the heart of our solution, they compose the first two chapters (Chapters 2 and 3). The design of a good solution for spilling is still a work under progress. For now, one can use the already very well designed approach of Braun et Hack [32]. So we focused on the coalescing part in Chapter 4. This chapter also presents the spirit of our approach that I have not described yet: there is a clear incompatibility between being able to exploit all the nice structural properties listed above and the enforcement of all the internal compiler and architectural constraints. As an example, the colorability of a chordal graph with at least two pre-colored nodes is not polynomial any more. Our strategy consists in expressing the constraints in the objective functions of our optimizations, not has hard constraints: that way the nice environment provided by SSA properties is maintained. It then relies on a post-pass to fix the none fulfilled constraints (that we expect to be very limited), using low-cost, peep-hole like techniques. The last chapter (Chapter 5), is not related to register allocation anymore but comes as a complement to the whole discussion. It should actually be considered as a work in progress, as many problems such as updating are not addressed there.

1.5 Perspectives

This introduction part concludes with some perspective discussions beyond the strict content of this thesis. We will cover three problems.

- *Register allocation* Future prospective works concerning register allocation should mostly concern the spilling part. Our preliminary work on stacking allocation should be pursued to apply the approach up-to-now restricted to spill everywhere to the more realistic load-store optimization problem. This should answer quite efficiently to the question of what to spill. But once we decided to spill a variable, the question of where to place precisely the corresponding load instructions has a high impact on the quality of the generated code. We have several research leads to tackle this problem that we will expose hereafter.
- *Run-time dependence analysis* My current research project deals with doing loop transformations [47; 105] (such as join software pipelining [122] and register tiling [119]) at machine level either in the context of dynamic compilation or static compilation with profiling feedback. There are many cases when the expected huge factor of speedup (mostly by exposing locality and parallelism) enabled by some loop transformations does not show up. This is usually due to a too conservative alias/dependence information. This is the opportunity to show the promised virtues of dynamic compilation or profiling feedback: run-time should help to provide dependence information, at least for the current running execution. This is actually an exciting challenge that I will develop hereafter.
- *Over-approximating reachability in linear time* The graph labeling technique to encode basic-block reachability in the context of liveness analysis is a work under progress. In particular the hope is to find a more general class than structured control flow graphs, that could be accurately encoded using only two linear extensions. This technique can also be applied to the context of run-time analysis as it can be used for the encoding of data dependencies.

1.5.1 Register Allocation

Future prospective works that we will expose here mostly concern the spilling part. However some of our works on the coalescing part were a bit frustrating as we will explain after.

What to spill The story begins with two results stemming from our study on the complexity of spill everywhere under SSA [29]: the incremental allocation problem is polynomial while the incremental spill is NP-Complete. In other words, let r be a fixed integer; let Ω be the maximum amount of simultaneously live variables (MAXLIVE). Finding a set of allocated variables (the ones you keep) of maximum weight and such that the maximum amount of simultaneously live variables is lowered to r can be done in polynomial time (actually exponential in r). On the other hand, finding a set of spilling variables (the one you remove) of minimum weight and such that the maximum amount of simultaneously live variables is lowered to $\Omega - r$ is NP-complete, even for $r = 1$. As a consequence, in the perspective of the design of a spill algorithm, incrementally increasing the register pressure is easier than incrementally lowering it. As we experimented the idea of incremental allocation

in the context of split compilation [58] (the goal was to encode in the bytecode the allocation for several different number of registers for a family of processors), we observed it to be extremely close to the optimal. This lead to the stacking solution [57]. Even if it shows quite impressive results, it is restricted to spill everywhere. There remains a gap of performance [49] between a solution that spills the entire live-range of variables (spill everywhere) with a solution that spills only part of it (load-store).

There are two research leads to exploit this idea of incremental allocation for the load-store optimization problem. The first is to use the result of a spill everywhere as an oracle. The underlying motivation is because the complexity of load-store optimization comes from the asymmetry between loads and stores [65]. Let us illustrate this point, by supposing to simplify, the code to be under SSA form, the store to be placed just after the (unique) definition point, and the loads just before the uses. For a given variable, the cost of the store has to be paid whatever the (non null) number of sub live-ranges are spilled. Hence, a heuristic that would incrementally spill sub live-ranges would favor spilling the one corresponding to a variable already spilled elsewhere. The role of the oracle is to anticipate which variable shall be spilled. In this context, the spill everywhere problem can be solved (without being committed) to provide an insight of which variables are globally the more profitable to spill.

The second research lead is to perform directly the stacking approach on the load-store problem. Unfortunately this is not as straightforward as just saying it: the polynomial complexity of allocation with few registers only holds for a code that fulfills the Static Single Reaching Occurrence (SSRO) property advocated by Philip Brisk. The SSRO form is the SSA form of a code where every use is also considered as a definition. To this end SSRO adds new ϕ-functions to an already under SSA code. SSRO is obviously not magic: when two copy-related variables are both spilled and memory coalesced, the load and the store at the copy could be saved and the copy eliminated. Not surprisingly, taken this gain into account (which is required to make SSA and SSRO solutions equivalent) makes the load-store optimization problem under SSRO NP-complete again. The pertinence of using SSRO depends on the amount of ϕ-functions it adds to the program. As most variables have only two uses, the intuition is that going into SSRO should add only a few additional shuffle code. But this has to be experimentally validated.

Where to insert load instructions Suppose one have designed an efficient algorithm to decide what sub live-ranges to spill. The question of where to place precisely the corresponding load instructions has a high impact on the quality of the generated code. First of all, and obviously, a load outside a loop is preferable to one inside. More generally this optimization is related to the elimination of loads redundancies (a load that can be hoisted to outside the loop is redundant with itself). To improve the quality of a load-store algorithm, the idea is to incorporate and adapt within the spilling algorithm register promotion technique such as the one described in [93].

The second reason why placement of load instructions has a high impact, even within a basic-block, is because memory accesses involve long dependence latencies. The latency of a load, inserted way before the instruction that uses its value could be hidden by the computations executed in between. In other words, the part of a live-range does not have the same spilling cost when ending far or close to a use. One can address this problem during the spilling phase itself, by decomposing live-ranges into two parts: the one that could be spilled almost for free, and the one

which we really want to avoid. Hence the low-cost sub-part of a variable's live-range could first be spilled, and the remaining part of it that involves loads just before the uses could be spilled later on if really required.

Within a basic-block, the placement of load instructions is the role of the scheduling. But this last might just be stuck by some anti-dependencies (the live-range of another variable uses the same register) that prevents the load to be scheduled sooner. *Trying* to avoid this situation is the role of the coloring phase: one need to express that two variables (the loaded one and the ones live in the "costly" region) should, *if possible*, be assigned a different color. This is precisely the semantic of the antipathies introduced in Chapter 4 to cope with repairing.

Aliasing and storage optimization As already mentioned, we believe our work on coalescing to be quite accomplished: our exhaustive study of its complexity [28] gave rise to a very efficient and simple heuristic for conservative coalescing [30]. However, when using the techniques of alienation and the more accurate notion of interference with values (see Chapter 4) the coalescing problem does not show to be that hard. The quality of our tree scan compared to a very aggressive algorithm confirms this observation. Still we had the intuition that coloring/coalescing in the presence of aliasing would be much more difficult. But our work in this context was a bit frustrating, because we were not able to generate cases with substantial amount of aliasing [135]. We still have this intuition, and challenging our algorithm with storage optimization [63] (allocation of arrays in memory) would be very interesting. The difficulty is to find realistic problems where arrays are partitioned and code is tiled to expose data locality. The code used by existing research groups such as the one we borrowed to J. Xue [90] for this purpose are unfortunately too artificial.

1.5.2 Run-time dependence analysis

My current research project deals with doing register tiling jointly with software pipelining and vectorization. As the tuning of such loop transformations involves a precise knowledge of the processor resources usage, we believe they should be done at the compiler's back-end level (code generation). As we will explain here, our design choice is in favor of doing it in the context of dynamic compilation or static compilation with profiling feedback.

One of the promised virtues of dynamic compilation is exploiting run-time information to generate better code quality than static compilation. One of the most important source of performance is data locality and parallelism which can be exposed through loop transformations such as tiling, fusion, interchange, vectorization, etc. The effectiveness of those optimizations is highly dependent on the quality of the dependence information that is available. A static compiler is often limited by a too conservative alias and dependence analysis. This is especially true at code generation level where some information although available in the very first phases of the compiler have been dropped away (not maintained) by some transformations. However, some programing languages such as C are inherently already quite low level, and the task of (inter-procedural) alias/dependence analysis is very hard.

The way we plan to exploit run-time information is through the use of specialization: the compiler creates several versions of a the same loop body, some optimized one, and a fail-safe non-optimized one. Before entering the loop-nest, run-time tests are performed so as to choose the appropriate version, depending on the absence or presence of some aliasing/dependencies. The simplest test corresponds

to tracking that the base pointer of two different accesses correspond to two different memory allocations. This can be done by instrumenting the memory allocation manager. Unfortunately, not all "false" dependencies can be discarded this way. In particular there might be a dependence between two instructions, but with a distance (vector) higher than just a single iteration. Characterizing this distance (a function of the iteration vector) is obviously the role of the dependence analysis. But, algebraic techniques that are very powerful for this purpose require precise information on the memory access function, most of time not available as it should be. In this context, the role of run-time analysis is twofold. First dependences of the current execution instance should be characterized. This will be done through the combination of static analysis and an abstract monitoring of the memory accesses. Second, a possible transformation should be identified, the corresponding required properties characterized, and finally the associated run-time test generated. We will develop the problems related to the monitoring of memory accesses here.

The basic idea, is to build the single assignment form of the current execution instance. In its more expended form this corresponds to a data-flow acyclic tasks graph where each operation of the execution trace gives rise to a task node. A monitoring of memory accesses, caches for a given memory slot the identifier of the last operation that modified it. A write followed by a read leads to a flow dependence edge between the two corresponding nodes. Dependences that are affine functions of the iteration vector can easily be detected and the graph abstracted and compressed accordingly. The graph can be even more compressed by conservatively approximating dependences and by applying (on-line) transformations such as constant/copy propagation. However, even if building and compressing the corresponding flow graph of a given execution can be done, with reasonable efforts, in linear time, the multiplicative constant is too high for problems bigger than just a 100×100 matrix multiplication... The challenge is thus to be able to perform the same analysis using trace sampling: only the first, say ten, iterations along each loop dimension are instrumented and monitored. The none monitored memory accesses should be extrapolated and abstracted using some algebraic models.

Building and abstracting on-line the dependence graph of the current execution instance with a reasonable overhead thanks to monitoring sampling is an exciting challenge. We believe this to be a requirement for exploiting run-time information to expose more parallelism and locality. Obviously [1] the polytope model [66] will play an important role in our abstraction effort. However, as our goal is to handle as many class of code as possible (not just with affine dependences...), irregularities shall be represented thanks to the manipulation of graphs. Such graphs will be potentially very big, and linear time complexity algorithms are mandatory. As illustrated in the next item, the graph labelling technique advocated in this dissertation to encode reachability can be developed and applied to this context.

1.5.3 Over-approximating reachability in linear time

Graph labeling technique to encode basic-block reachability in the context of liveness analysis is a work under progress. In particular the hope is to find a more general class than structured control flow graphs, that could be accurately encoded using only two linear extensions. This technique can also be applied to the context of run-time analysis as it can be used for the encoding of data dependencies.

[1] This probably seems obvious mostly for French people...

The problem can be stated as follow: given a directed acyclic graph (DAG), the goal is to "compute" its transitive closure so as to be able to answer in constant time if a given node can reach another one. We identify two applications of this problem. (1) liveness: is a variable is dead or is there a future use? If not the resource to store it can be released. The original DAG is the forward control flow graph. (2) dependencies/data-flow: for two instructions/tasks a and b, is the value of a used in some other instruction/task a_1, itself used in another one a_2, and so on up to b? If yes, a should be executed before b, and if possible closed (in terms of both space and time) to b. The original DAG that is manipulated in this context is a dependence (resp. data-flow) graph. The challenge is to avoid having to pay the price of a quadratic complexity. To make this possible, over-approximation is allowed. The goal is obviously to be as accurate as possible.

Our approach consists in greedily building a few topological orders using depth-first-search (reverse post order) traversals or through "executions" of the corresponding petri-net. A nodes b is then considered to be reachable from another node b if a precedes b in each of the computed orders.

When the DAG is a forward control-flow graph of a program, we believe that loop-nesting forest should be used to bias the traversals. If the program has a maximum loop depth of one, building a realizer of cardinality three (i.e. encoding accurately the reachability using only three linear extensions) is quite straightforward: one just need to ensure that one of the linear extensions does order all natural loops after the remaining of the CFG. This allows to express the fact that once you enter in a natural loop [2] you cannot go back in the rest of the CFG using only forward edges. The opened problem is to see if one can generalize this approach for nested loops or if the dimension of the forward CFG grows with its maximum loop depth.

When the DAG is a dependence flow graph of a given execution instance the structure resembles more like a mesh. In that case, the traversal is biased toward some directions of the underlying (multi-dimensional) iteration space. However, whenever the order restricts to any affine relation, the use of graph labelling to encode is not necessarily relevant. An important opened problem is to see if mixed representations can be used involving graph labelling, algebraic relations, and simple enumerations.

[2] Only while loop constructions create such outgrowth in the forward control flow graph, do-until constructions do not impact its structure. See Section 2.7.1

2

Computing and Querying Liveness Information: Speed, Memory Footprint and Conservative Approximations

2.1 Introduction

Static single assignment (SSA) form is a popular program representation used by most modern compilers today. As its name suggests, SSA imposes a strict discipline where each variable must have textually (as opposed to (dynamic) SA used in automatic parallelization) a unique definition. Most of the time SSA implicitly enforces the dominance property where each (unique) definition dominates all its uses. Such SSA is also referred as strict SSA. Initially developed to facilitate the development of high-level program transformations, SSA form has gained much interest due to its favorable properties that often allow to simplify algorithms and reduce computational complexity. Today, SSA form is even adopted for the final code generation phase [89], i.e., the backend. Several industrial and academic compilers, static or just-in-time, use SSA in their backends, e.g., LLVM [92], Java HotSpot [87], LAO [61], LibFirm [91; 46], Mono [100]. Recent research on register allocation [27; 74; 108] even allows to retain SSA form until the very end of the code generation process.

This work investigates the use of strict SSA properties to simplify and accelerate *liveness analysis*, i.e., an analysis that determines for all variables the set of program points where the variables' values are eventually used by subsequent operations. Liveness information is essential to solve storage assignment problems, eliminate redundancies, and perform code motion. For instance, optimizations like software pipelining, trace scheduling, register-sensitive redundancy elimination, if-conversion, as well as register allocation heavily rely on liveness information. Our SSA destruction algorithm described in Chapter 3 and our tree-scan based coalescing developed in Chapter 4 require Liveness to devise interference information and take advantage of the techniques described here.

Traditionally, liveness information is obtained by data-flow analysis: liveness sets are computed for all basic blocks and variables in parallel by solving a set of data-flow equations [10]. These equations are usually solved by an iterative algorithm, propagating information backwards through the control-flow graph (CFG) until a fixed point is reached and the liveness sets stabilize. The number of iterations depends on the control-flow structure of the considered program, more precisely on the structure of its loops.

In this chapter, we show that, for strict SSA-form programs, the live-range of a variable, say v, have nice properties that can be expressed in terms of loop nesting forest of the control flow graph (CFG) and its corresponding directed acyclic graph the forward-CFG [116]. If, to make it simple we restrict to reducible CFG for now, roughly speaking those properties are:

- v is live at a program point q if and only if, v is live at the entry h of the largest loop that contains q but not the definition of v.
- v is live at h if and only if there is a path in the forward-CFG from h to a use of v that does not contain the definition.

A direct consequence of this property is, as a first contribution, the design of a data-flow algorithm that computes liveness sets *without the requirement of any iteration* to reach a fixed point. Instead, at most two passes over the CFG are necessary. The first pass, very similar to traditional data-flow analysis, computes partial liveness sets by traversing the forward-CFG backwards. The second pass refines the partial liveness sets and computes the final solution by propagating forward along the loop-nesting forest. For the sake of clarity, we first present our algorithm for reducible CFGs. Irreducible CFGs can be handled with a slight variation of the algorithm, with no need to modify the CFG itself (Section 2.4.3). Since our algorithm exploits advanced program properties some prerequisites have to be met by the input program and the compiler framework:

- $G = (V, E, r)$, the CFG of the input program is available.
- The program has to be in strict SSA form.
- A loop-nesting forest of the CFG is available. Also computable in $O(|V| \log^* |E|)$.

These assumptions are weak and easy to meet for clean-sheet designs. The (strict) SSA requirement is the main obstacle for compilers not already featuring it.

For SSA programs, another approach is possible that follows the classical definition of liveness: a variable is live at a program point q, if q belongs to a path of the CFG leading from a definition of that variable to one of its uses without passing through another definition of the same variable. Therefore, the live-range of a variable can be computed using a backward traversal starting on its uses and stopping when reaching its (unique) definition. For comparison, we designed optimized implementations of this path-exploration principle (see Section 2.5), for both SSA and non-SSA programs, and compared the efficiency of the resulting algorithms with our novel non-iterative data-flow algorithm.

Our experiments using the SPECINT 2000 benchmark suite in LAO demonstrate that the non-iterative data-flow algorithm outperforms the standard iterative data-flow algorithm by a factor of 2 on average. By construction, our algorithm is best suited for a set representation, such as bitsets, favoring operations on whole sets. In particular, for optimized programs, which have non-trivial live-ranges and a larger number of variables, our algorithm achieves a speed-up of 43% on average in comparison to the fastest alternative based on path exploration.

Another application to the properties of live-ranges under strict SSA-form is the design of an extremely simple liveness check algorithm. In contrast to classical data-flow analyzes liveness check [22] does not provide the set of variables live at a block, only its characteristic function. Liveness check provides a query system to answer questions such as "is variable v live at location q?" The results of the analysis remain valid during most program changes and, at the same time, allow for an *efficient* algorithm. Its main features are:

1. The algorithm itself consists of two parts, a *precomputation* part, and an *online* part executed at each liveness query. It is not based on setting up and subsequently solving data-flow equations.
2. The precomputation is *independent of variables,* it only depends on the structure of the control-flow graph. Hence, precomputed information *remains valid* upon adding or removing variables or their uses.
3. An actual query uses the def-use chain (see below) of the variable in question and determines the answer essentially by testing membership in precomputed sets.

It relies on the following prerequisites to be meet:

- $G = (V, E, r)$, the CFG of the input program is available.
- The dominance tree of the CFG is available. Otherwise it is computable in $O(|V|)$.
- A loop nesting forest of the CFG is available. Also computable in $O(|V| \log^* |E|)$.
- A list of uses for each variable, also known as def-use chain is available. Having an easy-to-maintain def-use chain is one of the major advantages of the SSA form. Hence, def-use chains are often available in SSA-based compilers. Updating the def-use chain when adding or removing uses of a variable incurs virtually no costs, quite contrary to updating liveness information on each change.

Again, as one can see, our assumptions are weak and easy to meet for clean-sheet designs and the (strict) SSA requirement is the main obstacle for compilers not already featuring it.

Before detailing our two-passes data-flow algorithm (Section 2.4) and the algorithms based on path-exploration (Section 2.5), we summarize in Section 2.2 different approaches for liveness analysis and provide in Section 2.3 some concepts that form the theoretical underpinning of our algorithm. Experiments are described in Section 4.6. We finish by revisiting the liveness check algorithm in Section 2.7 before concluding in Section 4.7.

2.2 Related Work

Liveness information is usually computed with iterative data-flow analysis, which goes back to Kildall [86]. The algorithms are, however, not specialized to the computation of liveness sets and may incur overhead. Several strategies are possible, leading to different worst-case complexities and performance in practice. Round-robin algorithms propagate information according to a fixed block ordering derived from a depth-first spanning tree and iterate until it stabilizes. The complexity of this scheme was analyzed by Kam et al. [84], see Section 2.4. Node Listing algorithms specifies, a priori, the overall sequence of nodes, where repetitions are allowed, along which data-flow equations are applied. Kennedy [85] devises for structured flow graphs node listings of size $2|V|$, with $|v|$ the number of control-flow-nodes, and mentions the existence of node listings of size $O|V|\log(|V|)$ for reducible flow graphs. Worklist algorithms focus on blocks that may need to be updated because the liveness sets of their successors (for backward problems) changed. Empirical results by Cooper et al. [51] indicate that the order in which basic blocks are processed is critical and directly impacts the number of iterations. They showed that, in practice, a mixed solution, called "single stack worklist", based on a worklist initialized with a round-robin order, is the most efficient one for liveness analysis. In

contrast, our non-iterative data-flow algorithm requires at most two passes over the basic blocks, in all cases. In practice, for strict SSA programs, it is on average twice as fast as the "single stack worklist" approach (see Section 4.6).

Alternative ways to solve data-flow problems belong to the family of elimination-based algorithms [125]. Through recursive reductions of the CFG, variables of the data-flow system are successively eliminated and equations are reduced until the CFG reduces to a single node. The best, but unpractical, worst case complexity elimination algorithm has an almost-linear complexity $O(|E|\alpha(|E|))$. It requires the CFG (resp. the reverse CFG) to be reducible for a forward (resp. backward) analysis. For non-reducible flow-graphs, non of the existing approaches can guarantee a worst case complexity better than $O(|E|^3)$. In practice, irreducible CFGs are rare, but liveness analysis is a backward data-flow problem, which frequently leads to irreducible reverse CFGs. In contrast, the handling of irreducibility by our algorithm is extremely simple and does not change its overall linear complexity (see Section 2.4.3).

Gerlek et al. [70] use so-called λ-operators to collect upward exposed uses at control-flow split points. Precisely, the λ-operators are placed at the iterated dominance frontiers, computed on the reverse CFG, of the set of uses of a variable. These λ-operators and the other uses of variables are chained together and liveness is efficiently computed on this graph representation. The technique of Gerlek et al. can be considered as a precursor of the live variable analysis based on the Static Single Information (SSI) form conjectured by Singer [128] and revisited by Boissinot et al. [20]. In both cases, insertion of pseudo-instructions guarantee that any definition is post-dominated by a use.

Another approach to compute liveness was proposed by Appel [10, p. 429]. Instead of computing the liveness information for all variables at the same time, variables are handled individually by exploring paths in the CFG starting from variable uses. Using logic programming, McAllester [97] presented an equivalent approach to show that liveness analysis can be performed in time proportional to the number of instructions and variables. However, his theoretical analysis is limited to a restricted input language with simple conditional branches and instructions. A more generalized analysis will be given later, both in terms of theoretical complexity (Section 2.5.4) and of practical evaluation (Section 4.6).

Liveness analysis for strict SSA was first addressed by Boissinot et al. [22] who introduced the liveness check approach. Each query relies in forward reachability checks from some nodes of the CFG. The framework developed in this paper for computing liveness sets follows the same idea of "path decomposition", but the use of loop-nesting forest allows to reformulate it in a more elegant and simpler way. Wimmer et al. [141] gave an algorithm, specialized to linear scan register allocation, to build the "intervals" of basic blocks where each variable is live. Although a possible extension to irreducible CFGs is sketched, the algorithm restricts itself to reducible CFGs or to a form of SSA where live-ranges are cut at loop-entry blocks with ϕ-functions. The algorithm we propose is a generalization [1] for computing liveness sets: it uses the concept of loop-nesting forest and is proved correct with no restriction on the CFG, on the strict SSA form, or on the loop-nesting forest as long as it respects the minimal properties stated by Ramalingam [116]. As a by-product, this proves the correctness of the algorithm of [141] and how a suitable order of basic blocks can be chosen thanks to a loop-nesting forest. Such orders were also ex-

[1] Actually, we designed this algorithm in 2009-2010 independently of [141] (see [18]).

ploited for liveness analysis in static single information (SSI) [20].

2.3 Foundations

This section introduces the notations used throughout this chapter and presents the necessary theoretical foundations. Readers familiar with flow graphs, loop-nesting forests, dominance, graph-labeling, and SSA form can skip ahead to Section 2.4.

2.3.1 Control Flow and Loop Structure

A *control-flow graph* $G = (V, E, r)$ is a directed graph, with nodes V, edges E, and a distinguished node $r \in V$ with no incoming edges. Usually, the CFG nodes represent the basic blocks of a procedure or function, every block is in turn associated with a list of operations or instructions.

Paths Let $G = (V, E, r)$ be a CFG. A *path* $\mathscr{P}$ of length k from a node u to a node v in G is a non-empty sequence of nodes $(v_0, v_1, \ldots, v_k)$ such that $u = v_0$, $v = v_k$, and $(v_{i-1}, v_i) \in E$ for $i \in [1..k]$. Implicitly, a single node forms a (trivial) path of length 0 and a self-loop forms a path of length 1. We assume that the CFG is connected, i.e., there exists a path from the root node r to every other node.

Dominance A node x in a CFG *dominates* another node y if every path from the root r to y contains x. The dominance is said to be strict if, in addition, $x \neq y$. A well-known property is that the transitive reduction of the dominance relation forms a tree, the *dominator tree*.

Loop-nesting forest Ramalingam [116] gave a recursive constructive definition of *minimal loop-nesting forests* as follows:

1. Partition the CFG into its strongly connected components (SCCs). Every non-trivial SCC, i.e., with at least one edge, is called a *loop*.
2. Within each non-trivial SCC, consider the set of nodes not dominated by any other node of the same SCC. Among these nodes, choose a non-empty subset and call it the set of *loop-headers*.
3. Remove all edges, inside the SCC, that lead to one of the loop-headers. Call these edges the *loop-edges*.
4. Repeat this partitioning recursively for every SCC after removing its loop-edges. The process stops when only trivial SCCs remain.

This decomposition can be represented by a forest, where each non-trivial SCC, i.e., every loop, is represented by an internal node. The children of a loop's node represent all inner loops (i.e., all non-trivial SCCs it contains) as well as the regular basic blocks of the loop's body. The forest can easily be turned into a tree by introducing an artificial root node, corresponding to the entire CFG. Its leaves are the nodes of the CFG, while internal nodes, labeled by loop-headers, correspond to loops. Note also that a loop-header cannot belong to any inner loop because all edges leading to it are removed before computing inner loops.

Reducible control-flow graphs A CFG is *reducible* if every loop has a single loop-header that dominates all nodes of the loop [77]. In other words, the only way to enter a loop is through its unique loop-header. Because of its structural properties, the class of reducible control-flow graphs is of special interest for compiler writers. Indeed, the vast majority of programs exhibit reducible CFGs. Also, as pointed out earlier, unlike other approaches that compute liveness information, we only need to discuss the reducibility of the original CFG, not of the reverse CFG.

Forward control-flow graph Let $\mathscr{L}$ be a loop-nesting forest for a given CFG G, then we define the corresponding forward control-flow graph $\mathscr{F}_{\mathscr{L}}(G)$ as the graph obtained by removing all loop-edges of the CFG. This directed graph is acyclic [116]. For reducible control-flow graphs, the loop-nesting forest being unique, there is no possible ambiguity.

Computing a loop-nesting forest The loop-nesting forest of a reducible CFG is unique and can be computed in $O(|V|\log^*(|E|))$. For example, Tarjan's algorithm [134] performs a bottom up traversal in a depth-first search tree of the CFG, identifying inner (nested) loops first. Because irreducible loops have more than one undominated node, the loop-nesting forest of an irreducible graph is not unique [116]. An interesting and simple-to-engineer loop-nesting forest algorithm is the one of Havlak [76], later improved by Ramalingam [115] to fix a complexity issue. Havlak's algorithm is a simple generalization of Tarjan's algorithm. It identifies a loop as a set of descendants of a back-edge target that can reach its source. In that case, the set of loop-headers is restricted to a single entry node, the target of a back-edge. Also, during the process of loop identification, whenever an entry node that is not the loop-header is encountered, the corresponding incoming edge (from a non-descendant node) is replaced by an edge to the loop-header.

2.3.2 Graphs, graph labeling

Reachability Let $F=(V,E)$ be a directed acyclic graph (DAG). The *transitive closure* of G is a new graph $F'=(V,E')$ where there is an edge (u,v) in E' if an only if there exists a non-null length path from u to v in F. We say that v is *reachable* from u in F if $(u,v)\in E'$. The transitive closure of a DAG can easily be computed in $O(|V||E|)$ using a reverse post-order traversal. To any DAG F, we can associate its partial order $\prec_G$ where $u\prec_G v$ if and only if $(u,v)\in E'$. A *realizer* of a partial order is a set of total orders which intersection gives rise to the partial order. The cardinality of a realizer is the number of total orders that compose it. The *dimension* of a partial order is the least cardinality of its realizers. Any finite total order of a DAG can be encoded using a injective mapping from its nodes to the set of integers: one node precedes another one if the number associated to the first is strictly lower than the number associated with the second. Any realizer of cardinality l of a DAG F can be encoded using l integers per node of F. The *labels* of two nodes, made up of those l integers for each, can be used to check if one can reach the other: one simply has to make the logical conjunction of l comparisons. This gives the basis of a more general technique, called graph labeling, used to encode, including reachability, diverse properties of graphs. We will see in this chapter that structured programs, lead to forward control-flow graph of "dimension" no more than two.

Least Common Ancestor Let $L=(V,E,r)$ be a tree with root r. The set of common ancestors of two nodes form a path from r to the so called least (or also nearest) common ancestor (LCA) of those two nodes. The static LCA query problem can be stated as follow: "given a rooted tree L how can L be pre-processed to answer any LCA queries quickly for any pair of nodes". From the simple remark that a prefix labeling (pre-computed in $O(|V|)$) allows for a binary tree to perform any query in $O(1)$, Harel and Tarjan [75] derived an algorithm for arbitrary trees. Another approach [14], that we will exploit further in this chapter, is based on the reduction [2] to the Range Minimum Query (RMQ) problem. Given an array A of numbers, the

[2] Any RMQ problem can actually also be reduced to an LCA problem

range minimum of two indices $0 \leq i \leq j < |A|$ is the index of the smallest element in $A[i..j]$ [3]. The reduction uses a depth first search (DFS) traversal of the tree to build an array, say A, of length $2|E|+1$, by pushing the tree-level (i.e. distance from the root) of a node whenever it visits it. Between the visit of node i and node j during this Euler tour, the node of lowest level is precisely the LCA of those two nodes. Pre-computing this array to find such a node in $O(1)$ can be done in $O(|A|)$.

2.3.3 Static Single Assignment Form

Static single assignment (SSA) form [54], is a popular program representation used in many compilers nowadays. In SSA form, each scalar variable is defined only once statically in the program text. To construct SSA form, variables having multiple definitions are replaced by several new *SSA-variables*, one for each definition. A problem appears when a use in the original program was reachable from multiple definitions. The new variables need to be disambiguated in order to preserve the program's semantic. The problem is solved by introducing ϕ-functions that are placed at control-flow joins. Depending on the actual execution flow, a ϕ-function defines a new SSA-variable by selecting the SSA-variable corresponding to the respective definition.

In this chapter, we require that the program under SSA form is *strict*. In a strict program, every path from the root r to a use of a variable contains the definition of this variable. Because there is only one (static) definition per variable, strictness is equivalent to the *dominance property*, which states that each use of a variable is dominated by its definition. This is true for all uses including a use in a ϕ-operation by considering that such a use actually takes place in the predecessor block from where it originates.

2.3.4 Liveness

Liveness is a property relating program points to sets of variables which are considered to be *live* at these program points. Intuitively, a variable is considered live at a given program point when its value is used in the future by any dynamic execution. Statically, liveness can be approximated by following paths, backwards, through the control-flow graph leading from uses of a given variable to its definitions - or in the case of SSA form to its unique definition. The variable is live at all program points along these paths. For a CFG node q, representing an instruction or a basic block, a variable v is *live-in* at q if there is a path, not containing the definition of v, from q to a node where v is used. It is *live-out* at q if it is live-in at some successor of q.

The computation of live-in and live-out sets at the entry and the exit of basic blocks is usually termed *liveness analysis*. It is indeed sufficient to consider only these sets since liveness within a basic block is trivial to recompute from its live-out set, either by traversing the block or by pre-computing which variables are defined or upward-exposed (see Section 2.4). *Live-ranges* are closely related to liveness. Instead of associating program points with sets of live variables, the live-range of a variable specifies the set of program points where that variable is live. Live-ranges in programs under strict SSA form exhibit certain useful properties, some of which have been exploited for register allocation [74; 27], some of which can be exploited during the computation of liveness information. However, the special behavior of

[3] when the smallest element appears multiple time, while this is not mentioned in [14], one can easily impose to pick up the last.

ϕ-operations often causes confusion on where exactly its operands are actually used and defined.

For a regular operation, variables are used and defined where the operation takes place. However, the semantics of ϕ-functions (and in particular the actual place of ϕ-uses) should be defined carefully, especially when dealing with SSA destruction. In all algorithms for SSA destruction, such as [34; 130; 17], a use in a ϕ-operation is considered live somewhere inside the corresponding predecessor block, but, depending on the algorithm and, in particular, the way copies are inserted, it may or may not be considered as live-out for that predecessor block. Similarly, the definition of a ϕ-operation is always considered to be at the beginning of the block, but, depending on the algorithm, it may or may not be marked as live-in for the block. To make the description of algorithms easier, we follow the definition by Sreedhar [130]. For a ϕ-function $a_0 = \phi(a_1, \ldots, a_n)$ in block B_0, where a_i comes from block B_i, then:

- a_0 is considered to be live-in for B_0, but, with respect to this ϕ-function, it is not live-out for B_i, $i > 0$.
- a_i, $i > 0$, is considered to be live-out of B_i, but, with respect to this ϕ-function, it is not live-in for B_0.

This corresponds to placing a copy of a_i to a_0 on each edge from B_i to B_0. The data-flow equations given hereafter and the presented algorithms follow the same semantics. They require minor modifications when other ϕ-semantics are desired. We will come back to these subtleties in Section 2.4.2.2.

2.3.5 Complexity of Liveness Algorithms

The running times of algorithms that compute liveness sets depend on several parameters. Some of them can only be evaluated by experiments, for example the locality in data structures, the cost of function calls instead of inlined operations, etc. This will be discussed in Section 4.6. However, some of them can be evaluated statically:

- How often are the program's instructions visited?
- How often are the CFG edges and nodes traversed?
- How many operations are performed on the algorithm's data structures and how costly are they?

Usually, liveness sets algorithms do not consider *local variables*, i.e., those defined in a block and used only there, as they are not part of live-in and live-out sets. The complexity of operations on variable sets is then measured in terms of $|W|$, where W is the set of non-local variables, called *global variables*. However, to identify local and global variables, to identify uses and definitions, all instructions of the program P need to be visited. Traversing its internal representation is costly and, moreover, is not necessarily linked to $|W|$ as it involves all variables. In other words, any liveness sets algorithm requires at least $|P|$ operations to read the program and, in practice, it is better to read it only once.

After possibly some precomputations in $O(|P|)$ operations, liveness sets algorithms work on the CFG $G = (V, E, r)$. The number of operations can then be evaluated in terms of $|V|$ and $|E|$, i.e., the number of times blocks and control-flow edges are visited. Hereafter, we assume $|V| - 1 \leq |E| \leq |V|^2$. The costs of these operations depend on the data structures used, both for intermediate results (e.g., uses of a variable or upward-exposed uses in a block) and for the final results, the live-in and live-out sets. For these sets, either lists (ordered or unordered) or bitsets can be used (we will not consider hash tables). The complexity has then to be discussed

according to the operations performed: test if an element is in a set, insertion in a set, union of two sets, sorting of a set. The best choice of the data structures may depend on the liveness sets algorithm used, but also on the algorithms that will use the live-in and live-out sets afterward. Such a complexity analysis will be done for each algorithm given hereafter.

Apart from its engineering advantages, Liveness check is, in terms of performance, a trade-off between the time spent on the precomputation phase and the time spent to answer queries. On one extremity we have a liveness sets algorithm that provides bitsets of live variables which allow each check at basic-block boundary to be performed in $O(1)$. On the other extremity we have no precomputation at all and each check corresponds to a backward traversal of the CFG from the set of uses to the query point or, if not live, to the definition point... In between we have the algorithm we describe here, which performs precomputations only based the CFG. Its complexity, discussed in Section 2.7, will involve the number of edges $|E|$ and the number of vertices $|V|$. Then the worst case complexity of each query will involve $|U|$ reachability tests, where U is the set of uses of the variables. But depending on how we encode the result of the precomputation, and in particular the transitive closure of the corresponding forward control-flow graph of the CFG, the performance of the queries will vary. If bitsets are used to store the set of reachable vertices, then the test can be done in $O(1)$. But we can choose to trade space with time and encode the reachability using graph labeling. In this last case, each test will require the number of "labels" per node used to encode the reachability, which is bounded by the dimension of the DAG. Many other "tricks" can be used. Some of them will be discussed along with their complexity in Section 2.7.

2.4 Data-Flow Approaches

A well-known and frequently used approach to compute the live-in and live-out sets of basic blocks is backward data-flow analysis [10]. The liveness sets are given by a set of *equations* that relate the *upward-exposed uses* and the *definitions* occurring within a basic block to the live-in and live-out sets of the predecessors and successors in the CFG. A use is said to be *upward-exposed* when a variable is used within a basic block and no definition of the same variable precedes the use locally within that basic block. The sets of upward-exposed uses and definitions do not change during liveness analysis and can thus be precomputed.

In the following equations, we denote by PhiDefs(B) the variables defined by ϕ-operations at entry of the block B and by PhiUses(B) the set of variables used in a ϕ-operation at entry of a block successor of the block B.

$$\begin{aligned} \text{LiveIn}(B) &= \text{PhiDefs}(B) \cup \text{UpwardExposed}(B) \cup (\text{LiveOut}(B) \setminus \text{Defs}(B)) \\ \text{LiveOut}(B) &= \bigcup_{S \in \text{succs}(B)} (\text{LiveIn}(S) \setminus \text{PhiDefs}(S)) \cup \text{PhiUses}(B) \end{aligned}$$

2.4.1 Complexity of Standard Data-Flow Approaches

The equations of the data-flow analysis can be solved efficiently using a simple iterative work-list algorithm that propagates liveness information among the basic blocks of the CFG. The liveness sets are refined on every iteration of the algorithm until a fixed point is reached, i.e., the algorithm stops when the sets cannot be refined any further. When the work-list contains edges of the CFG, the number of set

operations can be bounded by $O(|E||W|)$ [103], as each set can be modified (grow) at most $|W|$ times. As recalled in Section 2.2, the *round robin* algorithm [79; 84] allows another bound to be derived based on $d(G,T)$, the *loop connectedness* of the reverse CFG G, i.e., the maximal number of back edges (with respect to a depth-first spanning tree T) in a cycle-free path in G. The algorithm traverses the complete CFG on every iteration, at most $(d(G,T)+3)$ times, and thus results in $O(|E|(d(G,T)+3))$ set operations. These operations are mainly unions of sets, which can be performed in $O(|W|)$ for bitsets or ordered lists. The complexity is higher for unordered lists as the union is more costly, unless an intermediate sparse-set is used [37].

Depending on the structure of the program being analyzed, either of the two algorithms leads to a faster termination. In addition, both need a preliminary step to compute the upward-exposed uses and definitions of each basic block. This requires visiting every instruction of the program once, thus in time $O(|P|)$ where $|P|$ is the size of the program representation. Each operation consists in possibly inserting a global variable in a set, which is $O(1)$ for a bitset, $O(\log(|W|)$ for an ordered list, and $O(|W|)$ for an unordered list. For this last case, it is only $O(1)$ if a flag for each variable attests that the variable has not been already inserted, as it is for example done in Algorithm 10. Finally, assuming that the insertion is indeed $O(1)$, thus in particular for bitsets, the overall complexity is either $O(|P|+|E||W|^2)$ or $O(|P|+|E||W|(d(G,T)+3))$ depending on the update strategy. Our contribution in the rest of this section is the design, for strict SSA programs, of a liveness data-flow algorithm whose complexity is only $O(|P|+|E||W|)$, in other words, near-optimal as it includes the time to read the program, i.e., $O(|P|)$, and the time to propagate/generate the output, i.e., $O(|E||W|)$. We point out that it is also possible to design optimized algorithms based on path exploration, with the same near-optimal complexity $O(|P|+|E||W|)$, and operating at basic block level. This will be explained in Section 2.5.

2.4.2 Liveness Sets On Reducible Graphs

Instead of computing a fixed point, we show that liveness information can be derived in two passes over the control-flow graph. The first version of the algorithm requires the CFG to be reducible. We then show that arbitrary control-flow graphs can be handled elegantly and with no additional cost, except for a cheap pre-processing step on the loop-nesting forest.

The key properties of live-ranges under strict SSA form that we exploit for this purpose and that we will formalize and prove later on, can be outlined as follow:

1. Let q be a CFG node that does not contain the definition d of a variable v, h be the header of the maximal loop containing q but not d. Let h be q if such maximal loop does not exist. Then v is live-in at h if and only if there exists a forward path that goes from h to a use of v without going through the definition of v.

2. If v is live-in at the header of a loop then it is live at any node inside the loop.

Those two properties pave the way for describing the two steps that make up our liveness set algorithm:

1. A backward pass propagates partial liveness information upwards using a postorder traversal of the CFG.
2. The partial liveness sets are then refined by traversing the loop-nesting forest, propagating liveness from loop-headers down to all basic blocks within loops.

Algorithm 1 shows the necessary initialization and the high-level structure to compute liveness in two-passes.

```
Function Compute_LiveSets_SSA_Reducible(CFG)
begin
    for each basic block B do
        mark B as unprocessed;
    end
    DAG_DFS(R)                                   ▷ R is the CFG root node;
    for each root node L of the loop-nesting forest do
        LoopTree_DFS(L)
    end
end
```

Algorithm 1: Two-passes liveness analysis: reducible CFG

The postorder traversal is shown by Algorithm 2 which performs a simple depth-first search and associates every basic block of the CFG with partial liveness sets. The algorithm roughly corresponds to the precomputation step of the traditional iterative data-flow analysis. However, loop-edges are not considered during the traversal (Line 2). Recalling the definition of liveness for ϕ-operations, PhiUses(B) denotes the set of variables live-out of basic block B due to uses by ϕ-operations in B's successors. Similarly, PhiDefs(B) denotes the set of variables defined by a ϕ-operation in B.

```
Function DAG_DFS(block B)
begin
    for each S ∈ succs(B) if (B,S) is not a loop-edge do
        if S is unprocessed then DAG_DFS(S)
    end
    Live = PhiUses(B) for each S ∈ succs(B) if (B,S) is not a loop-edge do
        Live = Live ∪ (LiveIn(S) \ PhiDefs(S)) ;
    end
    LiveOut(B) = Live;
    for each program point p in B, backward do
        remove variables defined at p from Live;
        add uses at p to Live;
    end
    LiveIn(B) = Live ∪ PhiDefs(B) ;
    mark B as processed;
end
```

Algorithm 2: Partial liveness, with postorder traversal

The next phase, traversing the loop-nesting forest, is shown by Algorithm 3. The live-in and live-out sets of all basic blocks within a loop are unified with the liveness sets of its loop-header. This is sufficient in order to compute valid liveness information due to the fact that a variable whose live-range crosses a back-edge of the loop

is live-in and live-out at all basic blocks of the loop (see the proofs in Section 2.4.2.2).

```
Function LoopTree_DFS(node N of the loop-nesting forest)
begin
    if N is a loop node then
        Let B_N = Block(N);                          ▷ The loop-header of N
        Let LiveLoop = LiveIn(B_N) \ PhiDefs(B_N) ;
        for each M ∈ LoopTree_succs(N) do
            Let B_M = Block(M);                      ▷ Loop-header or block
            LiveIn(B_M) = LiveIn(B_M) ∪ LiveLoop;
            LiveOut(B_M) = LiveOut(B_M) ∪ LiveLoop;
            LoopTree_DFS(M);
        end
    end
end
```

Algorithm 3: Propagate live variables within loop bodies.

2.4.2.1 Complexity

In contrast to iterative data-flow algorithms, our algorithm has only two phases. The first traverses the CFG once, the second traverses the loop-nesting forest once. The number of operations performed during the CFG traversal of Algorithm 2 can be bounded by $O(|V|+|E|)$ unions of sets and $O(|P|)$ set insertions. Thus, assuming $|V|-1 \leq |E|$, the complexity of the first phase is $O(|E||W|+|P|)$ for bitsets. It is $O(|E||W|+|P|\log(|W|))$ if ordered lists are used instead.

The traversal of the loop-nesting forest follows a similar pattern. The size of the forest is at most twice the number of basic blocks $|V|$ in the CFG, because every loop node in the loop-nesting forest has one child node representing a basic block (a leaf in the forest). The loop body is executed exactly once for every node of the loop-nesting forest, which gives an upper bound for the number of set (union) operations for Algorithm 3 in $O(|V|)$. Since $|V|-1 \leq |E|$, this phase does not change the overall complexity mentioned above. The same is true for the unmark initialization phase. Our non-iterative data-flow algorithm has thus the expected near-optimal complexity $O(|P|+|E||W|)$, as claimed before. It avoids the multiplicative factor that bounds the number of iterations in standard iterative data-flow algorithms.

2.4.2.2 Correctness

The previous algorithms were specialized for the case where ϕ-functions are interpreted as copies at the CFG edges preceding the ϕ-functions. For the correctness proofs, we resort to the following, more generic, ϕ-semantics. A ϕ-function $a_0 = \phi(a_1,\ldots,a_n)$ at basic block B_0, receiving its arguments from blocks B_i, $i > 0$, is represented by a fresh variable a_ϕ, a copy $a_0 = a_\phi$ at B_0, and copies $a_\phi = a_i$ at B_i, for $i > 0$. Now, with respect to this ϕ-function, a_i, for $i > 0$, is not live-out at B_i and a_0 is not live-in at B_0 anymore. As for a_ϕ, since it is not a SSA variable, it is not covered by the following lemmas. But its live-range is easily identified: it is live-in at B_0 and live-out at B_i, $i > 0$, and nowhere else. Other ϕ-semantics extend the live-ranges of the ϕ-operands with parts of the live-range of a_ϕ and can thus be handled by locally refining the live-in and live-out sets. This explains why, in Algorithm 2, PhiUses(B) is added to LiveOut(B) (Line 2), PhiDefs(B) is added to LiveIn(B) (Line 2), and PhiDefs(S) is removed from LiveIn(S) (Line 2). This ensures

that the variable defined by a ϕ-function is marked as live-in and its uses as live-out at the predecessors. A similar adjustment appears on Line 3 of Algorithm 3.

The first pass propagates the liveness sets using a postorder traversal of the forward control-flow graph $\mathscr{F}_{\mathscr{L}}(G)$ of the CFG, obtained by removing all loop-edges. from the CFG. We first show that this pass correctly propagates liveness information to the loop-headers of the original CFG.

Lemma 2.1. *Let G be a reducible CFG, v a SSA variable, and d its definition. If L is a maximal loop not containing d, then v is live-in at the loop-header h of L iff there is a path in $\mathscr{F}_{\mathscr{L}}(G)$, not containing d, from h to a use of v.*

Proof. If v is live-in at h, there is a cycle-free path in the CFG from h to a use of v that does not go through d. Suppose this path contains a loop-edge (s,h') where h' is the header of a loop L', and $s \in L'$. Since the path has no cycle, $h' \neq h$ and thus $L \neq L'$. Now, two cases could occur:

- If $h \in L'$, L is contained in L'. As L is a maximal loop not containing d, $d \in L'$. Thus h' dominates d. This contradicts the fact that d strictly dominates all nodes where the variable v is live-in, in particular h'.
- If $h \notin L'$, then the path from h enters the loop L' a first time before going through the loop-edge (s,h'). Since the graph is reducible, the only way to enter L' is through h', thus there are two occurrences of h' in the path. Impossible since the path is cycle-free.

Thus, the path does not contain any loop-edges, which means that it is a valid path in $\mathscr{F}_{\mathscr{L}}(G)$. Conversely, if there exists a path in $\mathscr{F}_{\mathscr{L}}(G)$, then, of course, the variable v is live-in at h, since $\mathscr{F}_{\mathscr{L}}(G)$ is a sub-graph of the original graph G. □

Lemma 2.1 does not apply if there is no loop L satisfying the conditions. The following lemma covers this case.

Lemma 2.2. *Let G be a reducible CFG, v a SSA variable, and d its definition. Let p be a node of G such that all loops containing p also contain d. Then v is live-in at p iff there is a path in $\mathscr{F}_{\mathscr{L}}(G)$, from p to a use of v, not containing d.*

Proof. If v is live-in at p, there exists a cycle-free path in G from p to a use of v that does not contain d. Suppose this path contains a loop-edge (s,h) where h is the loop-header of a loop L, and $s \in L$:

- If $p \in L$ then $d \in L$ by hypothesis. Thus h dominates d, which is again, as in Lemma 2.1, impossible.
- If $p \notin L$, since s is in the loop, there has to be a previous occurrence of h on the path. Indeed, because the CFG is reducible, h is the only entry of L. This contradicts the fact that the path is cycle-free.

It follows that the path cannot contain any loop-edges. The path is thus a valid path in $\mathscr{F}_{\mathscr{L}}(G)$. Conversely, if there exists a path in $\mathscr{F}_{\mathscr{L}}(G)$, then v is live-in at p, since $\mathscr{F}_{\mathscr{L}}(G)$ is a sub-graph of the original graph G. □

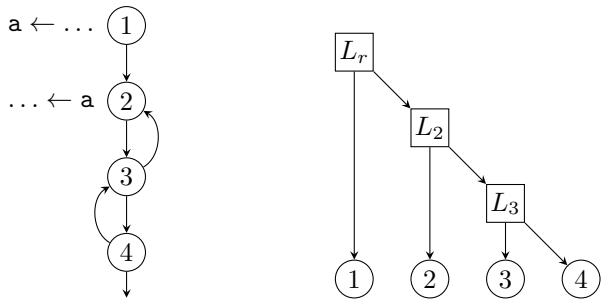
Algorithm 2, which propagates liveness information along the DAG $\mathscr{F}_{\mathscr{L}}(G)$, can only mark variables as live-in that are indeed live-in. Furthermore, if, after this propagation, a variable v is missing in the live-in set of a CFG node p, Lemma 2.2 shows that p belongs to a loop that does not contain the definition of v. Let L be such a maximal loop. According to Lemma 2.1, v is correctly marked as live-in at the header of L. The next lemma shows that the second pass of the algorithm (Algorithm 3) correctly adds variables to the live-in and live-out sets where they are missing.

Lemma 2.3. *Let G be a reducible CFG, L a loop, and v a SSA variable. If v is live-in at the loop-header of L, it is live-in and live-out at every CFG node in L.*

Proof. If v is live-in at h, the loop-header of L, then the definition d of v strictly dominates the CFG node h, thus $d \notin L$. Indeed, h cannot be dominated by any other node in L. Let p be a CFG node in L. Since L is strongly connected, there is a non-trivial path from p to h. It does not contain d as $d \notin L$. Since v is live-in at h, there is a path from h to a use of v that does not contain d. Concatenating these two paths proves that v is live-in at p. It is also live-out at p since p has a successor, where v is live-in, on the path from p to h. □

This lemma proves the correctness of the second pass, which propagates the liveness information within loops. Every CFG node, which is not yet associated with accurate liveness information, is properly updated by the second pass. Moreover, no variable is added where it should not be added.

Example. The CFG of Figure 2.1a is a pathological case for iterative data-flow analysis. The precomputation phase does not mark variable a as live throughout the two loops. An iteration is required for every loop-nesting level until the final solution is computed. In our algorithm, after the CFG traversal, the traversal of the loop-nesting forest (Figure 2.1b) propagates the missing liveness information from the loop-header of loop L_2 down to all blocks within the loop's body and all inner loops, i.e., blocks 3 and 4 of L_3. □



(a) Control-flow graph (b) Loop-nesting forest

Figure 2.1: Bad case for iterative data-flow analysis.

2.4.3 Liveness Sets on Irreducible Flow Graphs

The algorithms based on loops that we described above, are only valid for reducible graphs. We can derive an algorithm that works for irreducible graphs as well, in the following way: transform, the irreducible graph to a reducible graph, such that the liveness in both graphs is *equivalent*. First of all we would like to stress two points: (1) as opposed to well-known CFG transformations such as node splitting [81; 2] our transformation does not involve any size explosion. The key reason is that we do not impose the transformed graph to be *semantically equivalent* to the original one, only isomorphism of liveness is required. (2) In practice the graph is not actually modified, but Algorithm 2 can be changed to simulate the modification of some edges, on the fly.

The transformation can be done for any *minimal* loop-nesting forest as defined by Ramalingam but is simpler to explain and implement when each loop as a unique header, in particular with Havlak's loop-nesting forest. As we will see further, the transformation, in this case, simply relies in redirecting any edge (s,t) to the header of the outermost loop (if exists) that contain t but not s.

For loop-nesting forests with multiple headers per loop, similarly to Ramalingam [4] [116] we can add a dummy node to represent the headers. As it allows a simple proof by induction, we give here an iterative construction that transforms each irreducible loop into a reducible one. Performing the transformation this way, and especially from inner to outer loops would obviously lead, in the worst case, to a quadratic number of edge redirections. As mentioned earlier, in practice edges are virtually and directly redirected to the header (or representative) of the outermost loop.

For every loop L, $\mathit{EntryEdges}(L)$ denotes the set of entry-edges, i.e., the edges leading, from a basic block that is not part of the loop L, to a block within L. $\mathit{Entries}(L)$ denotes the set of L's entry-nodes, i.e., the nodes that are target of an entry-edge. Similarly, $\mathit{PreEntries}(L)$ denotes the set of blocks that are the source of an entry-edge. The set of loop-edges is given by $\mathit{LoopEdges}(L)$. Given a loop L from a graph $G = (V,E,r)$, we define the graph $\Psi_L(G) = (E',V',r)$ as follows. The graph is extended by a new node δ_L, which represents the (unique) loop-header of L after the transformation. All edges entering the loop from preentry-nodes are redirected to this new header. The loop-edges of L are similarly redirected to δ_L and additional edges are inserted leading from δ_L to L's former loop-headers. More formally:

$$\begin{aligned} E' \quad = \quad & E \setminus \mathit{LoopEdges}(L) \setminus \mathit{EntryEdges}(L) \cup \{(s,\delta_L) \mid s \in \mathit{PreEntries}(L)\} \\ & \cup \{(s,\delta_L) \mid \exists (s,h) \in \mathit{LoopEdges}(L)\} \cup \{(\delta_L,h) \mid h \in \mathit{LoopHeaders}(L)\} \end{aligned}$$

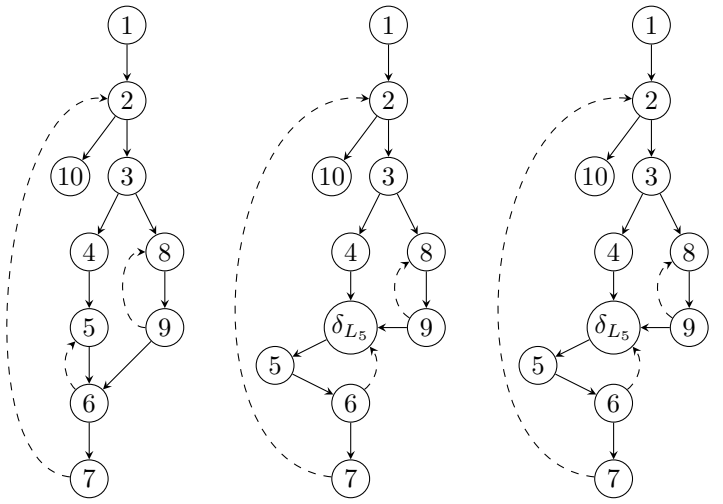
Repeatedly applying this transformation yields a reducible graph, slightly larger than the original graph, in which each node is still reachable from the root r. As already mentioned, depending on the order in which loops are considered, entry-edges may be updated several times during the processing in order to reach their final positions. But the loop-nesting forest structure remains the same. The next example illustrates this transformation.

To avoid building this transformed graph explicitly, an elegant alternative is to modify the CFG traversal (Algorithm 2). To make things simpler, we assume that the loop-nesting forest is built so that, as in Havlak's loop-nesting forest construction, each loop L has a single [5] loop-header, which can thus implicitly be fused with δ_L. It is then easy to see that, after all CFG transformations, an entry-edge (s,t) is redirected from s to $t.\mathsf{OLE}(s)$ the outermost loop containing t and excluding s, i.e., of the highest ancestor of t in the loop-nesting forest that is not an ancestor of s. Thus, whenever an entry-edge (s,t) is encountered during the traversal, we just have to visit $t.\mathsf{OLE}(s)$ instead of t, i.e., to visit the representative of the largest loop containing the edge target, but not its source. To perform this modification, we replace all occurrences of S by $S.\mathsf{OLE}(B)$ at Lines 2 and 2 of Algorithm 2, in order to handle irreducible flow graphs.

Example. Consider the CFG of Figure 2.2a and the loop-nesting forest in Figure 2.3.

[4] Our transformation differs from the one proposed by Ramalingam as we want to keep the liveness information here, not only the dominance.

[5] To handle loop-nesting forests with loops having several loop-headers, we can select one particular loop-header to be *the* loop representative (B_N in Algorithm 3). But then we need to "add" edges from this header to the other headers of the loop.



(a) Irreducible CFG G (b) Reducible CFG $\Psi_L(G)$; general case (c) Reducible CFG; single header case

Figure 2.2: A reducible CFG derived from an irreducible CFG, using the loop-nesting forest depicted in Figure 2.3. When loops are single header, such as for Havlak's loop-nesting forests, the transformation reduces to redirect some edges.

An artificial root node r for the entire CFG have been added to turn it into a tree. The loop L_5 containing the nodes 5 and 6 is irreducible as it has two entry-nodes, via the preentry-nodes 4 and 9. We suppose that node 5 was selected as its loop-header. The transformed reducible graph $\Psi_{L_5}(G)$ is depicted in Figure 2.2b. The graph might not reflect the semantics of the original program during execution, but it preserves the liveness properties of the original graph for a strict SSA program, as we will show in Theorem 2.5. As for Havlak's loop nesting forest, the original irreducible loop L_5 has a single loop-header 5 that can play the role of the loop-header of the transformed reducible loop (in place of δ_{L_5}): in Figure 2.2c edge $(9,6)$ is simply redirected to 5 the highest loop-nesting forest ancestor of 6 not ancestor of 9. □

2.4.4 Computing the outermost excluding loop

Finding the maximal loop not containing a node s but containing a node t is a problem similar to finding the least common ancestor (LCA) of the two nodes s and t in the rooted loop-nested forest: the loop in question is the only direct child of $\mathsf{LCA}(s,t)$, ancestor of t. As described in [14] and explained in Section 2.3, a LCA query can be reduced to a RMQ query that can itself be answered in $O(1)$, with a pre-computation of $O(n)$. Recall that the reduction to the RMQ problem is based on the Euler tour of the tree, that is the sequence of nodes as they are visited by a depth first search (DFS). On the example of Figure 2.3 such a tour would create the array $A = [\langle L_R,0\rangle, \langle 1,1\rangle, \ldots, \langle \mathbf{2},2\rangle, \langle L_2,1\rangle, \langle 3,2\rangle, \langle L_2,1\rangle, \ldots, \langle \mathbf{9},3\rangle, \langle L_8,2\rangle, \langle L_2,1\rangle, \langle L_5,2\rangle, \langle 5,3\rangle, \langle L_5,2\rangle, \langle \mathbf{6},3\rangle, \ldots]$ that reports between chevrons both the id of the visited node and its level, i.e. its distance from L_R, in the tree. Between the (unique) occurrences of the leafs 9 and 6 the occurrence of minimum level in A is $\langle L_2,1\rangle$, found in $O(1)$,

gives L_2 as the LCA of 9 and 6. On this example, the outermost loop containing 6 but excluding 9 is L_5, given by the occurrence $\langle L_5, 2\rangle$ just following the occurrence of the LCA. The attentive reader would have noticed that another order used for the Euler tour could show $\langle L_8, 2\rangle$ as the immediate following occurrence of $\langle L_5, 2\rangle$. Also, there might exists more than one occurrence of the same node in a sub-array of A: suppose for example that we are looking for the outermost loop containing 9 but excluding 2, then L_2, the LCA of 9 and 2 is visited multiple times in between...

To make sure that the loop-node we are looking for, i.e. the outermost loop containing t and excluding s, occurs in A just after the queried $\mathsf{RMQ}(s,t)$, we do the following:

1. take a topological order of the forward CFG, and visit the children of each node along this order during the DFS of the Euler tour.

2. restrict to queries such that s is before t in the topological order used for the Euler tour. Notice that this condition will always be fulfilled if s dominates t or if (s,t) is a forward edge of the CFG.

3. parametrize the RMQ algorithm to pick up the *last* occurrence of minimum level.

This gives a sophisticated but $\langle O(n), O(1)\rangle$ solution the OLE problem, that we will not develop further as in practice maximum loop-depth is usually small. Simpler solutions can be used such as the naive one that would just walk upward in the tree starting at both nodes and stops when it encounters a common ancestor.

The other alternative method (shown in Algorithm 5) is to pre-compute the set of ancestors from the loop-tree for every node. Then a simple set operation can find the node we are looking for: the ancestor of the definition node are removed from the ancestor of the query point. From the remaining ancestors, we pick the shallowest. Using bitsets, indexed with a topological order of the loop tree, this operations are easily implemented. The removal is a `bit-inversion` followed by a `bitwise-and` operation, and the shallowest node is found by searching for the `first-bit-set` in the bitset. Since the number of loops (and thus the number loop-headers) is rather small, the bitsets are themselves small as well and this optimization does not result in much wasted space.

Consider a topological indexing of loop-headers accessible using: $n.LTindex$ (n being a loop-header) or reciprocally $i.node$ (i being an index). For each node, we need a bitset (indexed by loop-headers) of all its ancestors in the loop tree: $n.ancestors$. This can be computed using any topological traversal of the loop-tree by a call of DFS_COMPUTE_ANCESTORS(L_r). Notice that some compiler intermediate representation sometimes consider L_r as a loop header. Considering so in DFS_COMPUTE_ANCESTORS will not spoil the behavior of OLE.

Using this information, finding the outermost excluding loop can be done by simple bitset operations:

Example. Consider the example of Figure 2.3 again and suppose the loops L_2, L_8, and L_5 are respectively indexed 0,1, and 2. Using big-endian notations for bitsets Algorithm 4 would give the labels to nodes 9 and 6, 110 and 101 respectively. The outermost loop containing 6 but not 9 is given by the leading bit of $101 \wedge \neg 110 = 001$ i.e. L_5.

```
Function DFS_compute_ancestors(node n)
begin
    if n ≠ L_r then
        n.ancestors ← n.LTparent.ancestors;
    else
        n.ancestors ← bitset_empty;
    end
    if n.isLoopHeader then
        n.ancestors.add(n.LTindex)
    end
    for s in n.LTchildren do
        DFS_compute_ancestors(s)
    end
end
```

Algorithm 4: Compute the loop-nesting forest ancestors

```
Function OLE(node self, node b)
begin
    nCA ← bitset_and(self.ancestors, bitset_not(b.ancestors));
    if nCA.isempty then
        return self
    else
        return nCA.bitset_leading_set.node;
    end
end
```

Algorithm 5: Outermost excluding loop

2.4.4.1 Complexity

The changes to the original forest algorithm are minimal and only involve the invocation of OLE to compute the outermost excluding loop. The other operations are not impacted and complexity results obtained previously still hold as the number of edges $|E|$ does not change. The OLE invocation that solely depends on the structure of the loop-nesting forest, involves a pre-computation part and the queries themselves. The RMQ based solution has a complexity of $\langle O(|L|), O(1)\rangle$, while algorithms 4 and 5 has a complexity of $\left\langle O\left(|V|\left\lceil\frac{|L|}{B}\right\rceil\right), O\left(\frac{|L|}{B}\right)\right\rangle$, with L the amount of loop headers, B the size of a bitset. Assuming the use of the RMQ based solution, modifying the algorithm with OLE to handle irreducible CFGs does not change the overall complexity.

2.4.4.2 Correctness

We now prove that, for strict SSA programs, the liveness of the resulting reducible CFG is equivalent to the liveness of the original CFG. The following results hold even for a loop-nesting forest whose loops can have more than one loop-header. First, to be able to apply the lemmas and algorithms of Section 2.4.2 to the reducible CFG $\Psi_L(G)$, we need to prove that any definition of a variable still dominates its uses.

Lemma 2.4. *If d dominates u in G, then d dominates u in $\Psi_L(G)$.*

Proof. Suppose that d does not dominate u in $\Psi_L(G)$: there is in $\Psi_L(G)$ a cycle-free path $\mathscr{P}$ from the root node r to u such that $d \notin \mathscr{P}$. Since d dominates u in G, the

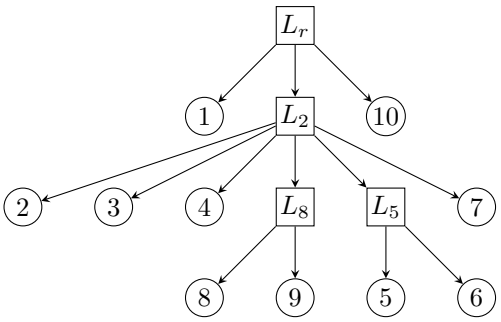


Figure 2.3: A rooted loop-nested forest for the CFG of Figure 2.2.

path $\mathscr{P}$ contains edges that do not belong to G, in particular, it enters the loop L at the unique loop-header δ_L from a preentry-node s of L. In G, this edge corresponds to an entry-edge from s to an entry-node t of L. As $\mathscr{P}$ has no cycle, it goes through δ_L only once, thus the only edges of $\mathscr{P}$ that do not belong to G are (s,δ_L) and (δ_L,h) for some loop-header h of L. As L is strongly connected, there is a path in G from t to h whose nodes are all in L. Concatenating the subpath of $\mathscr{P}$ from r to t, the path from t to h, and the subpath of $\mathscr{P}$ from h to u defines a path in G. Since d dominates u, d belongs to the subpath from t to h, thus $d \in L$. By definition of a loop-nesting forest, the loop-header h cannot be dominated by d. Thus, as $d \neq h$, there is a path in G from r to h that does not contain d. Increasing this path with the subpath from h to u contradicts the fact that d dominates u. □

It remains to show that, for every basic block present in both graphs, the live-in and live-out sets are the same. This is proved by the following theorem.

Theorem 2.5. *Let v be a SSA variable, G a CFG, transformed into $\Psi_L(G)$ when considering a loop L of a loop-nesting forest of G. Then, for each node q of G, v is live-in (resp. live-out) at q in G iff v is live-in (resp. live-out) at q in $\Psi_L(G)$.*

Proof. If v is live-in (resp. live-out) at q in G, there is a path $\mathscr{P}$ in G, from q to u that does not contain its definition d (except possibly $d = q$ if v is live-out). As d dominates u, it also dominates any node of this path. Two cases can occur:

- If $d \in L$, then $\mathscr{P}$ does not contain any loop-edge or entry-edge of L because the target of such an edge, by definition of a loop-nesting forest, is not dominated by any other node in L, in particular d. Thus, the path $\mathscr{P}$ from q to u exists in $\Psi_L(G)$ with no modification.
- If $d \notin L$, $\mathscr{P}$ can be modified into a path in $\Psi_L(G)$ as follows. If $\mathscr{P}$ contains a loop-edge (s,h) of L, we replace it by the two edges (s,δ_L) and (δ_L,h). Now consider an entry-edge (s,t) of L in $\mathscr{P}$. As L is strongly connected, for at least one loop-header h of L, there is a path $\mathscr{P}'$ in G from h to t, with no loop-edge, thus also a path in $\Phi_L(G)$. We then replace the edge (s,t) by the two edges (s,δ_L) and (δ_L,h), followed by the path $\mathscr{P}'$, which is fully contained in L, so does not contain d. Thus v is live-in and live-out at q in $\Psi_L(G)$.

Conversely, consider a cycle-free path $\mathscr{P}$ in $\Psi_L(G)$ from q to u, that does not contain d, except possibly $d = q$. According to Lemma 2.4, d dominates u in the graph $\Psi_L(G)$ too, thus all nodes in $\mathscr{P}$.

- If $d \in L$, then $\mathscr{P}$ does not contain δ_L, because d dominates any node in $\mathscr{P}$ and δ_L is not dominated by any node in L (also $\delta_L \neq d$ because δ_L is an empty node, but not d). Hence $\mathscr{P}$ is also a valid path in G.
- If $d \notin L$ and if $\mathscr{P}$ does not contain δ_L, then $\mathscr{P}$ is a valid path in G. Otherwise, the only edges in $\mathscr{P}$ with no direct correspondence in G are the two edges (s, δ_L) and (δ_L, h) where, with respect to the loop-forest of G, the edge (s,t) is an entry-edge of L and h a loop-header of L. As L is strongly connected, there is a path $\mathscr{P}'$ in G, from t to h, fully contained in L, thus not containing d. The edges (s, δ_L) and (δ_L, h) can then be replaced by the edge (s,t) followed by the path $\mathscr{P}'$.

The liveness sets are thus the same in both CFGs. □

2.5 Liveness Sets using Path Exploration

Another maybe more intuitive way of calculating liveness sets is closely related to the definition of the live-range of a given variable. As recalled earlier, a variable is live at a program point p, if p belongs to a path of the CFG leading from a definition of that variable to one of its uses without passing through the definition. Therefore, the live-range of a variable can be computed using a backward traversal starting at its uses and stopping when reaching its (unique) definition. This idea was first proposed by Appel in his "Tiger" book [10] (Pages 208 and 429). We distinguish two implementation variants of the basic idea.

2.5.1 Processing Variables by Use

The first variant relies solely on the CFG of the input program and does not require any additional preprocessing step. Starting from a use of a variable, all paths where that variable is live are followed by traversing the CFG backwards until the variable's definition is reached. Along the encountered paths, the variable is added to the live-in and live-out sets of the respective basic blocks.

Algorithm 6 performs the initial traversal discovering the uses of all variables in the program. Every use is the starting point for a path exploration performed by Algorithm 7. The presented algorithm has also some similarities with the liveness algorithm used by the open-source compiler infrastructure LLVM.

```
Function Compute_LiveSets_SSA_ByUse(CFG)
begin
    for each basic block B in CFG do            ▷ Consider all blocks successively
        for each v ∈ PhiUses(B) do              ▷ Used in the φ of a successor block
            LiveOut(B) = LiveOut(B) ∪ {v};
            Up_and_Mark(B, v);
        end
        for each v used in B (φ excluded) do    ▷ Traverse B to find all uses
            Up_and_Mark(B, v);
        end
    end
end
```

Algorithm 6: Compute liveness sets by exploring paths from variable uses.

```
Function Up_and_Mark(B, v)
begin
    if def(v) ∈ B (φ excluded) then
        return                                   ▷ Killed in the block, stop
    end
    if v ∈ LiveIn(B) then
        return                                   ▷ Propagation already done, stop
    end
    LiveIn(B) = LiveIn(B) ∪ {v};
    if v ∈ PhiDefs(B) then
        return                                   ▷ Do not propagate φ definitions
    end
    for each P ∈ CFG_preds(B) do                 ▷ Propagate backward
        LiveOut(P) = LiveOut(P) ∪ {v};
        Up_and_Mark(P, v);
    end
end
```

Algorithm 7: Explore all paths from a variable's use to its definition.

2.5.2 Processing Variables by Definition

The second variant follows the initial idea of Appel [10, p. 429], but adapted and optimized to work on blocks instead of instructions. Depending on the particular compiler framework, a preprocessing step that performs a full traversal of the program (i.e., the instructions) might be required in order to derive the def-use chains for all variables, i.e., a list of all uses for each SSA-variable. Algorithm 8 adapts the pseudo-code shown previously to make use of these def-use chains. The algorithm to perform the path exploration stays the same, i.e., Algorithm 7.

```
Function Compute_LiveSets_SSA_ByVar(CFG)
begin
    for each variable v do
        for each block B where v is used do
            if v ∈ PhiUses(B) then           ▷ Used in the φ of a successor block
                LiveOut(B) = LiveOut(B) ∪ {v};
            end
            Up_and_Mark(B, v);
        end
    end
end
```

Algorithm 8: Compute liveness sets per variable using def-use chains.

A nice property of this approach is that the processing of different variables is not intermixed, i.e., the processing of one variable is completed before the processing of another variable begins. This enables to optimize the Up_and_Mark phase by using a stack-like set representation. Unlike in Algorithm 7, the expensive set-insertion operations and set-membership tests can then be avoided. It is indeed sufficient to test the top element of the stack, see Algorithm 9. Note also that, in strict SSA, in a given block, no use can appear before a definition. Thus, if v is live-out or used in a block B, it is live-in iff it is not defined in B.

```
Function Up_and_Mark_Stack(B, v)
begin
    if def(v) ∈ B (φ excluded) then
        return                                   ▷ Killed in the block, stop
    end
    if top(LiveIn(B)) = v then
        return                                   ▷ propagation already done, stop
    end
    push(LiveIn(B), v);
    if v ∈ PhiDefs(B) then
        return                                   ▷ Do not propagate φ definitions
    end
    for each P ∈ CFG_preds(B) do                 ▷ Propagate backward
        if top(LiveOut(P)) ≠ v then
            push(LiveOut(P), v)
        end
        Up_and_Mark_Stack(P, v);
    end
end
```

Algorithm 9: Optimized path exploration using a stack-like data structure.

2.5.3 Path Exploration for non-SSA-form Programs

Interestingly, we can show that, with an additional preprocessing step, the path exploration approach can also be applied to programs that are *not* in SSA form. Similar to the precomputation of the def-use chains for the variable-by-variable approach (Section 2.5.2), we can avoid multiple traversals of the internal program representation by precomputing information on uses and definitions of all variables in the program. First, using a forward scan of each block (see Algorithm 10), we compute, for each variable v, the list of blocks, denoted by UpwardExposed(v), where v is live-in and upward-exposed, i.e., the blocks where the first access to v is a use and not a definition. We also compute the list of blocks, denoted by Defs(v), where the variable is defined.

```
Function Compute_Killing_and_UpwardExposed_Stack(CFG)
begin
    for each basic block B in the CFG do
        for each access to a variable v, from start to end of block do
            if top(Defs(v)) ≠ B then                 ▷ No definition yet
                if v is a use then                   ▷ Upward-exposed use at B
                    if top(UpwardExposed(v)) ≠ B then
                        push(UpwardExposed(v), B);
                    else
                        push(Defs(v), B)             ▷ First definition in B
                    end
                end
            end
        end
    end
end
```

Algorithm 10: Compute the upward-exposed uses and definitions of variables.

The algorithm to compute the liveness information is similar to the optimized variable-by-variable algorithm presented in the previous section. The main difference is that multiple definitions of the same variable might appear in the program. In order to avoid expensive checks to find definitions during the path exploration, basic blocks are marked with a variable during the processing. The marking indicates that the path exploration algorithm should stop following the current path any further. Also, when the variable is already known to be live-in, the path exploration stops. Algorithms 11 and 12 show the modified pseudo-code of the liveness algorithm for programs that are not in SSA form.

```
Function Compute_LiveSets_NonSSA_ByVar_Stack(CFG)
begin
    for each basic block B of CFG do
        mark B with ⊥
    end
    for each variable v do
        for each block B in Defs(v) do
            mark B with v
        end
        for each block B in UpwardExposed(v) do
            if top(LiveIn(B)) ≠ v then                    ▷ Not propagated yet
                push(LiveIn(B), v)                        ▷ Insert in live-in set
                for P ∈ CFG_preds(B) do                   ▷ Propagate backward
                    Up_and_Mark_NonSSA_Stack(P, v);
                end
            end
        end
    end
end
```

Algorithm 11: Compute liveness per variable for non-SSA-form programs.

```
Function Up_and_Mark_NonSSA_Stack(B, v)
begin
    if top(LiveOut(P)) ≠ v then push(LiveOut(P), v);
    if B is marked with v then return;                    ▷ Killed in the block, stop
    if top(LiveIn(B)) = v then return;                    ▷ Already processed
    push(LiveIn(B), v);                                   ▷ Not propagated yet
    for each P ∈ CFG_preds(B) do                          ▷ Propagate backward
        Up_and_NonSSA_Mark_Stack(P, v);
    end
end
```

Algorithm 12: Compute liveness sets per variable for non-SSA-form programs.

2.5.4 Complexity

All path-based approaches yield essentially the same complexity results, if they are optimized, as we propose, to traverse the internal program representation only once. The outermost loops of Algorithm 6 and the def-use chain precomputation for Algorithm 8 visit every instruction once per variable in order to start a path traversal, which results in an $O(|P|)$ bound ($|P|$ is the size of the program). The depth-first traversal of the CFG similarly visits every edge of the graph once per variable, thus the number of set insertions, respectively stack operations, performed by the loop

of Algorithm 7 and 9 is limited by $O(|E||W|)$ ($|E|$ is the number of CFG edges, $|W|$ is the number of global variables). The insertions outside of the loop are performed only once per basic block per variable, and thus do not appear in the final bound as we assumed $|V|-1 \leq |E|$ ($|V|$ is the number of CFG basic-blocks). The overall complexity is therefore $O(|P|+|E||W|)$, assuming unit time set insertions.

The algorithm for programs not in SSA form shows a similar structure and thus also behaves similarly. However, we need to account for the precomputation of the upward-exposed uses and variable definitions for every block in the program – see Algorithm 10. The algorithm visits every instruction once per variable, which does not change the bound stated above. The algorithm also incurs some initialization overhead due to the marking of basic blocks. The first for-loop is executed once for every basic block, while the second loop at Line 5 of Algorithm 11 gives $O(|V||W|)$. Again, assuming a connected CFG, this leaves the bound unchanged. All path-based algorithms thus share the same complexity bound $O(|P|+|E||W|)$, as our non-iterative data-flow algorithm.

This bound is in line with the $O(|N||W|)$ bound obtained – for a simplified model – by the bottom-up logic approach of McAllester [97], where $|N|$ is the number of instructions. McAllester's algorithm (as the approach of Appel based on path exploration [10, p. 429]) works at the granularity of instructions and not of basic blocks. It is assumed that branching instructions have at most two successors, i.e., $|E| \leq 2|N|$, and that each instruction has at most two uses and one definition, thus $|P|$ (the program size) is in the order of $|N|$. Therefore, with McAllester's simplifying assumptions, $O(|P|+|E||W|) = O(|N||W|)$. But this result is not directly applicable for general program representations appearing in actual compilers. A direct generalization – e.g., expressing the constraints at the granularity of instructions, with no preprocessing, and following the algorithm for the satisfiability of Horn formulae as exposed by Minoux [99] – would lead to sub-optimal complexity bounds. In particular, it is important to avoid traversing the program multiple times to get $O(|P|)$ and not $O(|P||W|)$, or, even worse, a complexity that depends on the total number of variables, and not just global variables. The optimized algorithms we just proposed in this section, based on path exploration, achieve this goal: they operate at basic block level with complexity $O(|P|+|E||W|)$.

2.6 Experiments

As previously shown, the theoretical complexity of the three liveness algorithms we propose (use-by-use, variable-by-variable, or loop-forest-based) is the same and it is near-optimal: it includes the time to read the program, i.e., $O(|P|)$, and the time to propagate/generate the output, i.e., $O(|E||W|)$. Furthermore, variables are added to sets only exactly when needed. The algorithms differ by the order in which variables and blocks (i.e., the CFG) are processed. The first path-exploration variant, called use-by-use, traverses the program backwards and, for every encountered variable use, starts a depth-first search to find the variable's definition. The variable is added to the live-in and live-out sets along the discovered paths. The other variant, called variable-by-variable, processes one variable after the other and relies on precomputed def-use chains to find the variable's uses. The loop-forest-based algorithm also traverses the program and the CFG at the same time, as the use-by-use variant, but it treats all variables that are live in a block together. These differences induce important variations in terms of runtime, which are not visible in the theoretical

analysis. Also, the big O notation hides some constants. The goal of this section is to discuss the performances in practice, depending on the program characteristics being analyzed and the data structures used.

The algorithms were implemented using the production compiler for the STMicroelectronics ST200 VLIW family, which is based on GCC as front-end, the Open64 optimizers, and the LAO code generator [61]. We computed liveness relatively late during the final code generation phase of the LAO low-level optimizer, shortly before prepass scheduling. In addition, all algorithms were implemented and optimized for two different liveness set representations. We evaluated the impact of pointer-sets, i.e., ordered lists, which promise reduced memory consumption at the expense of rather costly set operations. In addition, plain bitsets were evaluated, which offer faster accesses, but are often considered to be less efficient in terms of memory consumption and are expected to degrade in performance as the number of variables increases, due to more cache misses and memory transfers. In the following, all measurements are relative to the iterative data-flow approach, which performed the worst in all our experiments.

We applied the various algorithms proposed in this work to the C programs of the SPECINT 2000 benchmark suite to measure the time required to compute all liveness sets, i.e., for all basic blocks in the program, the live-in and live-out sets for all global variables. To obtain reproducible results, the execution time is measured using the instrumentation and profiling tool *callgrind*, which is part of the well-known *valgrind* tool. The measurements include the number of dynamic instructions executed as well as memory accesses via the instruction- and data caches. Using these measurements, a cycle estimate is computed for the liveness computation only, which minimizes the impact, on the measurements, of other compiler components and other programs running on the host machine.

The number of global variables, i.e., variables crossing basic block boundaries, depends largely on the compiler optimizations performed before the liveness calculation. Programs that are not optimized usually yield very few global variables since most values are kept in memory locations by default. However, optimized programs usually yield longer and more branched live-ranges. We thus investigate the behavior for optimized and unoptimized programs using the compiler flags `-O2` and `-O0` respectively. Table 2.1 shows the number of global variables, basic blocks, and operations for the optimized benchmark programs. The statistics for unoptimized programs are not shown, since the number of global variables never exceeds 19.

2.6.1 Pointer-Sets

The pointer-sets in LAO are implemented as arrays ordered by decreasing numeric identifiers. This results in rather fast set operations such as union and intersection at the expense of a rather expensive insertion. Due to the ordering of the pointer-sets, insertions are the fastest when the inserted variable is known to have an index number larger than all other variables in the set. The implementation of the variable-by-variable algorithm thus performs best with the optimization for stack-like data structures presented in Section 2.5.2. Variables are considered in the right order to replace a logarithmic search by an insertion as first element. This is not the case for the use-by-use variant (Section 2.5.1). Note that the ordering of the set is preserved throughout the computation.

Figures 2.4 and 2.5 compare the average execution times measured for the individual benchmarks in comparison to the our new non-iterative data-flow algo-

	# Variables			# Blocks			# Operations		
Benchmark	min	avg	max	min	avg	max	min	avg	max
164.gzip	11	104	586	2	32	212	22	226	1312
175.vpr	10	84	573	2	33	492	21	224	1734
176.gcc	10	119	36063	2	37	1333	11	282	41924
181.mcf	12	52	118	2	18	52	24	135	439
186.crafty	11	147	1048	2	67	2112	22	547	9836
197.parser	10	58	1076	2	21	343	21	126	1942
253.perlbmk	10	61	1947	2	28	731	16	180	4876
254.gap	10	95	6472	2	31	778	13	244	9169
255.vortex	10	51	645	2	26	667	21	166	3361
256.bzip2	10	73	972	2	22	282	21	163	1931
300.twolf	10	186	3659	2	53	715	12	458	8691

Table 2.1: Program characteristics for optimized programs.

rithm, for both unoptimized and optimized programs. The results indicate that the variable-by-variable algorithm outperforms the loop-forest-based approach (Section 2.4) by 74% and 64% for optimized and unoptimized programs respectively. Indeed, the latter traverses all blocks for all variables and is better adapted for set operations, i.e., a bitset data structure. The results for the use-by-use algorithm highly depend on the characteristics of the input program. In particular for larger optimized programs such as `gcc`, `perlbmk`, and `twolf` the use-by-use approach shows poor results. This can be attributed to the unordered processing of the variables, resulting in costly insertion operations. Therefore, our tricks to design a stack-based implementation (Algorithm 9), at block level, of liveness analysis based on path-exploration, are worthwhile for ordered pointer-sets. The non-iterative data-flow analysis mainly applies set unification, which can be performed fast on the ordered set representation, and thus gains in comparison to the use-by-use variant.

2.6.2 Bitsets

The use of bitsets in data-flow analysis is very common since most of the required operations, such as insertion, union, and intersection, can be performed efficiently using this representation. In fact, our measurements show that these operations are so fast that the allocation and initialization of the bitsets becomes a major factor in the overall execution time of the considered algorithms.

Actually, cconsidering the program characteristics from Table 2.1, using sparse pointer-sets does not appear to be a good choice to represent liveness sets, and our experiments indicate that bitsets are overall superior. Indeed, the average number of variables per function is relatively low and does not exceed 184 for our benchmark set. In fact, 97% out of the 5848 functions contain less then 320 variables and almost 99% less then 640, which yields a size of merely 20 words on 32-bit machines in order to represent all variables as bitsets for almost all functions considered. It is thus not surprising that the baseline iterative data-flow algorithm using bitsets outperforms the same algorithm using pointer-sets by 69% and 85% for optimized and unoptimized input programs, and is thus even faster than the var-by-var approach on pointer-sets. The same is true for the three other algorithms studied in this chap-

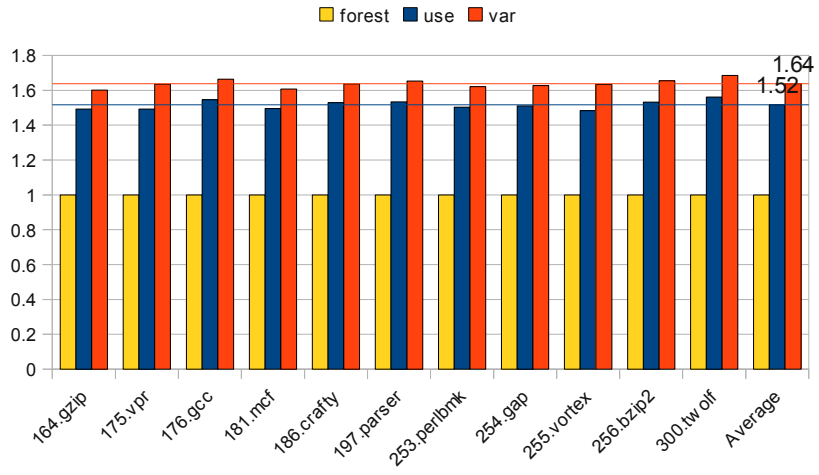


Figure 2.4: Speed-up with regard to our loop-forest-based approach using pointer-sets on unoptimized code.

ter.

For unoptimized programs, the results follow the observations for pointer-sets – see Figure 2.6. Since the number of variables is low and the extent of the respective live-ranges is short, the way sets are represented and how blocks are traversed is of less importance: the performances mainly reveal the intrinsic overhead of the different implementations (the constant hidden in the big O notation), including artifacts stemming from the host compiler and the host machine. The possible gain (for large sets) obtained by performing unions of bitsets instead of successive insertions does not compensate yet the overhead of the loop-forest-based algorithm. The program size, i.e., the number of basic blocks and operations, has less impact on the variable-by-variable algorithm, which simply iterates over the small set of global variables, with a very light precomputation of def-use chains. The two other approaches, however, have to traverse the CFG and its operations in order to find upward-exposed uses, possibly intermixed with function calls that are not inlined. Our loop-forest algorithm cannot reach the performances of the two path-exploration solutions, which show an average speed-up of 80% for the var-by-var algorithm and 63% for the use-by-use variant. However, we already observe a clear improvement of 22% on average in comparison to the state-of-the-art iterative data-flow analysis.

The characteristics of optimized programs are, however, different. The structure of live-ranges is more complex and liveness sets are larger. For such programs, the standard iterative data-flow analysis is still the worst but, now, the variable-by-variable algorithm is performing worse than the two others, see Figure 2.7. The loop-forest-based approach clearly outperforms both path-exploration algorithms, with speed-ups of 69% and 43% respectively. This is explained by the relative cost of the fast bitset operations, in particular set unions, in comparison to the cost of traversing the CFG. Furthermore, the locality of memory accesses becomes a relevant performance factor. Both the use-by-use and the loop-forest algorithms operate *locally* on the bitsets surrounding a given program point. The inferior locality, combined with the necessary precomputation of the def-use chains, explains the poor results of the variable-by-variable approach in this experimental setting.

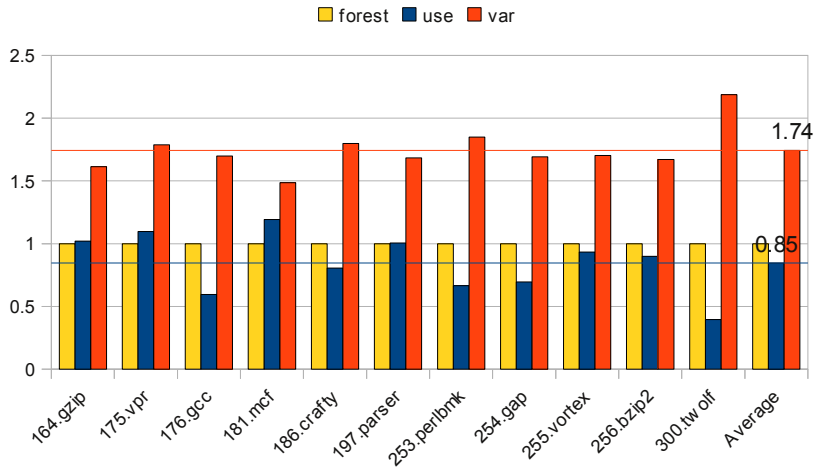


Figure 2.5: Speed-up with regard to our loop-forest-based approach using pointer-sets on optimized codes.

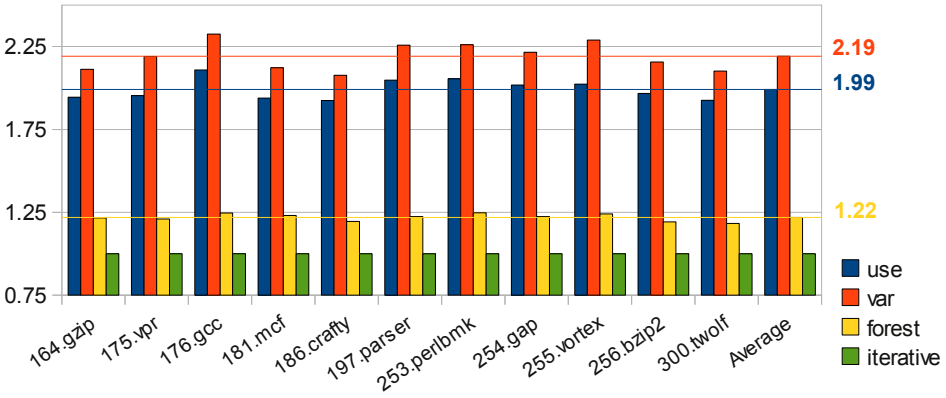


Figure 2.6: Speed-up w.r.t. iterative data-flow, bitsets, unoptimized programs.

Figure 2.8 shows more detailed results, relative to the standard iterative data-flow approach, on a per-module basis, i.e., using one data point for every source file. The loop-forest and the use-by-use algorithm on average clearly outperform the iterative computation by a factor of 2 and 1.4 respectively. The extreme cases showing speed-ups by a factor higher than 8 are caused by unusual – through relevant – loop structures in code generated by the parser generator *bison* (`c-parse.c` of `gcc`, and `perly.c` of `perlbmk`), which increase the number of iterations of the standard data-flow algorithm. On the other hand, all cases where the iterative approach outperforms the non-iterative are due to implementation artifacts: the analyzed functions do not contain any global variables thus slight variations in the executed code, the code placement, and the state of the data-caches become relevant. The variable-by-variable approach is often even slower than the iterative one and on average shows a speed-up of 18%.

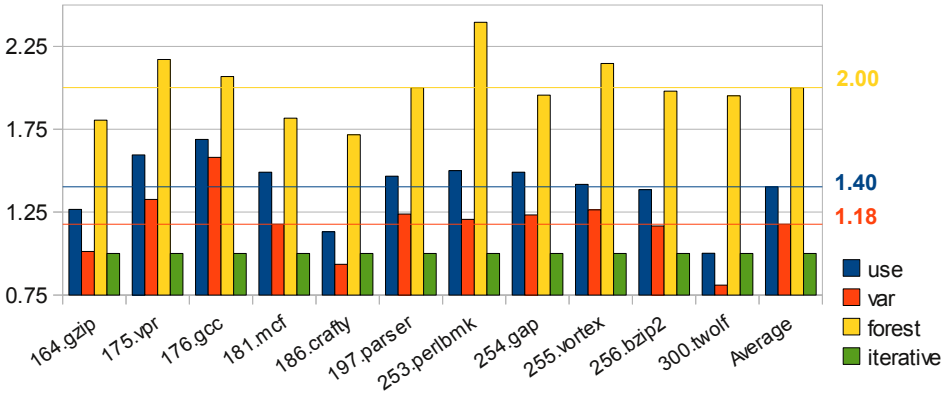


Figure 2.7: Speed-up w.r.t. iterative data-flow, bitsets, optimized programs.

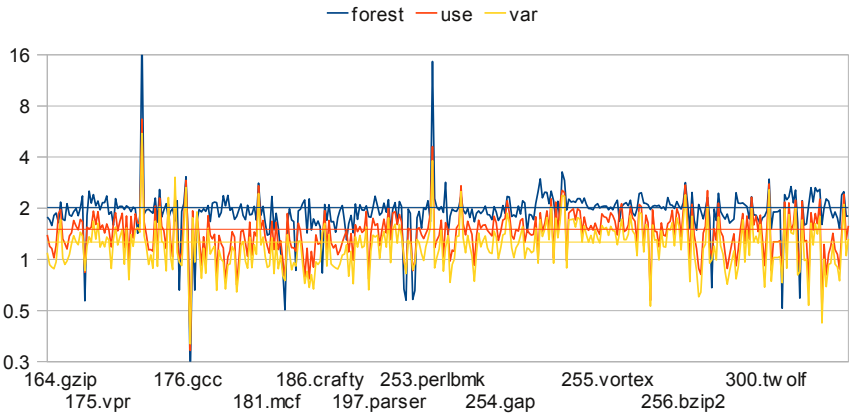


Figure 2.8: Speed-up w.r.t. iterative data-flow, bitsets, optimized programs.

2.6.3 Non-SSA-Form Programs

In addition to the algorithms that require SSA form to be available, we also considered the path-based approach for programs not under SSA. The implementation is based on pointer-sets, which showed the best speed-up for this particular algorithm variant in our previous experiments.

The algorithm requires a precomputation step in order to determine the sets of defined variables and upward-exposed uses. The relative speed-ups are thus diminished in comparison to the SSA-based algorithms, which have this information readily available. For unoptimized programs, the algorithm provides on average a gain of 12% in comparison to the standard iterative data-flow algorithm (also with pointer-sets), see Figure 2.9. The trend observed in our previous experiments is confirmed in this setting too. The results for optimized programs are much better, with an average speed-up of 22% over all benchmarks (Figure 2.10). Also, the results per-module (Figure 2.11) follow the previous findings, albeit with reduced gains. An interest-

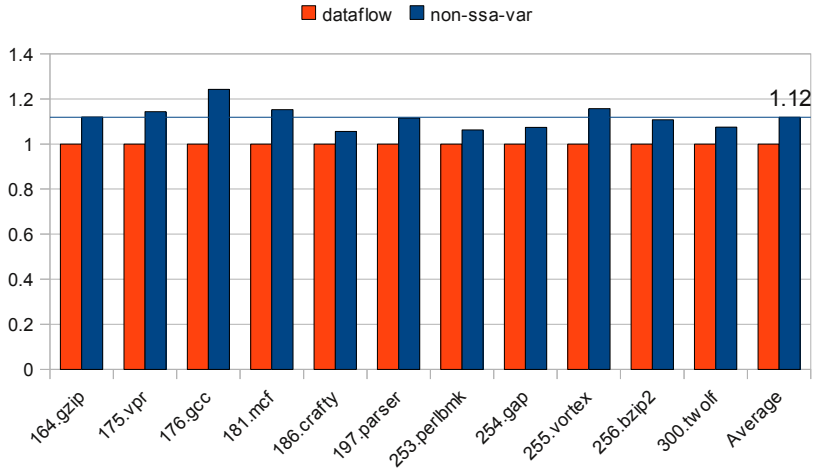


Figure 2.9: Speed-up of the variable-by-variable approach relative to iterative data-flow analysis using pointer-sets on unoptimized non-SSA programs.

ing detail is that the magnitude and the number of spikes indicating a slowdown in comparison to the data-flow algorithm is much smaller. Inspecting the involved benchmarks revealed that functions where the number of variables is exceedingly increased by SSA form and where the number of ϕ-operations is high are particularly affected.

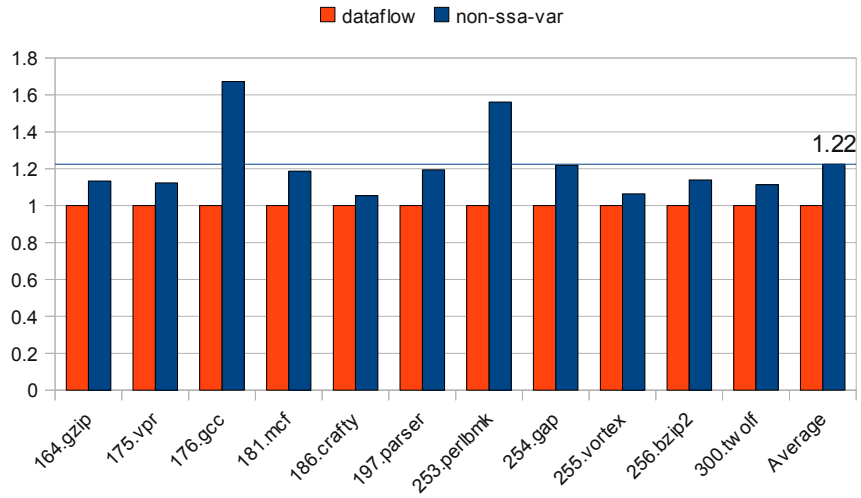


Figure 2.10: Speed-up of the variable-by-variable approach relative to iterative data-flow analysis using pointer-sets on optimized non-SSA programs.

If liveness sets are represented with bitsets, other alternatives may be designed, mixing the use-by-use approach with propagation of multiple variables together, as in standard data-flow algorithms. However, such a study is out of the scope of this chapter, which is primarily devoted to SSA programs.

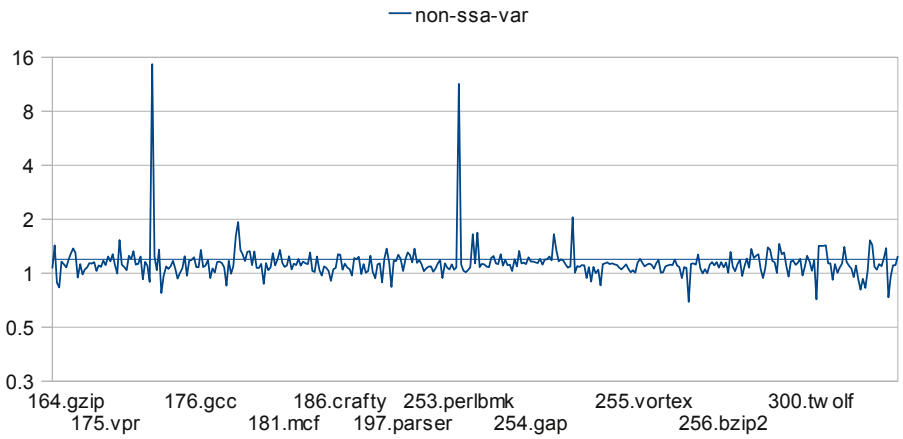


Figure 2.11: Speed-up of the variable-by-variable relative to iterative data-flow analysis on optimized non-SSA programs using pointer-sets.

2.7 Liveness Check using Loop Nesting Forest and Forward Reachability

The goal of this section is to revisit the liveness check algorithm proposed in [22] in the light of the properties and techniques developed in sections 2.4.2, 2.4.3, and 2.4.4 that we will recall below.

In contrast to liveness sets, liveness check does not provide the set of variables live at a block, but provides a query system to answer questions such as "is variable v live at location q?". Such a framework is well suited for tree-scan based register allocation [48], SSA destruction [17], or Hyperblock scheduling. Most register-pressure aware algorithms such as code-motion are not designed to take advantage of liveness check query system and still require sets. On the other hand, such a query system can obviously be built on top of precomputed liveness sets. Queries in $O(1)$ are possible, at least for basic block boundaries, providing the use of sparsesets [50] or bitsets to allow for efficient element-wise queries. If sets are only stored at basic block boundaries, to allow a query system at instructions granularity, the list of variables' uses or backward scans can be used. Constant time worst case complexity is lost in this scenario and liveness sets that have to be incrementally updated at each (even minor) code transformation, can be avoided and replaced by less memory consuming data structures that only depend on the CFG.

For completeness, we recall the prerequisites of the liveness check query system here:

- The CFG of the input program is available.
- The dominance tree of the CFG is available. Otherwise it is computable in $O(|V|)$. Within each basic block checking if one instruction precedes another should be possible in $O(1)$.
- A loop-nesting forest of the CFG is available. Also computable in $O(|V|\log^*|E|)$.
- A list of uses for each variable, also known as def-use chain is available. Having an easy-to-maintain def-use chain is one of the major advantages of the

SSA form. Hence, def-use chains are often available in SSA-based compilers. Updating the def-use chain when adding or removing uses of a variable incurs virtually no costs, quite contrary to updating liveness information on each change.

In the following, we consider the live-in query of variable a at node q. To avoid notational overhead, let a be defined in the CFG node $d := \mathsf{def}(a)$ and let $u \in \mathsf{uses}(a)$ be (w.l.o.g.) the single node where a is used. Suppose that q is strictly dominated by d (otherwise v cannot be live at q). Lemmas 2.1, 2.2, and 2.3 stated in Section 2.4.4.2 can be simplified as follow:

1. Let h be the header of the maximal loop containing q but not d. Let h be q if such maximal loop does not exist. Then v is live-in at h if and only if there exists a forward path that goes from h to u.

2. If v is live-in at the header of a loop then it is live at any node inside the loop.

In other words, v is live-in at q if and only if there exists a forward path from h to u where h is, if exists, the header of the maximal loop containing q but not d, q itself otherwise. Given the forward control-flow graph and the loop nesting forest, finding out if a variable is live at some program point can be done in two steps. First, if there exists a loop containing the program point q and not the definition, pick the header of the biggest such loop instead as the query point. Then check for reachability from q to any use of the variable in the forward CFG. As explained in Section 2.4.3, for irreducible CFG, the modified forward control flow graph that redirects any edge (s,t) to the loop header of the outermost loop containing t but excluding s (t.OLE(s)), has to be used instead. Correctness is proved from the theorems used for liveness sets.

Algorithm 13 puts a little bit more efforts onto the table to provide a query system at instructions granularity. If q is in the same basic block than d (lines 6-10, then v is live at q if and only if there is a use outside the basic block, or inside but after q. If h is a loop-header then v is live at q if and only if a use is forward reachable from h (lines 17-18). Otherwise, if the use is in the same basic block than q it must be after q to bring the variable live at q (lines 15-16). In this pseudo-code, upper cases are used for basic blocks while lower case are used for program points at instructions granularity. `def(a)` is an operand. `uses(a)` is a set of operands. basicBlock(u) returns the basic block containing the operand u. Given the semantic of the ϕ-function instruction, the basic block returned by this function for a ϕ-function operand can be different from the block where the instruction textually occurs. Also, `u.order` provides the corresponding (increasing) ordering in the basic block. For a ϕ-function operand, the ordering number might be greater than the maximum ordering of the basic block if the semantic of the ϕ-function places the uses on outgoing edges of the predecessor block. Q.OLE(D) corresponds to Algorithm 5 given in Section 2.4.4. forwardReachable(H,U) that tells if U is reachable in the modified forward CFG will be described further.

Live-out check algorithm given by Algorithm 14 only differs from Live-in check in lines 4, 9, and 14 that involve ordering comparisons. In line 4, if q is equal to d it cannot be live-in while it might be live-out; in lines 9 and 15 if q is at a use point it makes it live-in but not necessarily live-out.

```
Function IsLiveIn(programPoint q, var a)
begin
    d ← def(a);
    D ← basicBlock(d); Q ← basicBlock(q);
    if not(D sdom Q or (D = Q and order(d) < order(q))) then
        return false;
    end
    if Q = D then
        for u in uses(a) do
            U ← basicBlock(u);
            if U ≠ D or order(q) ≤ order(u) then
                return true;
            end
        end
        return false;
    end
    H ← Q.OLE(D);
    for u in uses(a) do
        U ← basicBlock(u);
        if (not isLoopHeader(H)) and U = Q and order(u) < order(q) then
            continue;
        end
        if forwardReachable(H, U) then
            return true;
        end
    end
    return false;
end
```

Algorithm 13: Live-In Check

```
Function IsLiveOut(programPoint q, var a)
begin
    d ← def(a);
    D ← basicBlock(d); Q ← basicBlock(q);
    if not (D sdom Q or (D = Q and order(d) ≤ order(q))) then      ▷ q must be
    dominated by the definition
        return false;
    end
    if Q = D then
        for u in uses(a) do
            U ← basicBlock(u);
            if U ≠ D or order(q) < order(u) then
                return true;
            end
        end
        return false;
    end
    H ← Q.OLE(D);
    for u in uses(a) do
        U ← basicBlock(u);
        if (not isLoopHeader(H)) and U = Q and order(u) ≤ order(q) then
            continue;
        end
        if forwardReachable(H, U) then
            return true
        end
    end
    return false;
end
```

Algorithm 14: Live-Out Check

2.7.1 Computing the modified-forward reachability

The liveness check query system relies on pre-computations for efficient OLE and forwardReachable queries. The outermost excluding loop is identical to the one used for liveness set. We explain how we compute the modified-forward reachability here. In practice we do not build explicitly the modified-forward graph. To compute efficiently the modified-forward reachability we simply need to traverse the modified-forward graph in a reverse topological order. A post-order initiated by a call to the recursive function DFS_Compute_forwardReachable(r) (Algorithm 15) will do the job. Bitsets can be used to efficiently implement sets of basic blocks. Once forward reachability have been pre-computed this way, forwardReachable(H,U) returns true if and only if $U \in H.forwardReachable$.

```
Function DFS_Compute_forwardReachable(block N)
begin
    N.forwardReachable ← ∅;
    N.forwardReachable.add(N);
    for each S ∈ succs(N) if (N,S) is not a loop-edge do
        H ← S.OLE(N);
        if H.forwardReachable = ⊥ then
            DFS_Compute_forwardReachable(H)
        end
        N.forwardReachable ← N.forwardReachable ∪ H.forwardReachable;
    end
end
```

Algorithm 15: Computation of modified-forward reachability using a traversal along a reverse topological order

Reachability and partial order realizers As explained in Section 2.3.2 encoding reachability in a directed acyclic graph can be done efficiently using a realizer of the corresponding partial order. To illustrate this point, let us consider the example of Figure 2.12a. This example corresponds to a switch region with no break at block A. A possible realizer of cardinality two is composed of the following topological orders $S \prec_1 A \prec_1 B \prec_1 C \prec_1 D \prec_1 E$ and $S \prec_2 D \prec_2 C \prec_2 A \prec_2 B \prec_2 E$. This gives the labeling represented in Figure 2.12a. Checking if B can reach C, simply relies in checking if $B \prec C$ for both orders i.e. if $(2,4) < (3,2)$: $B \not\prec_2 C$ (i.e. $4 \not< 2$) allows to conclude that there is no path from B to C. Reciprocally $S \prec C$ for both orders ($(0,0) < (3,2)$) allows to conclude that S can reach C.

To simplify the notations, when not ambiguous, there will be no difference between a DAG and the partial order associated to its transitive closure. Same for a graph labeling and a realizer. Hence, we can abusively talk about a realizer of a DAG and represent it formally using a labelling function, or talk about the dimension of a DAG...

The following theorem, derived from Hiragushi's dimension formula [138] for a modular decomposition substitution of a graph, allows to conclude that the forward CFG of a structured program is of dimension no more than two.

Definition 2.6 (module)**.** Let $G = (V,E)$ be a DAG. Let $G' = (M,E')$ be a sub-graph of G, i.e. $M \subset V$ and $E' = E \cap M \times M$. G' is a module of G if all vertices of M have the same set of neighbors among vertices not in M. In other words: (1) for each

$u \in V \backslash M$, if there exists $v' \in M$ reachable by u in G then every $v' \in M$ is reachable by u in G; (2) for each $u \in V \backslash M$, if there exists $v' \in M$ that can reach u in G then every $v' \in M$ can reach u in G;

Definition 2.7 (modular decomposition)**.** Let $G = (V, E)$ be a DAG, M a module of G. A modular decomposition of G by M, builds from G the graph $G'' = (V'', E'')$ by merging all vertices of M into a unique node m:

$$\begin{cases} V'' = V \cup \{m\} \backslash M \\ E'' = (E \cap V'' \times V'') \bigcup \{(v, m) | \exists v' \in M, (v, v') \in E\} \bigcup \{(m, v) | \exists v' \in M, (v', v) \in E\} \end{cases}$$

The reverse transformation is a substitution.

Theorem 2.8 (Hiragushi)**.** *Let G be a DAG, G' a module of G, G'' the modular decomposition of G by G'. Then,*

$$dim(G) = max(dim(G''), dim(G'))$$

As a direct consequence, if both G'' and G' are of dimension no more than d, then G is of dimension no more than d.

Lemma 2.9 (disconnected set of chains)**.** *A DAG made up of disconnected chains is of dimension no more than two.*

This can be seen as a direct consequence of Theorem 2.8 as: (1) a single chain is trivially of dimension one; (2) a graph made of two disconnected node is trivially of dimension two (take one order and its reverse).

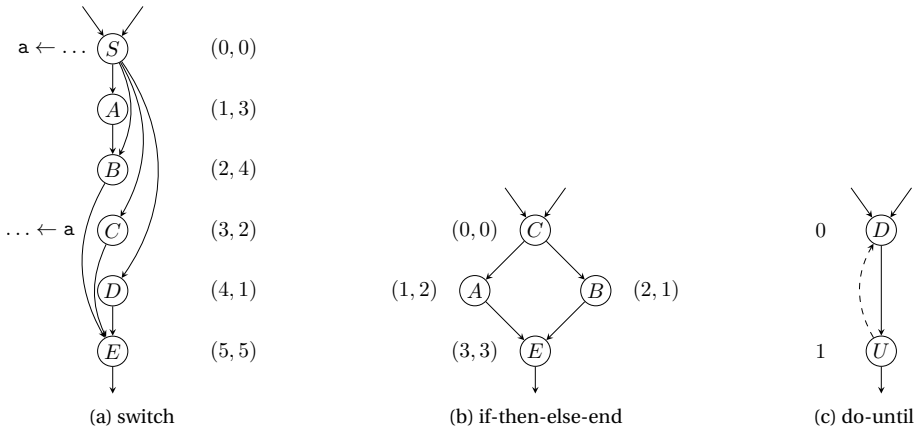
Corollary 2.10 (structured code)**.** *A structured code that uses if-then-else-end, switch, and do-until loop constructions is of dimension at most two.*

Proof. The DAG of a if-then-else-end is of dimension two: as represented in Figure 2.12b, a realizer is composed of the two following orders if$\prec_1$then$\prec_1$else$\prec_1$end and if$\prec_2$else$\prec_2$then$\prec_2$end. Indeed, this DAG can be decomposed using the disconnected set $M =$ {then,else} to the chain if-M-end.

The DAG of a switch is of dimension two. Indeed the sub-graph composed of all case nodes (such as $\{A, B, C, D\}$ for the example of Figure 2.12a) is a module of dimension no more than two: this sub-graph is a disconnected set of chains (the end of a chain being either a break or the default node); every node is reachable from the switch node (S in our example); and every node can reach the end node (E in our example). The modular decomposition leads to a simple chain switch-M-end of dimension one. Hence both orders fulfill switch$\prec M \prec$end, and chains of M are ordered in one order and its reverse.

The forward CFG of a do-until region is a chain, hence of dimension one.

Clearly, switch, if-then-else-end, and do-until that both have a single entry node and a single exit node are modules. By definition, structured code can be recursively reduced to a single node using those modules. The singleton and those modules being of dimension no more than two, this proves by induction that structured code are of dimension at most two. □



(a) switch (b) if-then-else-end (c) do-until

Figure 2.12: The CFG of a switch (implemented using a jump table), a if-then-else, and a do-until loop. Realizers of cardinality at most 2.

In the most general case, the dimension can be arbitrarily large (n for a crown graph with $2n$ vertices). Moreover finding a realizer of minimal cardinality is a NP-complete problem (for $d \geq 3$). However the problem is not to find a realizer of *minimal* cardinality, but one of *low* cardinality. Still, checking if a set of total orders is a realizer seems difficult to do in the general case in less than $O(|V|^2)$. On the other hand any set of topological orders, if not a realizer, provides a conservative *over-approximation* of reachability. In the most extreme situation, with no order at all, every node would be considered as reachable from every other node. A single order corresponds to the over-approximation used by a linear scan register allocation [113]. To illustrate this point consider the example of Figure 2.12a where only the first order is used for reachability queries. In that case, as $B \prec_1 C$, C is wrongly considered to be reachable from B. This over-approximation leads to a live-range for variable a made up of $\{S, A, B, C\}$ instead of $\{S, C\}$. Obviously the more orders, the more accurate is the reachability information. While a single order provides quite poor approximation in practice, we will describe a very simple mechanism for building two orders with a quite decent approximation.

The idea of our heuristic is two build two different reverse post orders using depth first search (DFS) traversals: during the first DFS the successors of a basic block are taken along the order of their respective index; during the second DFS, they are taken along the reverse order of their respective index. The pseudo-code, of linear complexity $O(|E|)$ is given in Algorithm 16.

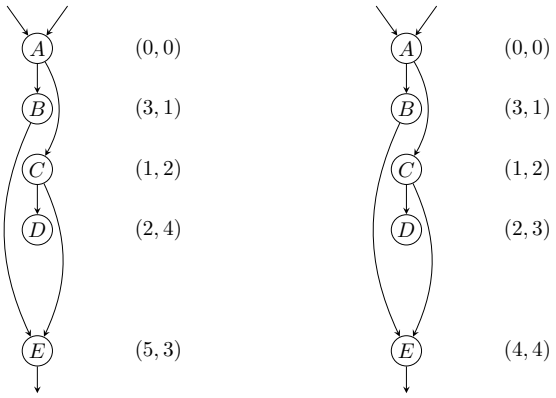
For structured code, DFS_Labeling_forwardReachability will provide a realizer. Unfortunately while loop construction, or early exit cannot be handled using modular decomposition and actual C codes are not structured. However, Table 2.2 shows that approximating reachability using two reverse post orders (RPO) as proposed by Algorithm 16 already pays compared to using only one RPO as a linear scan would do. In this table, the first column (linear) corresponds to approximating reachability using a single order (forwardReachable(U, V) restricts in checking if $U.label[1] \leq V.label[1]$) and leads to wrongly considering 26% of the couples $(u, v) \in V^2$ to be reachable; the second column (two orders) corresponds to approximating reachability using the two orders found by Algorithm 16 putting down the amount of false positive to 3.2%.

```
Function DFS_RPO(block N, integer o, integer phase)
begin
    Function CompareBlock(block a, block b)
    begin
        if phase = 1 then return Compare(index(a), index(b));
        else return Compare(index(b), index(a));
    end

    successors ← [] ;
    for each S ∈ succs(N) if (N,S) is not a loop-edge do
        H ← S.OLE(N);
        successors.add(H);
    end
    successors.Sort(&CompareBlock);
    for S in successors if S.label[phase] = ⊥ do
        o ← DFS_RPO(S, o, phase);
    end
    N.label[phase] ← o − 1;
    return o − 1;
end

Function DFS_Labeling_forwardReachability(block r)
begin
    DFS_RPO(r, |V|, 1);
    DFS_RPO(r, |V|, 2);
end

Function forwardReachable(block U, block V)
begin
    return U.label[1] ≤ V.label[1] and U.label[2] ≤ V.label[2];
end
```

Algorithm 16: Forward reachability using graph labeling.

benchmark	linear	two orders	two orders with pre-pass
164.gzip	8.6%	3.6%	0.69%
175.vpr	11%	0.1%	0.09%
176.gcc	24%	2.2%	1.8%
181.mcf	7.4%	0.9%	0.38%
186.crafty	37.5%	3.2%	0.39%
197.parser	12.8%	0.3%	0.16%
253.perlbmk	40.1%	14.9%	4.81%
254.gap	20.2%	5.2%	0.66%
255.vortex	6.2%	0.3%	0.38%
256bzip2	9%	0.1%	0.01%
300.twolf	10.8%	0.8%	0.57%

Table 2.2: Approximating reachability using sub-realizers. Percentage of false positive using one or two orders.



(a) With bias (b) Without

Figure 2.13: Graph labeling using two orders in the presence of a branch (through block D) that leaves the body of the program. Without bias, D is wrongly considered as being able to reach E.

The last column corresponds to a tentative for coping with early return statements, while loops, and continue statements that create branches that will never reach again the core of the program. Such branches are favored on one of the RPO traversal so as to assign them a big label as shown on Figure 2.13a. To implement this bias, Compare(index(a), index(b)) in CompareBlock is replaced by:

Compare(leafMin(a), leafMin(b)) || Compare(index(a), index(b)),

with leafMin(a) the minimum index over all leaf blocks (i.e. without successor) reachable from a. A pre-pass (along a depth-first-search) is required to compute leafMin value for each block. With this bias, a reachability query would be more than 99% of the time correct for most of the benchmarks. The overall ratio of false positive falls down to 2.1%. Finding a good labeling strategy that would take loop nesting forest into consideration so as to improve further the quality of the approximation is an opened problem.

2.7.2 Complexity

Live check framework is composed of two parts, the pre-computation which complexity depends only on the control flow graph, and the queries which complexity depends on the number of uses per variable $|U|$ and on the way loop nesting forest and reachability are encoded.

Each liveness query at block q of a variable v defined at block d and used at the set of blocks U, requires: (1) a dominance query performed in $O(1)$; (1) an outermost excluding loop query ($h = q$.OLE(d)); (2) and $|U|$ reachability queries (forwardReachable(h, u)), one for each use $u \in U$. The OLE query can be done in $O(d)$, d being the maximum loop depth of the program, if done naively; in $O\left(\frac{|L|}{B}\right)$, $|L|$ being the number of loops, and B the size of a bitset, if done using simple graph labeling; in $O(1)$ if done using sophisticated graph labeling. Each reachability query can be done in $O(1)$, whatever method is used. This gives a $O(|U| + d)$ complexity if the simplest method is implemented, and a $O(|U|)$ complexity if graph labeling is used.

If pre-computation is done for the outermost excluding loop queries: (1) basic graph labeling will have a space and time complexity of $O\left(|L|\left\lceil\frac{|L|}{B}\right\rceil\right)$; (2) RMQ based graph labeling will have a $O(|L|)$ space and time complexity.

Pre-computation for reachability information traverses the forward forward CFG and requires $O(|E|)$ OLE queries for this purpose. If bitsets of reachable blocks is computed for every block, $|E|$ merges (with $|E|$ the number of CFG edges) of cost $O(\frac{|V|}{B})$ are required. The space complexity to store this information would then be $O\left(|V|\left\lceil\frac{|V|}{B}\right\rceil\right)$ with $|V|$ the number of CFG blocks. Without any graph labeling, reachability thus has a time complexity of $O\left(|E|\left\lceil\frac{|V|}{B}\right\rceil\right)$ and a space complexity of $O\left(|V|\left\lceil\frac{|V|}{B}\right\rceil\right)$. With graph labeling, time complexity is $O(E)$ and space complexity $O(|V|)$.

2.8 Conclusion

Liveness information forms the basis for many compiler optimizations and transformations. However, many of these transformations invalidate the liveness information by introducing new variables and new instructions or by modifying the control-flow graph. Consequently, liveness analysis is performed several times throughout the compilation of an input program. Fast algorithms are thus required in order to minimize the penalty incurred by the steady re-computation.

The first contribution of this work is an improvement to the traditional iterative data-flow analysis in order to compute liveness information for programs in strict SSA form. The algorithm consists of only two major phases, instead of a variable number of iterations depending on the structure of the program. The first resembles the pre-computation phase of the original data-flow approach and provides partial liveness sets. The second phase replaces the iterative refinement of these partial liveness sets by a single traversal of a loop-nesting forest associated to the control-flow graph. The second contribution is the design and analysis of two algorithms that rely on path exploration to compute the program points where individual variables are known to be live and need to be added to the respective liveness sets. These algorithms similarly exploit properties provided by SSA-form programs in order to improve execution time. However, both variants can also be applied to regular programs, i.e., not in SSA form.

The computational complexity of our new algorithm is the same as these optimized techniques based on path exploration. As our experiments show, for strict SSA programs, all these algorithms outperform the iterative method by up to a factor of two on average for the SPECINT 2000 benchmark suite. Also, at least in our experiments, using bitsets always leads to faster implementations than with ordered lists. Depending on the program characteristics and the underlying representation of the liveness sets (ordered lists or bitsets), either the non-iterative data-flow algorithm or the algorithms using path exploration provide favorable execution times. For heavily optimized code having a high number of global variables and complex control flow, our non-iterative data-flow approach based on loop-forests is suited best, outperforming the others by at least 43%, whereas less optimized or even unoptimized code, having very few global variables, is best handled using one of the path exploration algorithms.

The second contribution of this work is to revisit liveness check using the properties developed for our fast liveness sets computation. The use of loop nesting forest simplifies both the theory and the implementation of the technique: given a variable defined at a program point d, checking if it is live at program point q, simply relies

in checking for each use u of v if the header of the outermost loop containing q but not d can forward-reach u. As a bonus, we took the opportunity to develop graph labeling techniques to allow $\langle O(|E|), O(1)\rangle$ (respectively $\langle O(|L|), O(1)\rangle$) reachability and outermost-excluding-loop queries. This extremely low complexity for reachability is to the cost of over-approximating the reachability (and thus liveness), just as a linear scan register allocator over-approximates live-ranges using intervals that cover parts of the code where the corresponding variables are actually not live. But if a linear scan shows 26% of false positive in our experiments (for SpecCint benchmarks at the code generation level of Open64 compiler), approximating reachability by intersecting only two linear extensions drops down the false positive ratio to 2%. The underlying theory of this technique is related to the dimension of partially ordered set (posets) that we show to be only two for structured code. Are posets of more general programs with continue, break, while loops, of bounded dimension? If yes, can a realizer be built with linear complexity as for structured code? Those are opened questions that would enable providing *exact* reachability in $\langle O(|E|), O(1)\rangle$ for more general realistic codes.

3

Machine Level SSA Destruction. Addressing Correctness, Quality, Speed and Memory Footprint

3.1 Introduction

SSA form is a sparse representation of program information, which enables simple, efficient code analysis and optimization. Once we have completed SSA based optimization passes, and certainly before code generation, it is necessary to eliminate ϕ-functions since these are not executable machine instructions. This elimination phase is known as *SSA destruction*.

Since freshly constructed, untransformed SSA code is conventional, its destruction is straightforward. One simply has to rename all ϕ-related variables (source and destination operands of a single ϕ-function are related) into a unique representative variable. Then each ϕ-function should have syntactically identical names for all its operands, and thus can be removed to coalesce the related live-ranges. We refer to a set of ϕ-related variables as a ϕ-web. Conventional SSA is defined as a flavor under which each ϕ-web is interference free. In particular if each ϕ-web's constituent variables have non-overlapping live-ranges then the SSA form is conventional. The discovery of ϕ-webs can be performed efficiently using the classical *union-find* algorithm with a disjoint-set data structure, which keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets.

While freshly constructed SSA code is conventional, this may not be the case after optimizations such as copy propagation have been performed. Going back to conventional SSA form implies the insertion of copies. The simplest (although not the most efficient) way to destroy non-conventional SSA form is to split all *critical edges*, and then replace ϕ-functions by parallel copies at the end of predecessor basic blocks. A critical edge is an edge from a node with several successors to a node with several predecessors. The process of splitting an edge, say (b_1, b_2), involves replacing edge (b_1, b_2) by (i) an edge from b_1 to a freshly created basic block and by (ii) another edge from this fresh basic block to b_2.

We stress that the above destruction technique suffers from several limitations and drawbacks: first, it works under implicit assumptions that are not necessarily fulfilled at machine level; second, it must rely on subsequent phases to improve back the bad code quality it generates; finally, it increases subsequently the size of the intermediate representation, thus making it not suitable for just-in-time compi-

lation.

Machine level code SSA at machine level complicates the process of destruction that can potentially lead to bugs if not performed carefully. The algorithm described above involves the splitting of every (critical) edges. Unfortunately, because of specific architectural constraints, region boundaries, or exception handling code, the compiler might not permit the splitting of a given edge. Fortunately, as we will see further, this obstacle can easily be overcame. But then it becomes essential to be able to append a copy operation at the very end of a basic block which might neither be possible. Also, care must be taken with duplicated edges, i.e. when the same basic block appears twice in the list of predecessors. This can occur after control flow graph structural optimizations like dead code elimination or empty block elimination. In such case, the edges should be considered as critical and then split.

SSA imposes a strict discipline on variable naming: every "name" must be associated to only one definition which is obviously most of the time not compatible with the instruction set of the targeted architecture. As an example, a two-address mode instruction, such as auto-increment would enforce its definition to use the same resource than one of its arguments (defined elsewhere), thus imposing two different definitions for the same temporary variable. This is why some prefer using, for SSA construction, the notion of versioning in place of renaming. Implicitly, two versions of the same original variable should not interfere, while two names can. The former simplifies the SSA destruction phase, while the latter simplifies and allows more transformations to be performed under SSA. Apart from dedicated registers for which optimizations are usually very careful in changing there live-range, register constraints related to calling conventions or instruction set architecture might be handled by the register allocation phase. However, there are some cases where we want those constraints to be expressed earlier (such as for the pre-pass scheduler), in which case the SSA destruction phase might have to cope with them.

Code quality The natural way of getting rid of ϕ-functions and expressing register constraints is through the insertion of copies (when edge-splitting is not mandatory as discussed above). If done carelessly, the resulting code will contain many temporary-to-temporary copy operations. In theory, reducing the frequency of these copies is the role of the coalescing during the register allocation phase. A few memory and time-consuming existing coalescing heuristics [27] mentioned in Chapter 1 can handle the removal of these copies effectively. The difficulty comes both from the size of the interference graph (the information of colorability is spread out) and the presence of many overlapping live-ranges that carry the same value (so non-interfering). Coalescing can also, with less effort, be performed prior to the register allocation phase. As opposed to a (so-called conservative) coalescing during register allocation, this aggressive coalescing would not cope with the interference graph colorability. As we will see, strict SSA form is really helpful for both computing and representing equivalent variables. This makes the SSA destruction phase the good candidate for eliminating (or not inserting) those copies.

Speed and Memory Footprint The cleanest and simplest way to perform SSA destruction with good code quality is to first insert copy instructions to make the SSA form conventional, then take advantage of the SSA form to run efficiently aggressive coalescing (without breaking the conventional property), before eventually renaming ϕ-webs and getting rid of ϕ-functions. Unfortunately this approach will lead, in a transitional stage, to an intermediate representation with a substantial number of variables: the size of the liveness sets and interference graph classically used to perform coalescing become prohibitively large for dynamic compilation. To over-

come this difficulty one can compute liveness and interference on demand which, as we already mentioned, is made simpler by the use of SSA form. Remains the process of copy insertion itself that might still take a substantial amount of time. To fulfill constraints imposed by just-in-time compilation, the idea is to *virtually* insert those copies, and only *effectively* insert the un-coalesced ones.

This chapter addresses those three issues: handling of machine level constraints, code quality (elimination of copies), and algorithm efficiency (speed and memory footprint). The layout falls into three corresponding sections.

3.2 Machine Level Constraints

Isolating ϕ-node using copies In most cases, edge splitting can be avoided by treating symmetrically ϕ-usesand ϕ-def: instead of just inserting copies on the incoming control-flow edges of the ϕ-node (one for each use operand), a copy is also inserted on the outgoing edge (one for its defining operand). This has the effect of isolating the value associated to the ϕ-node thus avoiding (as discussed further) SSA destruction issues such as the well known lost-copy problem. The corresponding pseudo-code is given in Algorithm 17. If, because of different ϕ-functions, several copies are introduced at the same place, they should be viewed as parallel copies. For that reason, an empty parallel copy is inserted both at the beginning and at the end of each basic-block. Note that, as far as correctness is concerned, copies can be sequentialized in any order, as they concern different variables.

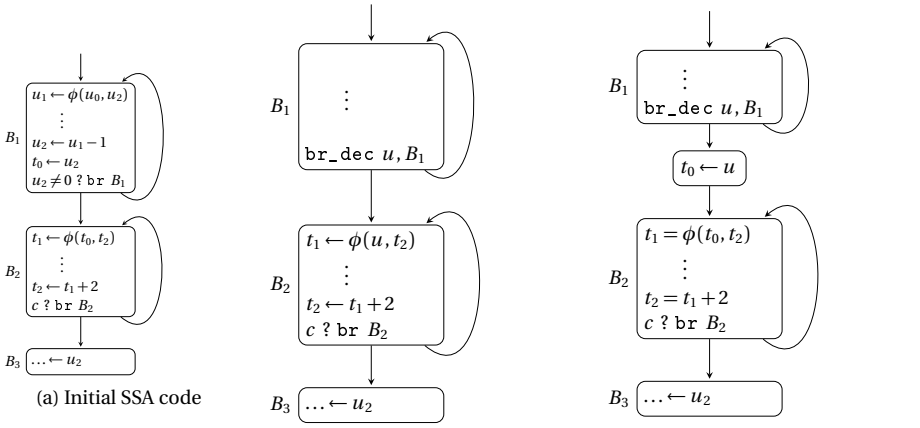
As soon as incoming edges are not split, without the copy from a'_0 to a_0, the ϕ-function defines directly a_0 whose live range can be long enough to intersect the live range of some a'_i, $i > 0$, if a_0 is live out of the block B_i where a'_i is defined. Two cases are possible: either a_0 is used in a successor of $B_i \neq B_0$, in which case the edge from B_i to B_0 is *critical* (as in the "lost copy problem"), or a_0 is used in B_0 as a ϕ-function argument (as in the "swap problem"). In this latter case, if parallel copies are used, a_0 is dead before a'_i is defined but, if copies are sequentialized blindly, the live range of a_0 can go beyond the definition point of a'_i and lead to incorrect code after renaming a_0 and a'_i with the same name. ϕ-node isolation allows to solve most of the issues that can be faced at machine level. However, there remains subtleties listed below.

Limitations There is a tricky case, when the basic block contains variables *defined after* the point of copy insertion. This is the case for some DSP-like branch instructions with a behavior similar to hardware looping. In addition to the condition, a counter u is decremented by the instruction itself. If u is used in a ϕ-function in a direct successor block, no copy insertion can split its live range. It must then be given the same name as the variable defined by the ϕ-function. If both variables interfere, this is just impossible! To solve the problem, the SSA optimization could be designed with more care, or the counter variable must not be promoted to SSA, or some instruction must be changed, or the control-flow edge must be split somehow. SSA destruction by copy insertion alone is not always possible, depending on the branch instructions and the particular case of interferences.

For example, suppose that for the code of Figure 3.1(a), the instruction selection chooses a branch with decrement (denoted Br_dec) for Block B_1 (Figure 3.1(b)). Then, the ϕ-function of Block B_2, which uses u, cannot be translated out of SSA by standard copy insertion because u interferes with t_1 and its live range cannot be split. To go out of SSA, one could add $t_1 = u - 1$ in Block B_1 to anticipate the branch.

```
begin
    foreach B: basic block of the CFG do
        insert an empty parallel copy at the beginning of B;
        insert an empty parallel copy at the end of B;
    end
    foreach B_0: basic block of the CFG do
        foreach φ-function at the entry of B_0 of the form
        a_0 = φ(B_1 : a_1, ..., B_n : a_n) do
            foreach a_i (argument of the φ-function corresponding to B_i) do
                let PC_i be the parallel-copy at the end of B_i;
1
                let a'_i be a freshly created variable;
                add copy a'_i ← a_i to PC_i;
                replace a_i by a'_i in the φ-function;
            end
            begin
                let PC_0 be the parallel-copy at the beginning of B_0;
2
                let a'_0 be a freshly created variable;
                add copy a_0 ← a'_0 to PC_0;
                replace a_0 by a'_0 in the φ-function;
            end
            /* all a'_i can be coalesced and the φ-function removed
               */
        end
    end
end
```

Algorithm 17: Algorithm making non-conventional SSA form conventional by isolating ϕ-nodes



(a) Initial SSA code

(b) Branch with decrement (Br_dec)

(c) CSSA with additional edge splitting

Figure 3.1: Copy insertion may not be sufficient.

Or one could split the critical edge between B_1 and B_2 as in Figure 3.1(c). In other words, simple copy insertions is not enough in this case.

There is another tricky case when a basic-block have twice the same predecessor block. This can result from consecutively copy-folding and control flow graph structural optimizations like dead code elimination or empty block elimination. This is the case for the example of Figure 3.2 where copy-folding would remove the copy $a_2 = b$ in basic-block L_2. If L_2 is eliminated, there is no way to implement the control dependence of the value to be assigned to a_3 other than through predicated code or through the re-insertion of a basic-block between L_1 and L_0 by the split of one of the edges.

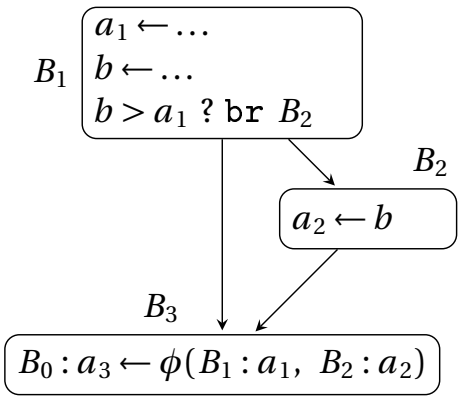


Figure 3.2: Copy-folding followed by empty block elimination can lead to SSA code for which destruction is not possible through simple copy insertion

The last difficulty SSA destruction can face when performed at machine level is related to register constraints such as instruction set architecture (ISA) or application binary interface (ABI) constraints. For the sake of the discussion we differ-

entiate two kinds of resource constraints that we will refer as *operand pinning* and *live-range pinning*. Live-range pinning expresses the fact that the *entire* live-range of a variable must reside in a given resource (usually a dedicated register). An example of live-range pinning are versions of the stack-pointer temporary that must be assigned back to register *SP*. On the other hand the pinning of an operation's operand to a given resource does not impose anything on the live-range of the corresponding variable. The scope of the constraint is restricted to the operation. Examples of operand pinning are operand constraints such as *2-address-mode* where two operands of the same instruction must use the same resource; or where an operand must use a given register. This last case encapsulates ABI constraints. Note that more loose constraints where the live-range or the operand can reside in more than one resource are not handled here. We assume this to always be the responsibility of register allocation.

The live-range pinning of a variable v to resource R will be represented R_v, just as if v were a version of temporary R. An operand pinning to a resource R will be represented using the exponent $^{\uparrow R}$ on the corresponding operand.

We first simplify the problem by transforming any operand pinning to a live-range pinning as sketched in Figure 3.3: parallel copies with new variables pinned to the corresponding resource are inserted just before (for use-operand pinning) and just after (for definition-operand pinning) the operation.

$$p_2^{\uparrow T} = p_1^{\uparrow T} + 1$$

(a) Operand pinning of an auto-increment

$$\begin{aligned} T_{p_2} &= p_1 \\ p_2 &= T_{p_2} + 1 \end{aligned}$$

(b) Corresponding live-range pinning

$$a^{\uparrow R_0} = f(b^{\uparrow R_0}, c^{\uparrow R_1})$$

(c) Operand pinning of a function call

$$\begin{aligned} b'_{R_0} &= b \parallel c'_{R_1} = c \\ a'_{R_0} &= f(b'_{R_0}, c'_{R_1}) \\ a &= a'_{R_0} \end{aligned}$$

(d) Corresponding live-range pinning

Figure 3.3: Operand pinning and corresponding live-range pinning

Detection of strong interferences The scheme we propose in this section to perform SSA destruction that deals with machine level constraints does not address compilation cost (in terms of speed and memory footprint). It is designed to be simple. It first inserts parallel copies to isolate ϕ-functions and operand pinning. Then it checks for interferences that would persist. We will denote such interferences as strong, as they cannot be tackled through the simple insertion of temporary-to-temporary copies in the code. We consider that fixing strong interferences should be done on a case by case basis and restrict the discussion here on their detection.

As far as correctness is concerned, Algorithm 17 splits the data-flow between variables and ϕ-nodes through the insertion of copies. For a given ϕ-function $a_0 = \phi(a_1, \ldots, a_n)$, this transformation is correct as soon as the copies can be inserted close enough to the ϕ-function. It might not be the case if the insertion point (for a use-operand) of copy $a'_i \leftarrow a_i$ is not dominated by the definition point of a_i; symmetrically, it will not be correct if the insertion point (for the definition-operand) of copy $a_0 \leftarrow a'_0$ does not dominate all the uses of a_0. Precisely this leads to inserting in Algorithm 17 the following tests:

- line 1: "**if** the definition of a_i does not dominate PC_i **then** continue;"

- line 2: "**if** one use of a_0 is not dominated by PC_0 **then** continue;"

For the discussion, we will design as *split operands* the newly created local variables to differentiate them to the ones concerned by the two previous cases (designed as *non-split operands*). We suppose a similar process have been performed for operand pinning to express them in terms of live-range pinning with very short (when possible) live-ranges around the concerned operations.

At this point, the code is still under SSA and the goal of the next step is to check that it is conventional: this will obviously be the case only if all the variables of a ϕ-web can be coalesced together. But not only: the set of all variables pinned to a common resource must also be interference free. We say that x and y are *pin-ϕ-related* to one another if they are ϕ-related or if they are pinned to a common resource. The transitive closure of this relation defines an equivalence relation that partitions the variables defined locally in the procedure into equivalence classes, the pinne-ϕ-webs. Intuitively, the pin-ϕ-equivalence class of a resource represents a set of resources "connected" via ϕ-functions and resources. The computation of ϕ-webs mentioned earlier can be generalized easily to compute pin-ϕ-webs. The resulting pseudo-code is given by Algorithm 18.

Now, one need to check that each web is interference free. A web contains variables and resources. A variable and a resource do not interfere while two distinct physical resources will interfere with one another.

If any interference have been discovered, it has to be fixed on a case by case basis. Note that some interferences such as the one depicted in Figure 3.2 can be detected and handled initially (through edge splitting if possible) during the copy insertion phase.

```
begin
    for each resource R do
        web(R) ← {R};
    end
    for each variable v do
        web(v) ← {v};
        if v pinned to a resource R then
            union(web(R), web(v))
        end
    end
    for each instruction of the form a_dest = φ(a_1, ..., a_n) do
        for each source operand a_i in instruction do
            union(web(a_dest), web(a_i))
        end
    end
end
```

Algorithm 18: The pin-ϕ-webs discovery algorithm, based on the union-find pattern

3.3 Code Quality

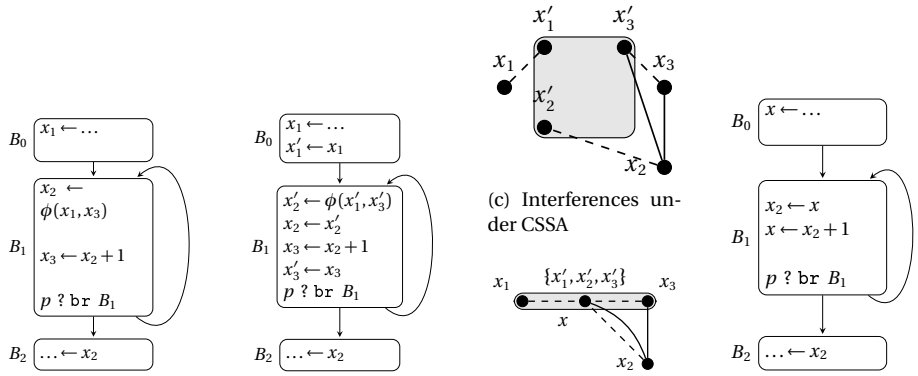
Once the code is in Conventional SSA, destructing it is straightforward, which solves the correctness problem.

Aggressive coalescing To improve the code however, it is important to remove as many copies as possible. This can be treated with classic coalescing as conventional SSA allows to get rid of ϕ-functions: the set of variables of a SSA web can be coalesced leading to a single non-SSA variable; liveness and interferences can then be defined as for regular code (with parallel copies). An interference graph (as depicted in Figure 3.4d) can be used. A plain edge between two nodes (e.g. between x_2 and x_3) materialize the presence of an interference between the two corresponding variables (e.g. between x_2 and x_3), i.e. expressing the fact that they cannot be coalesced and share the same resource. A dashed edge between two nodes materializes an affinity between the two corresponding variables, i.e. the presence of a copy (e.g. between x_2 and u_0) that could be removed by their coalescing.

This process is illustrated by Figure 3.4: the isolation of the ϕ-function leads to inserting the three copies that respectively define u_1 and u_2 and uses u_0; the corresponding ϕ-web $\{u_0, u_1, u_2\}$ is coalesced into a representative variable u; according to the interference graph of Figure 3.4d, x_1, x_5 can then be coalesced with u leading to the code of Figure 3.4e.

Liveness under SSA If the goal is not to destruct SSA completely but remove as many copies as possible while maintaining the conventional property, liveness of ϕ-function operands should reproduce the behavior of the corresponding non-SSA code as if the variables of the ϕ-web were coalesced all together. The semantic of the ϕ-operation in the so called *multiplexing* mode fits the requirements. The corresponding interference graph on our example is depicted in Figure 3.4c.

Definition 3.1 (multiplexing mode)**.** Let a ϕ-function $B_0 : a_0 = \phi(B_1 : a_1, \ldots, B_n : a_n)$ be in *multiplexing* mode, then its liveness follows the following semantic: its definition-operand is considered to be at the entry of B_0, in other words variable a_0 is live-in of B_0; its use-operands are at the exit of the corresponding predecessor basic-blocks, in other words variable a_i for $i > 0$ is live-out of basic-block B_i.



(a) Lost-copy problem (b) Corresponding CSSA code (c) Interferences under CSSA (d) Interferences and coalescing (e) After copy optimization

Figure 3.4: Out-of-SSA translation for the lost-copy problem.

Value-based interference As said earlier, after the ϕ-isolation phase, and the treatment of operand pinning constraints the code contains many overlapping live-ranges

that carry the same value. Because of this, to be efficient coalescing must use an accurate notion of interference. It is common to find in the literature the following definition of interference "two variables interfere if their live ranges intersect" (e.g. in [68; 39; 129]) or its refinement "two variables interfere if one is live at a definition point of the other" (e.g. in [41]). In fact, a and b interfere only if they cannot be stored in a common register. Chaitin et al. discuss more precisely the "ultimate notion of interference" [42]: a and b cannot be stored in a common register if there exists an execution point where a and b carry *two different values* that are both defined, used in the future, and not redefined between their definition and use. This definition of interference contains two dynamic (i.e., related to the execution) notions: the notion of liveness and the notion of value. Analyzing statically if a variable is live at a given execution point is a difficult problem. This can be approximated (quite accurately in practice) using data flow reaching definition and upward exposed use [10]. In strict SSA form – in which each use is dominated by its unique definition – upward exposed use analysis as developed in Chapter 2 is sufficient. The notion of value is even harder, but may be approximated using data-flow analysis on specific lattices [5; 25]. This has been extensively studied in particular in the context of partial redundancy elimination. The scope of variable coalescing is usually not so large, and Chaitin proposed a simpler conservative test: *two variables interfere if one is live at a definition point of the other and this definition is not a copy between the two variables*. This interference notion is the most commonly used, see for example how the interference graph is computed in [10].

Chaitin et al. noticed that, with this conservative interference definition, when a and b are coalesced, the set of interferences of the new variable may be strictly smaller than the union of interferences of a and b. Thus, simply merging the two corresponding nodes in the interference graph is an over-approximation with respect to the interference definition. For example, in a block with two successive copies $b = a$ and $c = a$ where a is defined before, and b and c (and possibly a) are used after, it is considered that b and c interfere but that none of them interfere with a. However, after coalescing a and b, c should not interfere anymore with the coalesced variable. Hence, the interference graph has to be updated or rebuilt. Chaitin et al. [42] proposed a counting mechanism, rediscovered in [61], to update the interference graph, but it was considered to be too space consuming. Recomputing it from time to time was preferred [42; 41]. Since then, most coalescing techniques based on graph coloring use either live range intersection graph [130; 39] or Chaitin's interference graph with reconstructions [68; 36].

However, in SSA, each variable has, statically, a *unique* value, given by its unique definition. Furthermore, the "has-the-same-value" binary relation defined on variables is, if the SSA form fulfills the dominance property, an equivalence relation. The *value* of an equivalence class [1] is the variable whose definition dominates the definitions of all other variables in the class. Hence, using the same scheme as in SSA copy folding, finding the value of a variable can be done by a simple topological traversal of the dominance tree: when reaching an assignment of a variable b, if the instruction is a copy $b = a$, $V(b)$ is set to $V(a)$, otherwise $V(b)$ is set to b. The interference test in now both simple and accurate (no need to rebuild/update after a coalescing): if live(x) denotes the set of program points where x is live,

$$a \textit{ interfere with } b \textit{ if live}(a) \textit{ intersects live}(b) \textit{ and } V(a) \neq V(b)$$

[1] Dominance property is required here. e.g. consider the following loop body if($i \neq 0$) $\{b \leftarrow a;\}$ $c \leftarrow \ldots$; $\cdots \leftarrow b$; $a \leftarrow c$; the interference between b and c is actual.

The first part reduces to def(a) $\in$ live(b) or def(b) $\in$ live(a) thanks to the dominance property [39]. In the previous example, a, b, and c have the same value $V(c) = V(b) = V(a) = a$, thus they do not interfere.

Note that our notion of values is limited to the live ranges of SSA variables, as we consider that each ϕ-function defines a new variable. We could propagate information through a ϕ-function when its arguments are equivalent (same value). But, we would face the complexity of general value numbering. By comparison, our equality test in SSA comes for free.

Shared copies It turns out that after the ϕ-isolation phase, and the treatment of operand pinning constraints the code also contains what we design as shared copies. A *shared copy* corresponds precisely to the previous example of two successive copies $b = a$ and $c = a$ i.e. the presence of two copies from the same source. We have seen that, thanks to our definition of value, the fact that b is live at the definition of c does not imply that b and c interfere. Suppose however that a (after some other coalescing) interferes with b and c. Then, no coalescing can occur although coalescing b and c would save one copy, by "sharing" the copy of a. This sharing problem is difficult to model and optimize (the problem of placing copies is even worse), but we can optimize it a bit. We coalesce two variables b and c if they are both copies of the same variable a and if their live ranges intersect. This can be done in a second pass after all standard affinities have been treated. Note that if their live ranges are disjoint, such a coalescing may be incorrect as it would increase the live range of the dominating variable, possibly creating some interference not taken into account.

3.4 Speed and Memory Footprint

Implementing the technique of the previous section may be considered too costly. First, it inserts many instructions before realizing most are useless, and copy insertion is already by itself time-consuming. It introduces many new variables, too. The size of the variable universe has an impact on the liveness analysis and the interference graph construction. Also, if a general coalescing algorithm is used, a graph representation with adjacency lists (in addition to the bit matrix) and a working graph to explicitly merge nodes when coalescing variables, would be required. All these constructions, updates, manipulations are time-consuming and memory-consuming. We may improve the whole process by: a) avoiding the use of any interference graph and liveness sets; b) avoid the quadratic complexity of interference check between two sets of variables by an optimistic approach that first coalesces even interfering variables, then traverses each set of coalesced variables and un-coalesce one by one all the interfering ones(this is the technique advocated by Budimlic et al. in [39]); c) emulating ("virtualizing") the introduction of the ϕ-related copies.

Interference check Liveness sets and interference graph are the major source of memory usage. This motivates, in the context of JIT compilation, not to build any interference graph at all, and rely on the liveness check described in Chapter 2 to test if two live-ranges intersect or not. Let us suppose for this purpose that a "has-the-same-value" equivalence relation, is available thanks to a mapping V of variables to symbolic values:

$$\text{variables } a \text{ and } b \text{ have the same value} \Leftrightarrow V(a) = V(b)$$

As explained in Paragraph 3.3 this can be done linearly (without requiring any hash map-table) on a single traversal of the program if under strict SSA form. We also suppose that liveness check is available, meaning that for a given variable a and program point p, one can answer if a is live at this point through the boolean value of $a.islive(p)$. This can directly be used, under strict SSA form, to check is two variables live-ranges, say a and b intersect:

$$\begin{array}{lcl} \mathsf{intersect}(a,b) & \Leftrightarrow & liverange(a) \cap liverange(b) \neq \emptyset \\ & & \left\{ \begin{array}{l} a.def.op = b.def.op \\ a.def.op \text{ dominates } b.def.op \bigwedge a.islive\big(out(b.def.op)\big) \\ b.def.op \text{ dominates } a.def.op \bigwedge b.islive\big(out(a.def.op)\big) \end{array} \right. \\ & \Leftrightarrow & \end{array}$$

Which leads to our refined notion of interference:

$$\mathsf{interfere}(a,b) \Leftrightarrow \mathsf{intersect}(a,b) \bigwedge V(a) \neq V(b)$$

De-coalescing in linear time The interference checking outlined in the previous paragraph allows to avoid building an interference graph of the SSA form program. However, coalescing has the effect of merging vertices and interference queries are actually to be done between sets of vertices. Using an interference graph, one can manipulate a *working graph* which vertices corresponds to sets of coalesced variables, and update it along ongoing coalescing. Without any interference graph, one can "naively" test a quadratic number of interferences for each interference query. With more efforts [21], linear complexity is possible, but the technique considered here is even more radical. The idea is to first merge all copy and ϕ-function related variables together. A merged-set might of course contain interfering variables at this point. The idea is to identify some variables that interfere with some other variables within the merged-set, and remove them from the merged-set. As we will see here, thanks to the dominance property, this can be done linearly using a single traversal of the set.

In reference with register allocation, and graph coloring, we will associate the notion of colors to merged-sets: every variables of the same set are assigned the same color, and different sets are assigned different colors. The process of *de-coalescing* a variable is to extract it from its set; it is not put in another set, just isolated. So we will refer such a variable as *un-colored*. Actually, variables pinned together have to stay together. So the process of un-coloring a variable might have the effect of un-coloring some others. In other words, a colored variable is to be coalesced with variables of the same color, and any un-colored variable is to be coalesced only with the variables it is pinned with.

We suppose that variables have already been colored and the goal is to un-color some of them (preferably not all of them) so that each merged-set become interference free. We suppose that if two variables are pinned together they have been assigned the same color, and that a merged-set cannot contain variables pinned to different physical resources. Here we focus on a single merged-set and the goal is to make it interference free within a single traversal. The ideaadvocated by Budimlic et al. [39] exploits the tree shape of variables live-ranges under strict SSA. To this end, variables are identified by their definition point and ordered using dominance accordingly. Traversal of the set is performed along the dominance order, enforcing at each step the sub-set of already considered variables to be interference free. From now, we will abusively design as the dominators of a variable v, the set of variables

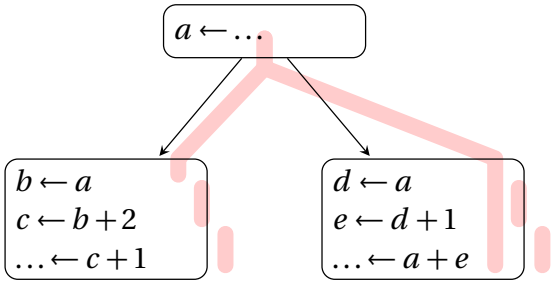


Figure 3.5: Variables live-ranges are sub-trees of the dominator tree

of *identical color than* v which definition dominate the definition of v. Variables defined at the same program point are arbitrarily ordered, so as to use the standard definition of immediate dominator (denoted $v.idom$, set to $\perp$ if not exists).

First, consider the example of Figure 3.5 to illustrate the following properties. Consider the current variable v, and an interfering dominating variable *cur_anc*. Clearly, *cur_anc* intersects (or is identical to) $v.idom$: for our example, as a does not intersect $c.idom = b$ it cannot intersect c. On the other-end, with e as the current variable, the intersection of e with a implies the intersection of a with $e.idom = d$. If by construction d and a do not interfere this implies that $V(a) = V(d)$... As a consequence, if one suppose that at a given step the set of dominating variables is interference free, checking if the current variable, say v, interferes with one of its dominating ones, consists in: (1) check if v interfere with its immediate dominator, $u = v.idom$; (2) if not, walk-up the dominators (*cur_anc*) of u that both intersect u and have the same value V than u: check the interference with v. Walking up along this set is done in Algorithm 19, through the use of $v.eanc$ that points to the immediate dominating such variable. In this algorithm, we suppose that traversing the merged-set along the dominance order is possible, and that dominance test is available. As for the standard renaming pass during SSA construction, the *idom* field is set during the traversal as the updated value of *cur_idom*.

Virtualizing ϕ-related copies The last step toward a memory friendly and fast SSA-destruction algorithm consists in emulating the initial introduction of copies and only actually insert them on the fly when they appear to be required. We use *exactly the same algorithms as for the solution without virtualization*, and use a special location in the code, identifies as a "virtual" parallel copy, where the real copies, if any, will be placed.

Because of this, any non-coalesced split operand of a ϕ-function is assumed to have a use (resp. definition) in the parallel copy, and are then usually (unless used further) not considered as live-out (resp. live-in) of the corresponding basic-block. For any non-split operand, or when a virtual copy is coalesced, the use (resp. definition) operand is assumed (as in the multiplexing mode) to be at the exit (resp. entry) of the corresponding basic-block, so it is considered as live-out (resp. live-in). When the algorithm decides that a virtual copy $a'_i \leftarrow a_i$ (resp. $a_0 \leftarrow a'_0$) cannot be coalesced, it is *materialized* in the parallel copy and a'_i (resp. a'_0) becomes explicit in its merged-set. The corresponding ϕ-operation is replaced and the use of a'_i (resp. def of a'_0) is now assumed, as in the multiplexing mode, to be on the corresponding control flow edge. This way, only copies that the first approach would finally leave

un-coalesced are introduced.

Without any virtualization, the process of transforming operand pinning into live-range pinning also introduces copies and new local variables pinned together. This systematic copy insertion can also be avoided and managed lazily just as for ϕ-nodes isolation(see Chapter 4 for an illustration). We will not address this aspect of the virtualization here: to simplify we consider any operand pinning to be either ignored (handled by register allocation) or expressed as live-range pinning.

In our scheme, every copy related (even virtual) variables are first coalesced (unless pinning to physical resources forbid it), then merged-sets are traversed and interfering variables are de-coalesced. A key implementation aspect is related to the handling of pinning. In particular, for correctness purpose coalescing have to be performed in two separated steps. First pinned-ϕ-webs have to be coalesced. Detection and fix of strong interferences is handled at this point. The so obtained merged sets (that contain local virtual variables) have to be identified as atomic i.e. they cannot be separated. After the second step of coalescing, atomic merged sets will compose larger merged sets. A variable cannot be de-coalesced from a set without de-coalescing its atomic merged-set from the set also. Non singletons atomic merged-sets have to be represented somehow. The trick is to use the ϕ-function itself as a placeholder for its set of local virtual variables: the pinning of the virtual variables is represented through the pinning of the corresponding ϕ-function. As a consequence, any ϕ-function will be pinned to all it non-split operands.

Algorithm 20 presents the process of de-coalescing with virtualization of ϕ-related copies, using a single traversal of the whole program. Here each merged set is identified by a color. We suppose that the (interference free) atomic merged set containing a variable v is available through the function atomic-merged-set(v). $c.cur\_idom$ stores the last processed variable of color c. Just as for SSA renaming phase, thanks to $u.idom$ link that points to the immediate dominator of variable u, the immediate dominator of the currently processed variable can be found. $u.eanc$ points to the lowest dominator of u which live-range intersects u and have the same value than u. Walking up along *eanc* links allows to find the lowest dominator that intersects the current variable. A correctness subtlety is related to the interferences with a variable that is not yet materialized or coalesced. Either one need to ensure that they can be ignored or one need to emulate them. We choose to emulate them, which allows to postpone the materialization of all copies along a single traversal of the program at the really end of the de-coalescing process. As the live-range of a local virtual variable is very small starting (resp. ending for a ϕ-definition operand) at the place of the parallel copy and ending at the exit (resp. starting at the entry) of the corresponding basic-block, the cases to consider is quite limited. A local virtual variable can interfere with a "real" variable *cur_anc*, which is detected in Algorithm 22 just as in Algorithm 21. It can also interfere with another virtual variable, the last processed being stored thanks to $c.curphi$. The materialization, is straightforward: whenever one of the two operands of a virtual copy is uncolored, or whenever the colors are different, the copy is materialized. This can be done through a single traversal of all ϕ-functions of the program.

Sequentialization of parallel copies During the whole algorithm, we treat the copies placed at a given program point as *parallel copies*, which are indeed the semantics of ϕ-functions. This gives several benefits: a simpler implementation, in particular for defining and updating liveness sets, a more symmetric implementation, and fewer

constraints for the coalescer. However, at the end of the process, we need to go back to standard code, i.e., write the final copies in some sequential order.

In most cases, a simple order of copies can be found, but sometimes more copies are needed (more precisely, one for each cyclic permutation, with no duplication) into one additional variable. Conceptually, the technique is simple but it is more tricky to derive a fast implementation. We designed a fast sequentialization algorithm that requires the minimum number of copies. We realized afterward that a similar algorithm has already been proposed by C. May [96].

Nevertheless, for completeness, we give here a detailed description of the algorithm as well as the complete pseudo-code (Algorithm 23).

Consider the directed graph G whose vertices are the variables involved in the parallel copy and with an edge from a to b whenever there is a copy from a to b (we write $a \mapsto b$). This graph has the following key property: each vertex has a unique incoming edge, the copy that defines it (a parallel copy $(b \mapsto a, c \mapsto a)$ is possible but only if $V(b) = V(c)$ in which case one of the copies can be removed). Thus, G has a particular structure: each connected component is a circuit (possibly reduced to one vertex) and each vertex of the circuit can be the root of a directed tree. The copies of the tree edges can be sequentialized starting from the leaves, copying a variable to its successors before overwriting it with its final value. Once these tree copies are scheduled, it remains to consider the circuit copies. If at least one vertex of the circuit was the root of a tree, it has already been copied somewhere, otherwise, we copy one of the circuit vertices into a new variable. Then, the copies of the circuit can be sequentialized, starting with the copy into this "saved" vertex and back along the circuit edges. The last copy is done by moving the saved value in the right variable. Thus, we generate the same number of copies as expressed by the parallel copy, except possibly one additional copy for each circuit with no tree edge, i.e., no duplication of variable. For example, for the parallel copy $(a \mapsto b, b \mapsto c, c \mapsto a, c \mapsto d)$, there is one circuit (a, b, c) and an edge from c to d, so we generate the copies $d = c$, $c = a$, $a = b$, and $b = d$ (and not $b = c$).

Algorithm 23 emulates a traversal of G (without building it), allowing to overwrite a variable as soon as it is saved in some other variable.

When a variable a is copied in a variable b, the algorithm remembers b as the last location where the initial value of a is available. This information is stored into `loc`(a). The initial value that must be copied into b is stored in `pred`(b). The initialization consists in identifying the variables whose values are not needed (tree leaves), which are stored in the list `ready`. The list `to_do` contains the destination of all copies to be treated. Copies are first treated by considering leaves (while loop on the list `ready`). Then, the `to_do` list is considered, ignoring copies that have already been treated, possibly breaking a circuit with no duplication, thanks to an extra copy into the fresh variable n.

3.5 Further Readings

SSA destruction was first addressed by Cytron et al. [54] who propose to simply replace each ϕ-function by copies in the predecessor basic-block. Although this naive translation seems, at first sight, correct, Briggs et al. [34] pointed subtle errors due to parallel copies and/or critical edges in the control flow graph. Two typical situations are identified, namely the "lost copy problem" and the "swap problem". The first solution, both simple and correct, was proposed by Sreedhar et al. [130]. They

address the associated problem of coalescing and describe three solutions. The first one, consists in three steps: a) translate SSA into CSSA, by isolating ϕ-functions; b) eliminate redundant copies; c) eliminate ϕ-functions and leave CSSA. The third solution that turns out to be nothing else than the first solution that would virtualizes the isolation of ϕ-functions shows to introduce less copies. The reason for that, identified by Boissinot et al., is due to the fact that in the presence of many copies the code contains many intersecting variables that do not actually interfere. Boissinot et al. [21] revisited Sreedhar et al.'s approach in the light of this remark and proposed the value based interference described in this chapter.

The ultimate notion of interference was discussed by Chaitin et al. [42] in the context of register allocation. They proposed a simple conservative test: *two variables interfere if one is live at a definition point of the other and this definition is not a copy between the two variables*. This interference notion is the most commonly used, see for example how the interference graph is computed in [10]. Still they noticed that, with this conservative interference definition, after coalescing some variables the interference graph has to be updated or rebuilt. A counting mechanism to update the interference graph was proposed, but it was considered to be too space consuming. Recomputing it from time to time was preferred [42; 41].

The value based technique described here can also obviously be used in the context of register allocation even if the code is not under SSA form. The notion of value may be approximated using data-flow analysis on specific lattices [5] and under SSA form simple global value numbering [123] can be used.

Leung and George [89] addressed SSA destruction for machine code. Register renaming constraints, such as calling conventions or dedicated registers, are treated with pinned variables. Simple data-flow analysis scheme is used to place repairing copies. By revisiting this approach to address the coalescing of copies Rastello et al. [118] pointed out and fixed a few errors present in the original algorithm. While being very efficient in minimizing the introduced copies, this algorithm is quite complicated to implement and not suited for just in time compilation.

The first technique to address speed and memory footprint was proposed by Budimlić et al. [39]. It proposes the de-coalescing technique, revisited in this chapter, that exploits the underlying tree structure of dominance relation between variables of the same merged-set.

Last, this chapter describes a fast sequentialization algorithm that requires the minimum number of copies. A similar algorithm has already been proposed by C. May [96].

```
cur_idom = ⊥;
foreach variable v of the merged-set in DFS pre-order of the dominance tree do
    /* Finds and set the immediate dominator of v                    */
    u ← cur_idom;
    while (u ≠ ⊥) ⋀ (¬(u dominates v) ∨ uncolored(u)) do
        u ← u.idom;
    end
    v.idom ← u;
    cur_idom ← v;
    /* Walk up variables that have the same value than u             */
    /* Do it until v is uncolored or no more interference with
       v                                                             */
    v.eanc ← ⊥;
    cur_anc ← u;
    while cur_anc ≠ ⊥ do
        /* Find the first one that intersects v with the same
           color than u                                              */
        while cur_anc ≠ ⊥ ⋀ ¬(colored(cur_anc) ∧ intersect(cur_anc, v)) do
            cur_anc ← cur_anc.eanc;
        end
        if cur_anc ≠ ⊥ then
            if V(cur_anc) = V(v) then
                v.eanc ← cur_anc;
                break ;
            else
                /* cur_anc and v interfere                           */
                if preferable to uncolor v then
                    uncolor v;
                    break;
                else
                    uncolor cur_anc;
                    cur_anc ← cur_anc.eanc;
                end
            end
        end
    end
end
```

Algorithm 19: De-coalescing of a merged-set

```
foreach c ∈ COLORS do c.cur_idom = ⊥; c.curphi = ⊥
foreach basic-block L in CFG in DFS pre-order of the dominance tree do
    traverses all variables defined by an operation of L;
    traverses all virtual variables defined by a φ-function of a successor of L;
end
```

Algorithm 20: De-coalescing with virtualization of ϕ-related copies

```
foreach Operation Op in L in topological order (including φ-functions) do
    foreach variable v defined by Op do
        if ¬colored(v) then continue else c ← color(v)
        /* Finds and set the immediate dominator of v          */
        u ← c.cur_idom;
        while (u ≠ ⊥) ⋀ (¬(u dominates v) ∨ ¬colored(u)) do
            u ← u.idom;
        end
        v.idom ← u; c.cur_idom ← v;
        /* Walk up variables that have the same value than u  */
        v.eanc ← ⊥; cur_anc ← u;
        while cur_anc ≠ ⊥ do
            /* Find the first one that intersects v with the same
               color than u                                     */
            while cur_anc ≠ ⊥ ⋀ ¬(colored(cur_anc) ∧ intersect(cur_anc, v)) do
                cur_anc ← cur_anc.eanc;
            end
            if cur_anc ≠ ⊥ then
                if V(cur_anc) = V(v) then
                    v.eanc ← cur_anc;
                    break ;
                else
                    /* cur_anc and v interfere                  */
                    if preferable to uncolor v then
                        uncolor atomic-merged-set(v);
                        break;
                    else
                        uncolor atomic-merged-set(cur_anc);
                        cur_anc ← cur_anc.eanc;
                    end
                end
            end
        end
    end
end
```

Algorithm 21: Traverses all variables defined by an operation of L

```
foreach basic-block L' successor of L do
    foreach Operation Phi : "L' : a_0 = φ(..., L : a', ...)" in L' do
        if ¬colored(Phi) then continue else c ← color(Phi)
        /* Finds and set the previous virtual variable of color
           c                                                        */
        Phi' ← c.curphi;
        if Phi' ≠ ⊥ then
            if L is a predecessor of the basic-block where Phi' is defined then
                cur_pred ← operand_comming_from_L(Phi');
            else
                c.curphi ← ⊥;
                Phi' ← ⊥
            end
        end
        if Phi' ≠ ⊥ ∧ V(cur_pred) ≠ V(a') then
            /* Interference                                         */
            uncolor atomic-merged-set(Phi);
            continue;
        else
            c.curphi ← Phi
        end
        /* Finds and set the immediate dominator of the local
           virtual variable                                         */
        u ← c.cur_idom;
        while (u ≠ ⊥) ⋀ (¬(u dominates L) ∨ ¬colored(u)) do
            u ← u.idom;
        end
        v.idom ← u;
        /* Walk up variables that have the same value than u  */
        v.eanc ← ⊥; cur_anc ← u;
        while cur_anc ≠ ⊥ do
            /* Find the first one that intersects v with the same
               color than u                                         */
            while cur_anc ≠ ⊥ ⋀ ¬(colored(cur_anc) ∧ cur_anc.islive(out(L))) do
                cur_anc ← cur_anc.eanc;
            end
            if cur_anc ≠ ⊥ then
                /* cur_anc and v interfere                          */
                if preferable to uncolor Phi then
                    uncolor atomic-merged-set(Phi);
                    break;
                else
                    uncolor atomic-merged-set(cur_anc);
                    cur_anc ← cur_anc.eanc;
                end
            end
        end
    end
end
```

Algorithm 22: Traverses all virtual variables defined by a ϕ-function of a successor of L

```
Data: Set P of parallel copies of the form a ↦ b, a ≠ b, one extra fresh variable
      n
Output: List of copies in sequential order
ready ← [] ; to_do ← [] ; pred(n) ← ⊥ ;
forall (a ↦ b) ∈ P do
    loc(b) ← ⊥ ; pred(a) ← ⊥ ;                          /* initialization */
end
forall (a ↦ b) ∈ P do
    loc(a) ← a ;                                /* needed and not copied yet */
    pred(b) ← a ;                                 /* (unique) predecessor */
    to_do.push(b) ;                             /* copy into b to be done */
end
forall (a ↦ b) ∈ P do
    if loc(b) = ⊥ then ready.push(b) ;          /* b is not used and can be
    overwritten */
end
while to_do ≠ [] do
    while ready ≠ [] do
        b ← ready.pop() ;                       /* pick a free location */
        a ← pred(b) ; c ← loc(a) ;                    /* available in c */
        emit_copy(c ↦ b) ;                      /* generate the copy */
        loc(a) ← b ;                            /* now, available in b */
        if a = c and pred(a) ≠ ⊥ then ready.push(a) ; /* just copied, can
        be overwritten */
    end
    b ← to_do.pop() ;                           /* look for remaining copy */
    if b = loc(pred(b)) then
        emit_copy(b ↦ n) ;                      /* break circuit with copy */
        loc(b) ← n ;                            /* now, available in n */
        ready.push(b) ;                         /* b can be overwritten */
    end
end
```

Algorithm 23: Parallel copy sequentialization algorithm

4

Tree-Scan Coalescing: As Simple as Linear-Scan and even Better than Graph-Coloring

4.1 Introduction

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

An important detail of the register assignment process is register constraints imposed by the Instruction Set Architecture (ISA) or the Application Binary Interface (ABI). For example, the first integer argument of a function call on the ARM Linux ABI must be passed in register R_0. Similarly the division in IA32 requires the source/destination operand to reside in register %eax and %edx. Instruction sets may also impose two operands of the same instruction to use the same register (two-address mode). These constraints, referred in the previous chapter as operand pinning, are local to instructions and are usually handled *prematurely* by the allocator by splitting live-ranges, i.e., by introducing copy instructions, prior to assignment. This places additional pressure on the coalescing to eliminate as many of these extra copies as possible. Moreover, coalescing is the most costly task of register allocation [35; 68] and is NP-complete (even with 3 registers) [28; 72].

This chapter proposes a new technique called *repairing* that deals with local register constraints without requiring preliminary live-ranges splitting. We emphasize that repairing is useful when certain instruction operands are restricted to a subset of registers, possibly a singleton [89; 9; 68; 142]. The idea is to *relax* register constraints during allocation and *repair* only afterward those that have been violated. This approach allows to handle operand pinning without loosing the benefits of the elegant formalisms that have made graph coloring [42], linear scan [113], and decoupled register allocation based on Static Single Assignment (SSA) form [23; 72; 38] appealing in the first place. Moreover, it saves the overhead of premature live-range splitting. Lastly, the cost of a potential repair can be integrated into a graph-coloring based register allocator, e.g., the IRC (Iterated Register Allocator [68]), through the

introduction of *antipathies* (*affinities* of negative weight) that can be handled with minor changes in the implementation.

We also present how repairing approach can be applied to a linear scan [113; 126; 137; 101; 142] or to its SSA form based improvement a *tree scan*. Those allocators use an approach that *decouples* the spilling to the coalescing phase [9]. SSA form enables the design of decoupled register allocation schemes very naturally as it provides to liveness and interferences nice properties [23; 38; 74] that guarantee the register pressure of the program to *equal* its register demand. Thus, in SSA-based register allocation, the spilling phase simply decreases the register pressure to the number of available registers K. Then, a tree scan that traverses the dominance tree can produce a register assignment in linear time *without* introducing further spilling [29].

Our repairing approach does *not* address register bank irregularities, such as aliasing [129] or register pairing; the state-of-the-art method [135] that handles those constraints in the context of a decoupled register allocation scheme tries to avoid inserting copies at every program points as in the elementary form [7] but still relies on live-range splitting. Handling aliasing constraints without excessive and preliminary live-range splitting remains an open problem, which we do not attempt to address here. Repairing is concerned with constraints that are local to individual instructions.

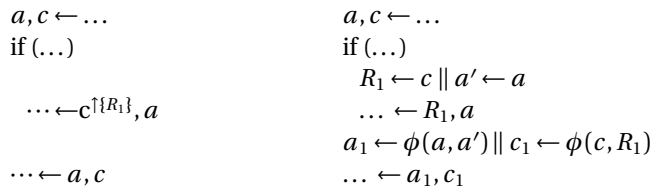
This chapter makes the following contributions:

- In Section 4.2, we extend the standard coalescing problem with *antipathies* between variables to express the fact that a variable should not be coalesced to another variable or register. Unlike *affinities* that have positive weight to express the potential gain of coalescing the corresponding variables (removal of a copy), antipathies can be seen as negative affinities that express the potential cost of assigning them to the same register (introduction of copies). While *coalescing* aims at merging as many affinity related variables as possible, *alienation* aims at making interfere as many antipathy-related variables as possible. Using affinities and antipathies, hints for register constraints can be modeled without significantly blowing up the size of the interference graph. We first show how antipathies can be modeled by interferences and (positive weight) affinities and can thus be incorporated into existing allocators by only modifying the interference graph construction phase. We then present an elegant extension to the IRC that directly handles antipathies, so avoiding the modification and size increase of the interference graph.

- In Sections 4.3 and 4.4, we show how repairing can be used in scan like allocators and describe a tree scan. We show how to minimize the number of repairing copies without the use of any graph-based coalescing. To this end, we present several *biased* heuristics for coloring.

- Related work is reported in Section 4.5.

- Section 4.6 presents an extensive experimental evaluation that shows the effectiveness of our techniques on the integer part of the Spec `CINT2000` benchmark suite. The use of repairing technique produces interference graphs that have 26% less nodes (33% less edges) compared to the state-of-the art solution with preliminary live-range splitting. Using antipathies and afterward repairing does not change the quality in terms of run-time of the compiled



(a) Initial code (b) Code with live-range splitting

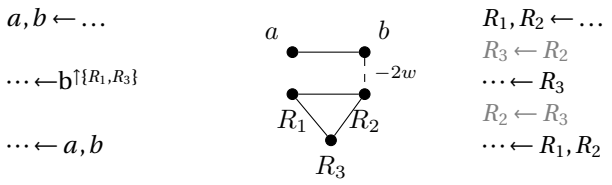
Figure 4.1: Effects of live-range splitting. $R_1 \leftarrow c \parallel a' \leftarrow a$ stands parallel copies where $R_1 \leftarrow c$ and $a' \leftarrow a$ are done in parallel. $c^{\uparrow\{R_1\}}$ indicates that operand c has to be in the related subset of registers, here $\{R_1\}$.

program. The base line tree scan algorithm produces code of the same quality as the IRC while showing an allocation time speedup of 8.81x. Activating biasing techniques outperforms the run time performance of the best IRC configuration while the allocation time speedup compared to IRC is still 6.43x. These good results also carry against the recent preference guided scan allocator from Braun et al. [33] where our algorithm is 4.72x faster for a similar run-time quality.

Finally, Section 4.7 concludes the chapter.

4.2 Graph Coloring with Repairing

Many compilers use an interference graph to guide register allocation; to save space, we assume that the reader is familiar with interference graphs and their related concepts such as liveness and affinities between copy-related variables (see [8] for example). In principle, any graph coloring register allocator can be modified to handle register constraints through the introduction of pre-colored vertices [68]. Any variable that should be assigned to register R is initially merged with the pre-colored vertex R. Any variable which assignment is constrained to a register subset is made interfering with any register not part of the subset. The fulfillment of operand constraints might require splitting live-ranges by inserting copies. Indeed, a given variable may appear in two operands which constraints are incompatible. Also, constraining at least two vertices to be assigned to some given colors can make a graph initially colorable not colorable anymore, thus causing additional spilling. To limit the lifetime of constrained variables, the allocator usually splits, prior to coloring, live-ranges by inserting copies around [101], or at least (for SSA code) just before [72] each constrained instruction. In general, this can reduce the amount of additional spilling, and for SSA form programs it guarantees the register pressure to equal its register demand. As illustrated in Figure 4.1 live-range splitting can be done through the use of *parallel copies* that correspond to set of copies to be executed simultaneously.



(a) Initial code (b) Interference graph (c) Allocated code

Figure 4.2: If a and b are respectively allocated to R_1 and R_2 some repairing code (in gray) is inserted. An antipathy (dashed lines) of weight $-2w$ is used to model this cost in the interference graph.

4.2.1 Model and restrictions

Register constraints have different variants. Commonly several registers are charged with a special meaning throughout the program such as the stack or frame pointer. Hence, they are usually not subject to register allocation and excluded from the set of available registers. In this chapter , we consider a more local constraint were an instruction dictates that *an operand* has to be in a specific (subset of) register(s), e.g., a register class. Such constraints often occur in calling conventions of the ABI. Each argument to a function call has to be put into a dedicated register. Figure 4.2 illustrates how this constraint is modeled using antipathies. In Figure 4.2a, $b^{\uparrow\{R_1,R_3\}}$ states that the corresponding operand that uses b is constrained to be in the register subset $\{R_1, R_3\}$. As in Chapter 3, we say [89; 118] that operand b is *pinned* to $\{R_1, R_3\}$. If, for some reason, b is assigned to R_2 then some shuffle code has to be inserted prior to (and after) the pinned operation to copy b to (and respectively from) either R_1 or R_3, as shown in Figure 4.2c.

As shown in Figure 4.2b, an *antipathy* of weight $-2w$ between b and R_2, where w stands for the weight of a copy instruction, indicates that assigning b to R_2 will require at least two repairing copies around the pinned operation. For coalescing, *affinities* that express the *benefit* of assigning two variables together are represented in the interference graph using dashed lines of positive weight. Similarly, antipathies that express the *repairing cost* of assigning two variables together are also represented using dashed lines but of negative weight. We say that an *affinity is satisfied* by a coloring if the two corresponding are given the same color (coalesced). Similarly, we say that an *antipathy is satisfied* by a coloring if the two corresponding nodes are given two different colors (made interfering).

4.2.2 Strategies

We have integrated support for antipathies into the IRC, a graph-coloring based register allocator by George and Appel [68]. The original IRC implementation performs spilling and coalescing together (see Figure 4.3); as our compiler uses a decoupled approach, and a different spilling algorithm, we focus on the coalescing part. In other words, the *potential spill*, *select*, and *actual spill* can be ignored at this stage of the discussion.

The IRC algorithm iteratively transforms the graph by merging (*coalescing*) some affinity related nodes. It also removes nodes of low degree (i.e., of degree smaller

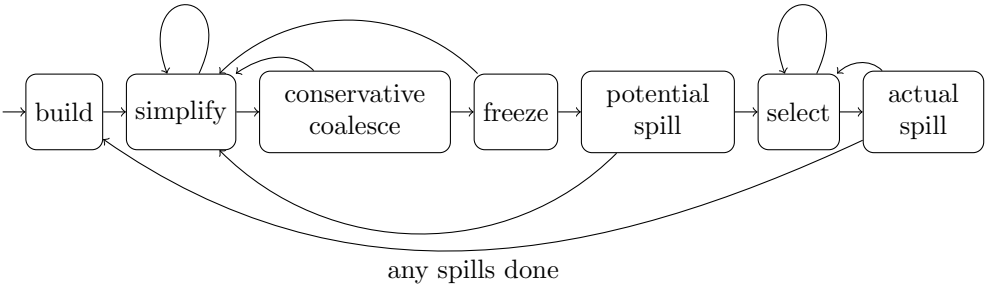


Figure 4.3: Iterated register coalescer from Figure 5 in George and Appel [68].

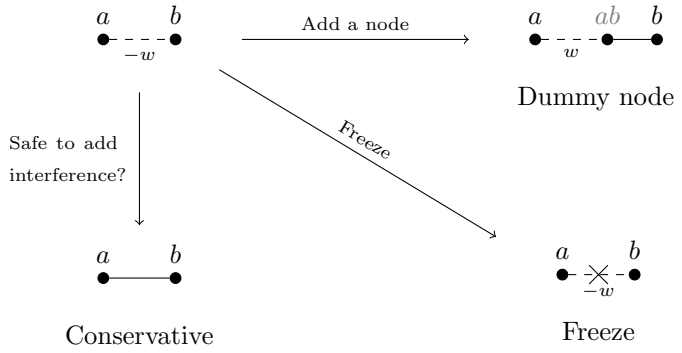


Figure 4.4: Strategies to deal with antipathies.

than the number of available registers) that are not affinity related (*simplification*). Every simplified node is pushed onto a stack. This is the coalescing-simplification phase. When all nodes are simplified it pops nodes from the stack and assigns a color. This is the color phase. The coalescing process uses an ordered (by decreasing weight) work-list of affinities (`worklistMoves` in [68]). For each affinity the algorithm checks by simple rules (namely Brigg's & George's) if both ends of the affinity can be coalesced conservatively (regarding the graph colorability). If it can, it merges the nodes, otherwise, put the affinity in some other lists. Optimistically, a judicious choice of color still has the possibility to satisfy some or all of the non-coalesced affinities when it is later popped from the stack and assigned a color; this is called *biased coloring*, as discussed by Briggs et al. [36].

Our goal is to handle antipathies within this algorithm. As the notation (dashed lines) suggests, one may want to consider antipathies as affinities of negative weight. This allows the following formalism:

Definition 4.1 (Optimal coloring)**.** Consider an interference graph $G = (V, E)$ and a weighted function that associates to each couple $(x, y) \in V \times V$ a number $w(x, y)$ (positive for affinities, negative for antipathies, null for others). A k-coloring associates to each vertex $x \in V$ a integer (color) $\mathsf{col}(x) \in [1, \ldots, k]$ such that for each $(x, y) \in E$, $\mathsf{col}(x) \neq \mathsf{col}(y)$. The weight $w(\mathsf{col})$ of a k-coloring col is the sum over each $(x, y) \in V \times V$ such that $\mathsf{col}(x) = \mathsf{col}(y)$ of $w(x, y)$. A k-coloring is said to be optimal if there is no other k-coloring with bigger weight.

This formalism imposes to have at most one affinity per pair of nodes. Thus affinities and antipathies have to be summed during the build phase of the interference graph. This is however always a good idea to merge affinities and antipathies between nodes as coloring algorithms that aim at maximizing the overall weight are heuristics. Notice also that this formalism allows an asymmetry for the function w. In theory one can choose to set $w(x,y) = w(y,x) = \omega$ for each affinity of weight ω between x and y or set (for example) $w(x,y) = \omega$ and $w(y,x) = 0$. One should just be coherent in his choice.

Using the IRC described above, we propose three different strategies to address our generalized optimization problem.

4.2.2.1 Freeze

Representing antipathies by affinities of negative weight, and letting the IRC cope with it is definitely a bad idea: Even if the weight of an affinity is negative, it will try to satisfy it, in other words merge the two corresponding nodes. Given a graph with affinities of negative weight, the simplest solution to avoid this behavior is to ignore them during the simplification-coalescing phase. This is done by initially *freezing* all negative affinities, i.e., by putting them in the `frozenMoves` work-list of [68]. The biased coloring approach of the color phase is modified to take the antipathies into account.

4.2.2.2 Dummy Nodes

The second technique consists in transforming a graph with antipathies into an equivalent graph with only (positive) affinities. Every antipathy (x,y) of weight $-w$ is replaced by a sequence of an interference edge (x,xy), with a new vertex xy called a *dummy node*, which does not correspond to an actual variable in the program, and a (positive) affinity (xy,y) of weight w. Any existing graph coloring algorithm can directly assign color for the resulting graph. Any optimal coloring of this new graph will provide an optimal coloring of the original graph.

Definition 4.2 (Graph with dummy nodes)**.** Consider an interference graph $G = (V,E)$ and a weighted function w. The corresponding graph with dummy nodes $G' = (V',E')$ and its corresponding weighted function w' is defined and built as follow: (1) for each $x \in V$ create a vertex x in V'; (2) for each $(x,y) \in E$, create an edge in E'; (3) for each $(x,y) \in V \times V$ such that $w(x,y) > 0$ set $w'(x,y) = w(x,y)$; (4) for each couple $(x,y) \in V \times V$ such that $w(x,y) < 0$, create a node xy in V', an edge (x,xy) in E', and set $w'(xy,y) = -w(x,y)$; (5) for all remaining couples $(x,y) \in V' \times V'$ set $w'(x,y) = 0$.

Theorem 4.3 (Equivalence with Dummy Nodes)**.** *Let $k \geq 2$. Consider an interference graph $G = (V,E)$ and a weighted function w. Consider its corresponding graph with dummy nodes $G' = (V \bigcup D, E')$, with w' its weighted function, and D the dummy nodes.*
(1) if there exists a k-coloring for G, then there also exists a k-coloring for G';
(2) let col be an optimal k-coloring for G', then the restriction of col to V is an optimal k-coloring for G.

Proof. (1) Consider a k-coloring of G with $k \geq 2$. For each dummy node xy of D interfering with x, set $\mathsf{col}(xy)$ to any color different than $\mathsf{col}(x)$. Such a color exists as $k \geq 2$. This provides a k-coloring for G'.

If we force $\mathsf{col}(xy)$ to be equal to $\mathsf{col}(y)$ when possible, i.e., when $\mathsf{col}(y) \neq \mathsf{col}(x)$, then we have

$$\begin{aligned} w(\mathsf{col}) &= \sum_{\substack{(x,y)\in V\times V \\ w(x,y)>0 \\ \mathsf{col}(x)=\mathsf{col}(y)}} w(x,y) + \sum_{\substack{(x,y)\in V\times V \\ w(x,y)<0}} w(x,y) - \sum_{\substack{(x,y)\in V\times V, \\ w(x,y)<0, \\ \mathsf{col}(x)\neq\mathsf{col}(y)}} w(x,y) \\ &= \sum_{\substack{(x,y)\in V\times V \\ w(x,y)>0 \\ \mathsf{col}(x)=\mathsf{col}(y)}} w'(x,y) + \sum_{\substack{(x,y)\in V\times V \\ w(x,y)<0}} w(x,y) + \sum_{\substack{(x,y)\in V\times V, \\ w(x,y)<0, \\ \mathsf{col}(xy)=\mathsf{col}(y)}} w'(xy,y) \\ &= \sum_{\substack{(x,y)\in V\times V \\ \mathsf{col}(x)=\mathsf{col}(y)}} w'(x,y) + \sum_{\substack{(x,y)\in V\times V \\ w(x,y)<0}} w(x,y) + \sum_{\substack{(xy,y)\in D\times V, \\ \mathsf{col}(xy)=\mathsf{col}(y)}} w'(xy,y) \end{aligned}$$

In other words, by letting

$$W^- = \sum_{\substack{(x,y)\in V\times V \\ w(x,y)<0}} w(x,y)$$

we get

$$w(\mathsf{col}) = w'(\mathsf{col}) + W^- \qquad (4.1)$$

(2) Consider an optimal k-coloring col of G'. First, the restriction of col to V provides a k-coloring of G. Indeed, given $(x,y)\in E$, by step (2) in the construction of G', $(x,y)\in E'$, so $\mathsf{col}(x)\neq\mathsf{col}(y)$.
Now, let us prove that for each $xy\in D$, we have $\mathsf{col}(xy)=\mathsf{col}(y)$ if and only if $\mathsf{col}(x)\neq\mathsf{col}(y)$. Indeed (by contraposition), if $\mathsf{col}(x)=\mathsf{col}(y)$, as $\mathsf{col}(xy)\neq\mathsf{col}(x)$ (xy interferes with x), this implies $\mathsf{col}(xy)\neq\mathsf{col}(y)$. Reciprocally, if $\mathsf{col}(x)\neq\mathsf{col}(y)$, $\mathsf{col}(xy)$ can be set to $\mathsf{col}(y)$ which satisfies the affinity between xy and y, and then provides a strictly better solution than if by absurd $\mathsf{col}(xy)\neq\mathsf{col}(y)$.
As equation 4.1 holds, this proves that if $w'(\mathsf{col})$ is maximal for G', $w(\mathsf{col})$ is maximal for G. □

4.2.2.3 Conservative Alienation

The basic idea of this third technique is to *conservatively* replace an antipathy (x,y) with an interference edge, when doing so does not affect the colorability of the interference graph. Recall that the work-list of affinities is sorted using their weight. Our first modification consists in putting both antipathies and affinities in this work-list and considering the absolute value of the weights in the way they are sorted. Whenever a (positive) affinity is popped from the work-list, the code is unchanged: The conservative coalescing tests [30] are performed and if successful the two corresponding nodes are merged. When an antipathy is popped from the work-list, the test consists in checking instead if the antipathy can by conservatively (regarding the graph colorability) replaced by an interference. If the test is successful the interference is actually added, the degrees of the corresponding nodes updated, and their position in the many work-lists handled by IRC updated also. The rule can be stated as follow:

Definition 4.4 (Conservative Alienation)**.** *let k be the number of available registers. Let (u,v) be an antipathy; (u,v) can be replaced with an interference edge if, u (or v) has at most $k-2$ neighbors of high degree i.e., of degree at least k.*

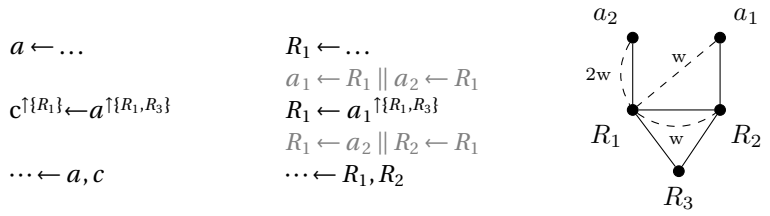
This rule is conservative regarding the greedy-k-colorability [28] of the graph. A graph is said to be greedy-k-colorable if it can be reduced to an empty graph by successively eliminating (simplification process mentioned above) low degree nodes (degree less than k).

Theorem 4.5 (Preservation of greedy-k-colorability)**.** *The* conservative interfering rule *preserves the greedy-k-colorability. In other words, consider a greedy-k-colorable interference graph $G = (V, E)$. Consider two nodes u and v in this graph such that u has at most $k-2$ high degree neighbors. Then the graph $G' = (V, E \bigcup \{(u, v)\})$ is greedy-k-colorable.*

Proof. Clearly a sub-graph of a greedy-k-colorable graph is also greedy-k-colorable: Any elimination order that fully reduces a graph can also be used to fully reduce any sub-graph, as nodes on the sub-graph have a lower degree than in the initial graph. Suppose u has at most $k-2$ high degree neighbors. Adding an interference between u and v does not change the degree of nodes other than u and v. All originally low degree neighbors of u (excluding v) can still be eliminated. Remains at most $k-1$ neighbors (including v), so u itself can then be eliminated. The obtained graph is a sub-graph of the initial interference graph. This proves that the introduction of such an interference does not change the greedy-k-colorability of the graph. □

4.2.3 Repairing Code

When coloring is over, repairing code has to be inserted for each actual antipathies that have not been satisfied, i.e., whenever two antipathy-related nodes have been assigned the same register. Repairing can be understood as an allocation problem restricted to a very small region around the pinned operation. Consider the example of Figure 4.5a. Suppose that, despite the affinity of c with R_1 and the antipathy of a with R_1 (as a is live-through), c and a have been assigned respectively R_2 and R_1. To repair the inconsistencies, every variable live-in of the pinned operation (a here) is copied to a new local variable (a_1 here). Any use in that operation is replaced by the corresponding freshly created variable; hence the use of a is replaced by a use of a_1. If, as a, a live-in variable is both used in the operation and live-out of the operation then it is duplicated, i.e., copied to another new local variable (here a_2): This duplication will be the one that will traverse the pinned operation. Note that a_1 and a_2 are not made interfering here. Every defined variable (here c) is also replaced by a new local variable; in our example, as for any variable whose constrained subset is a singleton, we directly replaced this new local variable by the only possible register it has to be allocated to, i.e., R_1. Now, for every variable live-out of the pinned operation (here c allocated to R_1, and a carried by a_2) a copy back from the corresponding new local variable is inserted just after the pinned operation. In our example, R_1 (that carries the definition of c) is copied to c (allocated to R_2), and a_2 is copied back to a (allocated to R_1). This leads to the code of Figure 4.5b where assigned variables have been replaced by registers, and where the freshly created local variables remain to be allocated. We end up with a classical allocation problem where copies are affinities to be satisfied and interferences link variables that cannot share the same register. The corresponding interference graph is represented in Figure 4.5c. Affinities between interfering nodes that could obviously not be satisfied have been represented for completeness. a_1 and a_2 respectively assigned R_1 and R_2 would lead to a final code with a copy $R_2 \leftarrow R_1$ before the operation and a swap of R_1 and R_2 after. In practice, the allocation problem being very local, the interference



(a) Initial code (b) Allocated code (c) Interference graph

Figure 4.5: a, c have been assigned R_1, R_2. Some parallel copies are introduced to repair the inconsistency. The new local variables a_1 and a_2 have to be allocated. The corresponding interference graph.

graph is not actually built. A greedy ad-hoc heuristic, such as the one developed in Section 4.3.2, is designed instead.

4.3 Tree Scan

In the general graph-coloring setting, the minimum number of registers required to color the graph might very well exceed the maximum register pressure of the program. Recent results on SSA-based register allocation show [23; 38; 74] that if the program is in SSA form, its register demand equals its maximum register pressure. This allows for decoupling spilling and register assignment: once the maximum register pressure in the program is lowered to the number of available registers, a *scan* algorithm manages to assign registers without causing further spills.

To this end, the tree scan algorithm traverses the dominator tree in pre-order, while processing the definitions and uses of variables in a manner similar to linear scan register allocation [113]. However, in contrast to the original linear scan algorithm, tree scan does not over-approximate the live-ranges of variables by intervals but uses precise liveness information.

Spilling techniques [32; 72] for SSA programs are not in the scope of this chapter ; we assume that spilling has already been performed and the register pressure is nowhere larger than the number of registers.

4.3.1 The Basic Algorithm

The CFG is processed in reverse post order (in general any dominance-preserving order works). Each basic block is traversed from top to bottom. A bit set of occupied registers is maintained. At the entry of a basic block this should be set to the registers used by variables that are live-in. However, SSA form allows to avoid the cost of pre-computing liveness sets in favor of the fast liveness check described in Chapter 2. The reason why liveness sets can be avoided under SSA is that, a variable live-in of a block is also live-out of its already processed immediately dominating block: the scan algorithm can reuse the occupancy set of the end of the immediate dominator block, tests which of those variables are live-in, and release unused registers accordingly. During the scan of a basic block, whenever a definition of a variable is encountered, it is assigned the next free register. Whenever a *death point*

of a variable is encountered (the variable is no more live after this program point), the corresponding register is released. For this last task, fast liveness check can also be used.

Main loop (Algorithms 24 and 25) The pseudo code of the main loop is given in Algorithm 24 and the details of the processing of a single operation is given in Algorithm 25. As register assignment is classically assimilated to graph coloring, the term colors will be used heavily in place of registers. In these algorithms, code in gray corresponds to repairing features explained in Section 4.3.2. The remaining code shows the basic algorithm that can be directly implemented as it if no repairing is involved or if repairing is done as a separate phase afterwards. In this case, the helper function CHOOSECOLOR called for each variables definition simplifies to providing the first available register.

The first task TREESCAN does when processing a basic block *block* is to initialize its set of live-in variables *block*.allocatedVariables: checking if variable v is live-in of basic block *block* is done through v.islivein(*block*). It is then updated, for each operation, *op*, by PROCESSOPERATION along with the corresponding (not reported in the pseudo-code) bit-sets of occupied and available registers. To avoid checking the set of all allocated variables, dead variables, i.e., variables not live-out of the current operation (tested through u.isliveout(*op*)), are extracted from the set of variables used by the operation (*op*.arguments). At this point ϕ-functions need a special treatment as explained below.

As every definition dominates all its uses, once an operation have been fully processed, all its operands can be replaced by the assigned registers. This is done through the call of function ASSIGNOPERANDSCOLOR which implementation subtleties related to ϕ-functions arguments are explained at the end of the next paragraph.

```
1:  procedure TREESCAN(Region region)
2:      for block in region.blocks using reverse post-order do
3:          // Initialize set of occupied registers
4:          block.allocatedVariables = if block.isEntry then ∅
                                       else block.idom.allocatedVariables
5:          block.allocatedVariables = {v ∈ block.allocatedVariables / v.islivein(block)}

6:          // Forward traversal of the operations
7:          for op in block.ops do
8:              PROCESSOPERATION(block, op)
9:              If op.next=⊥ or op.next.isLateOperation then
10:                 // Last point of the block where we can insert code
11:                 FIXGLOBALCOLOR(block, op.next)
12:         // If the late operation changes the global color, then the outgoing edges
13:         // have to be split and FIXGLOBALCOLOR called on all created blocks.
```

Algorithm 24: Tree Scan main loop. Code in gray represents repairing code.

Special treatments for ϕ-functions Even if the instruction used to represent a ϕ-function in the intermediate representation is usually placed at the beginning of a basic block, its uses should semantically be considered as being at the end of its corresponding predecessor basic blocks, or as here, on the corresponding incoming

```
Require: The set of all φ-functions of a basic-block should be encapsulated inside a
    single operation
 1: procedure ProcessOperation(BasicBlock block, Operation op)
 2:     dead = ∅
 3:     parallelCopy = []
 4:     // φ-function arguments are considered to be on the incomming edges, not
    here.
 5:     if op not is φ operation then
 6:         // Check arguments constraints and release last used colors
 7:         for u ∈ op.arguments do
 8:             // If current color does not match constraints, then repair
 9:             If u.ccolor ∉ op.constraints(u) then
10:                 success = RepairArgument(block, op, u, &parallelCopy)
11:                 If not success then
12:                     // Repairing heuristic failed. Replay all using graph coloring
13:                     GraphColoring(block, op, &parallelCopy)
14:                     goto end_of_coloring
15:             // Check whether u is last used here or not
16:             if not u.isliveout(op) then  dead = dead ∪ {u}
17:         // Release dead variables
18:         block.allocatedVariables = block.allocatedVariables \ dead

19:     // Assign definitions
20:     for d ∈ op.results do
21:         [d.gcolor, d.ccolor] = ChooseColor(block, op)
22:         If d.ccolor = ⊥ then
23:             success = RepairResult(block, op, d, &parallelCopy)
24:             If not success then
25:                 GraphColoring(block, op, &parallelCopy)
26:                 goto end_of_coloring
27:         block.allocatedVariables = block.allocatedVariables ∪ {d}

28:     label end_of_coloring:
29:     // Instanciate repairing
30:     InsertParallelCopy(block, op, parallelCopy)
31:     AssignOperandsColor(op)

32:     // Release dead definitions
33:     for d ∈ op.defs if not d.isliveout(op) do
34:         block.allocatedVariables = block.allocatedVariables \ {d}
```

Algorithm 25: Tree Scan operation processing. Code in gray represents repairing code.

edges. This explains why line 5 of Algorithm 25 filters out ϕ-functions: dead arguments (and in particular dead ϕ-arguments) are released when entering the basic block thanks to line 5 of Algorithm 24. Another subtlety related to ϕ-functions is that the set of ϕ-functions of a given basic block should be executed simultaneously. As an example, consider two ϕ-functions written in sequence in the intermediate representation of the program as follow: $a_1 = \phi(a_2, a_3)$; $b_1 = \phi(b_2, b_3)$. Suppose a_1 is not used *anywhere* in the program. The code should *not* be understood as the sequence (1) assign a_1; (2) release a_1; (3) assign b_1. But as (1) assign a_1 and b_1; (2) release a_1. For that reason, the ϕ-functions of a basic block should be treated all together: lines 21 and 34 of Algorithm 25 should iterate over *all* ϕ-definitions, a_1 and a_2 in our example. Lastly, as already mentioned, ϕ-function semantics also impacts the implementation of AssignOperandsColor: when reaching a ϕ-function, the arguments that flow from a back-edge, are not yet assigned. To avoid a special treatment of ϕ-functions arguments at the end of each basic blocks, a list of use operands (v.unassignedUses), is attached to each variable v. Those will be replaced by the assigned color as soon as the definition is processed and the variable allocated.

4.3.2 Repairing

The goal of this section is to describe how the tree scan can be extended to handle register constraints and inline the repairing process during the traversal.

Each variable is assigned one *global* color, called *gcolor*. This is the color that the variable has *across* basic blocks: the assignment at the *entry* and *exit* of each basic block must obey the global coloring. On the other hand, so as to fulfill some operand constraints *inside* a basic block, a variable can take, *locally* to that basic block, different colors than its global one. This follows the spirit of repairing advocated in the previous section: just as the repairing approach in graph coloring context allows to reduce the size of the interference graph, the repairing approach in scan context avoids the storage of each basic block boundary register assignment.

In other words, as the tree scan progresses, any allocated variable has a *current* color (called *ccolor*) that might be different than its global color. The current color of a variable can change (i.e., be different than at the immediately dominating operation) whenever a pinned operation is encountered. Note that its global color is not necessarily restored just after a constraining operation. This is done lazily instead: if live-out of the basic block, the variable can, later be allocated back to its global color when another pinned operation is encountered, or at least just before reaching the end of the basic block.

Repairing at the end of a basic block (Algorithm 24) In Algorithm 24, the repairing code inserted before a constrained operation is handled during the call to ProcessOperation. If, when reaching the end of the basic block, the current color of a variable is different than its global color, a copy is inserted to restore it by the call to FixGlobalColor (Algorithm 26). By "end of the basic block", we mean the last point where a copy can be inserted i.e., not necessarily at its really end but possibly just before an operation such as a jump (designed as a *late operation*). The repairing code of ProcessOperation (Algorithm 25) is detailed hereafter in the corresponding paragraph.

```
Require: All allocated variables at this point have a different global color.
1: procedure FixGlobalColor(BasicBlock block, Operation op)
2:     parallelCopy = []
3:     for var ∈ block.allocatedVariables do
4:         if var.ccolor ≠ var.gcolor then
5:             AddToParallelCopy(&parallelCopy, var, var.gcolor)
6:     InsertParallelCopy(block, op, parallelCopy)
```

Algorithm 26: Tree Scan fix global color process. For all variables that are not in their global color, copy them in parallel to their global color.

```
1: procedure AssignOperandsColor(Operation op)
2:     for i = 0 to op.operands.length() do
3:         v=op.operands[i].var
4:         if v.ccolor ≠ ⊥ then op.operands[i].color = v.ccolor
5:         else v.unassignedUses = v.unassignedUses ∪ (op,i)
6:     for u ∈ op.results do
7:         for (op',i) ∈ u.unassignedUses do op'.operands[i].color = u.gcolor
```

Algorithm 27: Tree Scan local assignment process.

Repairing at a constrained operation (Algorithm 25) When reaching a pinned operation, a parallel copy (parallelCopy in Algorithm 25) might have to be inserted just before the operation so as to match its register constraints. Recall that the restoring to the global color is not done just after the operation but lazily instead. The proposed heuristic that processes and fulfills constrained operands one after an other can fail in finding a coloring. Graph coloring is used as a fallback solution. Procedure GraphColoring (Algorithm 32) is detailed further in the corresponding paragraph. As the operands are processed, if repairing is required, parallelCopy and the corresponding ccolor variables attribute are updated by RepairArgument (Algorithm 29) for arguments and RepairResults (Algorithm 31) for results (both procedures are detailed further in the corresponding paragraphs). There are two situations that motivate the insertion of repairing code: (1) if a pinned argument is not already in the required register class (line 9 of Algorithm 25); (2) if the colors of a pinned result are already taken by other variables (line 22). For a variable v and an operation op, op.constraints(v) returns the register class v is restricted to on op. If no restrictions apply, the whole register class of v, v.regClass, is returned. If GraphColoring is called, repairing is done for all operands at once, parallelCopy and variables attributes ccolor and gcolor are set accordingly. During the processing of operands, parallelCopy is represented as a map that associates copies to variables. It is instantiated as an actual parallel copy and inserted just before the operation, only once all operands are processed through the call to InsertParallelCopy.

Selecting a color for a variable (Algorithm 28) Repairing affects the color choice in several ways. ChooseColor is called in three different contexts. First, at the definition point of some variable v (line 22 of Algorithm 25), both its global color and local one have to be set. Here the global color to choose must be different from the *global* colors used by interfering variables i.e., not in block.allocatedVariables.gcolor (that abusively represents the set { *var*.ccolor | *var* ∈ block.allocatedVariables and *var*.ccolor ≠ ⊥ }). However, it might be that a free global color is *locally* in use at v's definition (i.e., in block.allocatedVariables.ccolor). This happens because of

repairing: Another variable took that color to fulfill a certain constraint. The algorithm first checks if a color is both locally and globally available. Here, for a set *colors* PICK(*colors*) returns one of its elements if none empty and $\perp$ otherwise. Color biasing techniques as addressed by Section 4.4 can be applied at this point. If none of the allowed global colors are locally available, global and local assignment have to be different. This *temporary* state will be automatically restored later in the block thanks to the repairing process described further. The second situation where CHOOSECOLOR is called is during repairing e.g., when a live-in variable has to be recolored because of some local constraints. In that case, the current color is preferably set to its global color (already set at its definition point) if in allowedCColors. The last situation where CHOOSECOLOR is called is right after the graph coloring of the current operation. The global color is preferably set to its current color (set by graph coloring) if in allowedGColors.

Require: The register pressure does not exceeded the number of registers.
Ensure: Returns *ccolor* if called by repairing, *ggcolor* if called by graph coloring, *[gcolor,ccolor]* if called by the main tree-scan loop.

```
1:  function ChooseColor(BasicBlock block, Operation op, Variable var, RegisterSet allowedCColors = op.constraints(var) \ block.allocatedVariables.ccolor)
2:      AllowedGColors = var.regClass \ block.allocatedVariables.gcolor

3:      // Returns [gcolor, ccolor] (we have reached a definition point)
4:      if var.gcolor = ⊥ and var.ccolor = ⊥ then
5:          color = Pick(allowedCColors ∩ allowedGColors)
6:          if color ≠ ⊥ then return [color, color]
7:          else return [Pick(allowedGColors), Pick(allowedCColors)]

8:      // Returns the new ccolor (required for repairing)
9:      if var.gcolor ≠ ⊥ and var.ccolor ≠ ⊥ then
10:         if var.gcolor ∈ allowedCColors then return var.gcolor
11:         else return Pick(allowedCColors)

12:     // Returns gcolor (required by graph coloring that only sets ccolor)
13:     if var.gcolor = ⊥ and var.ccolor ≠ ⊥ then
14:         if var.ccolor ∈ allowedGColors then return var.ccolor
15:         else return Pick(allowedGColors)
```

Algorithm 28: Tree Scan color choice.

Repairing Arguments (Algorithm 29) REPAIRARGUMENT procedure is called whenever an operand is pinned to a register subclass fully occupied by some other variables. So as to release a color for the pinned operand, a variable (we say a pawn) has to be moved out from its place. As moving out a variable might require moving another variable, the procedure is recursive. forbidden, initialized to the empty set, is used to avoid endless loop. All the colors the variable var is allowed to take, are considered as candidates for receiving var (line 3). The one used by unconstrained variables are considered first as they will avoid recursion (line 6). For a given candidate, if the occupant (pawn) can move to another place (line 12) the process succeeds and the move is committed (lines 13-14). If it cannot, REPAIRARGUMENT is called recursively. The current color taken by var is made available for the recursively considered

pawns, but the color taken by pawn is marked forbidden so as to avoid considering it again in the recursion (line 17). If the repairing succeeds, the procedure returns true. In that case, parallelCopy contains the appropriate permutation of colors, and the current colors of all involved variables are updated accordingly. Otherwise, nothing is modified.

Note that, because of the recursion, the worst case complexity of this greedy ad-hoc heuristic is exponential in the number of pinned operands even-though a bi-partite matching (with lower worst case complexity) could probably do a better job in minimizing the amount of copies. We argue that repairing is rarely required, and that the exponential behavior (only pinned operands to more than one register impact the complexity) cannot appear at least for the architectures we are aware of.

Require: All variables live in front of the operation are in block.allocatedVariables. No color is available for var.
Ensure: Performs the repairing if possible (update parallelCopy and ccolors accordingly). Returns false otherwise.

```
1:  function RepairArgument(BasicBlock block, Operation op, Variable var, ParallelCopy& parallelCopy, RegisterSet available = allColors \ block.allocatedVariables.ccolor, RegisterSet forbidden = ∅)
2:      // Try out every possible moves
3:      allowed = op.constraints(var) \ forbidden
4:      while ccolor = ⊥ and allowed ≠ ∅ do
5:          // Not used in op ⇒ not constrained. So start trying not in op.uses first
6:          if allowed \ op.uses.ccolor ≠ ∅ then
7:              ccolor = ChooseColor(block, op, var, allowed \ op.uses.ccolor)
8:          else ccolor = ChooseColor(block, op, var, allowed)
9:          pawn = var ∈ allocatedVariables | pawn.ccolor = ccolor
10:         // Try to move out the pawn from ccolor
11:         pawnAllowed = op.constraints(pawn) ∩ (available ∪ {var.ccolor}) \ forbidden
12:         if pawnAllowed ≠ ∅ then
13:             pawnColor = ChooseColor(block, op, pawn, pawnAllowed)
14:             AddToParallelCopy(parallelCopy, pawn, pawnColor)
15:             success = true
16:         else
17:             success = RepairArgument(block, op, pawn, &parallelCopy,
                                         available ∪ {var.ccolor}, forbidden ∪ {pawn.ccolor})
18:         // Failed. Continue.
19:         if not success then
20:             allowed = allowed \ {ccolor}
21:             ccolor = ⊥

22:     // Commit if successed
23:     if ccolor ≠ ⊥ then
24:         AddToParallelCopy(parallelCopy, var, ccolor)
25:         return true
26:     return false
```

Algorithm 29: Tree Scan argument repairing process. No color is available, so we take from a "pawn" already in place, that might itself move another pawn... If success, makes the moves and recolor accordingly. If not return false.

```
1: procedure AddToParallelCopy(ParallelCopy& parallelCopy, Variable
   var, Color color)
2:     if parallelCopy[var] = ⊥ then
3:         set:  parallelCopy[var] = var.ccolor → color
4:     else
5:         replace in  parallelCopy[var]:  src → dst by src → color
6:     var.ccolor = color
```

Algorithm 30: Tree Scan parallel copy update. Parallel copy structure maps a variable to a pair of colors (source → destination).

Repairing Results (Algorithm 31) In theory repairing a result is similar to repairing an argument. However, a cascading strategy with recursion would requires a costly handling of sets of available colors depending on whether the variable to move is a last use, a definition or a live through. The proposed solution considers only the colors taken by live-through variables (designed as the pawn) as candidates for receiving var. If pawn finds an available spot (line 9), then the repairing succeeds. If not, the idea is to look for a last-use variable (designed as arg) to be swapped with pawn. To be possible, (1) as moving arg frees arg.ccolor only for the upper part of pawn's live-range (arg is a last-use), the lower part should already be free (line 14); (2) pawn should be allowed to take arg's color (line 15); (3) finally arg should be allowed to take pawn's color (line 16). If those three conditions are meet, the swap is committed (lines 18, 21), and as arg occupies only the upper part, the lower part becomes free for var that can take it without further ado (line 27).

Fallback: Graph Coloring of the Operation (Algorithm 32) The repairing process has a fall-back mechanism as soon as one of the heuristic fails to find a coloring that fulfills the constraints. These failures mostly occur when the register pressure is exceeded, which is unlikely unless the spilling phase gets it wrong, or when there is a need for duplications. As opposed to a live-range splitting that has the effect of moving a value from a resource to another, a *duplication* is a copy that lets the source variable alive. There are cases, such as for variable a in the example of Figure 4.5c, where a duplication cannot be avoided. In our register allocation scheme, such patterns are detected by the spilling phase and required duplications are inserted prior to the coloring/coalescing.

The fall-back mechanism, based on a graph coloring, corresponds to the repairing technique described in Section 4.2.3. First, every live-through variables are duplicated (lines 10-16). Then the interference graph is built. Every live-in variable should interfere with one another but for a variable with its duplicate (line 18); every variable live at the definition point should interfere with one another (line 19). Next operand constraints are expressed through interferences to non allowed colors (line 21). Affinity setting presents two subtle differences with the description of Section 4.2.3. First, as the tree scan restores the global color lazily instead of right after the pinned operation, the affinity of a live-through variable is 1 with its current color (line 23) plus 0.5 with its global color (lines 25). Second, as the global color of definitions are not set yet, antipathies with the global color of interfering variables are added (line 26). Once a coloring has been found, duplicated variables that have been assigned the same color than their respective parent can be deleted (lines 29-

```
Require:
1: function RepairResult(BasicBlock block, Operation op, Variable var,
   ParallelCopy& parallelCopy)
2:     // Trying a move among all live-through only variables
3:     allowed = op.constraints(var) \ op.defs.ccolors) \ op.uses.ccolors
4:     while allowed ≠ ∅ and ccolor = ⊥ do
5:         ccolor = ChooseColor(block, op, var, allowed)
6:         pawn = var | var.ccolor = ccolor
7:         pawnAllowed = op.constraints(pawn) \ op.defs.ccolors) \ op.uses.ccolors
8:         if pawnAllowed ≠ ∅ then
9:             // There is an available spot for pawn
10:            pawnColor = ChooseColor(()block, op, pawn, pawnAllowed)
11:        else
12:            // pawn's color could be free (for var) by swapping pawn with a last
   use
13:            for arg ∈ op.uses if not arg.isliveout(op) do
14:                if arg.ccolor ∉ block.allocatedVariables.ccolor and
15:                            arg.ccolor ∈ op.constraints(pawn) and
16:                            pawn.ccolor ∈ op.constraints(arg) then
17:                    pawnColor = arg.ccolor
18:                    AddToParallelCopy(&parallelCopy, arg, pawn.ccolor)
19:                    break

20:        if pawnColor ≠ ⊥ then
21:            AddToParallelCopy(&parallelCopy, pawn, pawnColor)
22:        else
23:            allowed = allowed \ {ccolor}
24:            ccolor = ⊥

25:    // Commit if success
26:    if ccolor ≠ ⊥ then
27:        var.ccolor = ccolor
28:        return true
29:    return false
```

Algorithm 31: Tree Scan result repairing process.

30). If not, the parallel copy could contain twice the same copy, which should be detected when sequentialized.

Our register allocation scheme is fully decoupled, meaning that no spilling is required during coloring/coalescing. However, a non fully decoupled approach using an optimistic *lightweight* spilling phase could be considered. In that case, Algorithm 32 should be able to perform spilling. Loads and stores for some live-through variables would be inserted around the current operation. So one iteration of the IRC would do the job.

4.4 Biased Coloring

The goal of coalescing/alienation is to remove as many copies as possible. Some are already present in the original code, some come from the use of SSA form (through the form of ϕ-functions), and the largest source of copies come from the accommodation of register constraints (through preliminary live-range splitting or repairing). Coalescing is a hard problem (it is already NP-complete for SSA programs without register constraints [28]) and efficient coalescing algorithms are too slow (see Section 4.6) in a context of Just-In-Time compilation.

The goal of this section is to present several heuristics to *bias* the color choice of the tree scan algorithm to give move-related variables the same color in the first place. As our experimental evaluation shows, these heuristics suffice to waive the coalescing pass completely. Hereafter, we quickly review the adoption of Mössenböck and Wimmer's register hints [142] for tree scan and then present new biasing approaches.

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

Aggressive pre-coalescing An aggressive coalescing merges as many copy and ϕ-function related variables as possible. It is easier than conservative coalescing as colorability of the resulting graph is not a concern. In particular the fast algorithm described in Chapter 3 that exploit SSA properties and do not even require the built of an interference graph can be used for this purpose. Instead of actually merging variables, our aggressive pre-coalescing phase puts as many copy and ϕ-function related variables into interference-free sets (called *equivalence classes* by Sreedhar in [130]). Classes are then used during the tree scan to bias the coloring of a variable to the "color" of the class it belongs to. The "color" of a class (initially undefined) corresponds to the global color of its last assigned variable. In other words, when assigning a color to a variable, the tree scan checks if the color of the class is available, if so, it takes it. If not, it picks a different color (based on the other heuristics presented here) and updates the class' color.

Caller-saved registers This technique tries to put variables that are live across a call site into registers that are saved by the *callee*. Thus, it tries to avoid *caller-saved*

Ensure: op is colored with respect to coloring constraints (current repairing is discarded)
Ensure: availableColors and block.allocatedVariables are updated

```
1: procedure GraphColoring(BasicBlock block, Operation op, ParallelCopy& parallelCopy)
2:     // Backtrack failed repairing
3:     for var: src → dst in parallelCopy do var.ccolor = src
4:     parallelCopy = []
5:     block.allocatedVariable = block.allocatedVariables \ op.defs
6:     for var ∈ op.defs do  var.gcolor = var.ccolor = ⊥

7:     // Build live-sets
8:     lastUses = {var ∈ op.uses | not var.isliveout(op)}
9:     liveThrough = block.allocatedVariables \ lastUses

10:    // Duplicate variables that are both used and live-through
11:    duplicates = []
12:    for i in op.arguments.indices if op.arguments[i].var ∈ liveThrough do
13:        dup = Duplicate(op.arguments[i].var)
14:        duplicates[var] = duplicates[var] ∪ {dup}
15:        op.arguments[i].var = dup
16:        lastUses = lastUses ∪ {dup}
17:        dup.ccolor = op.arguments[i].var.ccolor

18:    // Build the interference graph and do graph coloring potentially with local spill
19:    interferenceGraph.addCliqueButForDuplicates(lastUses ∪ liveThrough)
20:    interferenceGraph.addClique(defs ∪ liveThrough)
21:    for var ∈ op.operands do
22:        interferenceGraph.addInterferences({var}, allColors \ op.constraints(var))
23:    for var ∈ lastUses ∪ liveThrough do
24:        interferenceGraph.addAffinity(var, var.ccolor, 1)
25:    for var ∈ liveThrough do
26:        interferenceGraph.addAffinity(var, var.gcolor, 0.5)
27:    interferenceGraph.addAntipathies(op.defs, block.allocatedVariables.gcolor, 0.5)
28:    coloring = interferenceGraph.color(op)

29:    // Remove useless duplicates and apply the coloring result
30:    for var ∈ liveThrough do
31:        for dup ∈ duplicates[var] if coloring[var] = coloring[dup] do
32:            Delete(coloring[dup])
33:            Delete(dup)
34:    for var: color in coloring if var.ccolor ≠ color do
```

Algorithm 32: Tree Scan fall back repairing process.

registers for these variables. The fast liveness check method used by the original tree scan algorithm is not very helpful, as the question that arises at the definition point of the variable is to know whether that variable is live across a call: In that case every call site dominated by the variable's definition should be tested. Instead, when using the caller-saved heuristics, we resort to a classic liveness analysis. If aggressive pre-coalescing is used as well, the across-a-call information is also propagated to the equivalence classes.

Round robin assignment The usual choice for a fresh register is to take the first available color, usually in the order of the bit set that tracks the registers in use. However, this paradigm usually leads to an unequal distribution of the colors used. Freed registers are immediately reused by the variable defined next. Hence, some registers are more frequently used than other ones. This has two negative effects. First this usually decreases the chance that a move-related variable can reside in the same register. Second, the allocated code contains more anti-dependences, making the job of a post-pass scheduler much harder. A round-robin strategy that affects registers in a cyclic manner aims at making a more balanced assignment.

Move related To further increase the chance for move-related variables to get "their" color (the one of their equivalence class),register file is divided into two parts (of equal size in our case but could be tuned): The first part is reserved for move-related variables and is only used by non-move-related variables if registers of the second part are exhausted. Inside the move-related part, round-robin strategy is used to assign registers.

Figure 4.6 summarizes all presented bias techniques. It shows the different allocation results for each technique on an example.

4.5 Related Work

Graph coloring and register constraints Chaitin et al. [42] showed that every graph is the interference graph of a particular program, hence proving by reduction to k-COLORABILITY the NP-completeness of register allocation. In this situation, there was no interest in properties of the graph structure. Thus, register constraints were represented as interferences.

More recently, it was shown that the interference graphs of SSA-form programs are chordal, which allows for coloring in polynomial time [23; 38; 74]. However, checking the k-colorability of a chordal graph with at least two pre-colored nodes is not polynomial anymore. Thus, early SSA-based allocators [74] used premature live-range splitting in front of constrained instructions as well. Moreover, Odaira et al. [104] show that live-range splitting implies an overhead of 20% on average in the compile time of IRC.

Scan approaches The idea of linear scan register allocation goes back to Traub et al. [137] and Poletto and Sarkar [113]. Allocation is done with a linear scan over the assembly code. Poletto and Sarkar do not take control flow into account and over-approximate the live-range of a variable by an interval on the linearized assembly code. Thus, variables might occupy a register where they are not live and might

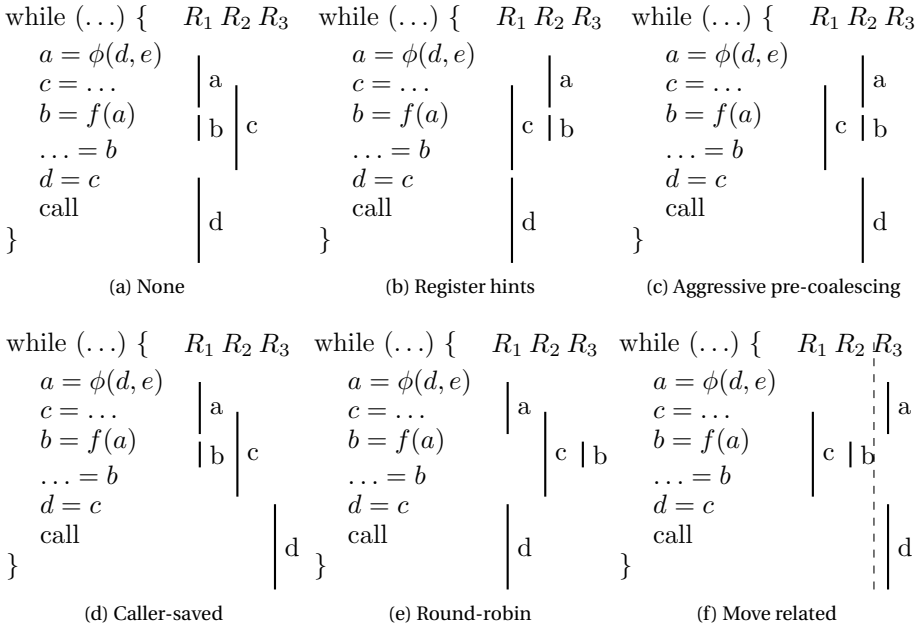


Figure 4.6: Different bias coloring strategies during tree scan. For each technique, the left part represents the source code; the right part shows the allocation of the variables with their live-range in the related column. The second argument e of the ϕ-function is supposed to be already assigned to R_2. R_1 and R_2 are caller saved registers. For the aggressive pre-coalescing strategy, equivalence classes are supposed to be $\{b\}$, $\{c\}$, and $\{a,d,e\}$. For the move-related strategy the reserved set for move-related variables is supposed to be $\{R_3\}$.

provoke unnecessary spill code. This method is simple and fast, but gives worse results than standard graph-coloring approaches. Traub et al. perform liveness analysis before and allow for holes in the intervals to avoid the over-approximation of live-ranges.

Mössenböck and Pfeiffer [101] proposed a modification of the original linear scan to work on SSA. Unlike our *tree scan,* they do not take advantage of SSA properties to allow for an optimal register assignment. Like Traub et al., their live-ranges have holes.

Mössenböck and Wimmer [142] further improved linear scan. In particular, they improved spill code placement and added on demand live-range splitting to avoid spilling in some context. In 2007, Sarkar and Barik [126] extended the linear scan. They explicitly split at basic block boundaries to avoid spilling and to handle register constraints at the cost of shuffle code. In our setting, the program is in SSA. Thus, introducing live-range splits in addition to ϕ-functions will *not* save any further spills [74]. In 2009, Rong [121] proposed the tree register allocation, which generalizes linear scan approaches. However, this algorithm needs global liveness information, in particular for the handling of pre-colored constraints. The same year, Barik in his thesis [12, Ch.6] proposed a linear scan approach that colors the basic intervals, i.e. part of the live-range that is contiguous in the linear ordering, independently. His algorithm tries to use the same color for the global interval, i.e. composed by several basic intervals, to minimize shuffle code between basic blocks. To minimize `move`cost, it builds another graph with all basic intervals and all `move`instructions, also the one expected to be inserted on edges, and uses this graph to get the preferred color of a basic interval when assigning its color. Overall, our approach is simpler as it does not require to build an additional graph for coalescing nor it requires to handle the shuffle code on the edges. Regarding coloring constraints, Barik proposed two different approaches: (1) Choose an order of the register class and assign them separately, starting with the most constrained one. (2) Do everything at the same time with a register pressure by register class. In that context, coalescing of basic intervals composing a variable may be incompatible with already chosen color, thus creating a lot of `moves`. To circumvent this bad behavior, instead of a top to bottom approach, i.e. basic interval sorted by start date, Barik defined a bucket sorted list. With our approach, the global interval is assigned a color, thus all basic intervals have the same color a priori. Then, repairing makes the proper adjustments. The frequency of these adjustments depend on the time invest in pre passes analysis. In 2010, Wimmer and Franz [141] pointed out the interest of relying on SSA to deal with liveness in linear scan. Finally, the same year, Braun et al. [33] proposed a preference guided register allocator. Like our tree scan, it works on SSA. But unlike our approach, it processes the program using a linear ordering of the basic blocks. This ordering is defined by a complete cover of the program by traces. Moreover, it has to insert shuffle code at join point if all predecessors have not been proceeded, using ϕ-functions. Regarding coloring, it uses a new bias technique, the preference sets, that gives the liked and disliked colors for each variables. Like us, their allocator repairs the register constraints on the fly but does not handle duplications, which must be set a priori. It splits all live variables when at least one of them does not match the instruction constraints and fixes the color for all split variables. It then solves a bipartite matching problem for all the new variables. Overall, the preference guided allocator is more complex than our approach.

Interestingly, already in 1999, Yang et al. [143] proposed a fast scan based register allocation than uses the CFG. This allocator presents some similarities with both

the preference guided allocator and our tree scan. Like the preference guided, it performs a two phases allocation. First, it computes some preference set as well as the last uses points[1] and second, it allocates the program. The allocation uses a reverse post-order ordering of the basic blocks, like us, and splits the CFG in single entry multiple exits regions, i.e. it deals with tree like live-ranges. Like the preference guided, at the end of each region it stores the result of the allocation. If there is a mismatch between several predecessors of one region, it inserts some shuffle code. When this process failed, it reallocates the related region. Overall, again, this is more complex than our tree scan, since we do not have to handle shuffle code between region.

Coalescings In graph-coloring register allocation, many different coalescing techniques have been developed. They fall into three categories: Aggressive, conservative, and optimistic coalescing. *Aggressive coalescing* removes as many copies as possible, regardless of the colorability of the interference graph [41]. While it removes many copies, it may also increase the register demand of the program which potentially causes spilling. Since we never want to trade a spill for a copy, aggressive coalescing has to be used with caution. *Conservative coalescing* uses conservative tests [36; 68; 23; 30] that ensure that the chromatic number of the graph is not increased, before a copy is coalesced. *Optimistic coalescing* uses aggressive coalescing and de-coalescing if the k-colorability was violated [106; 107].

Biased coloring tries to remove copies by giving the source and the target of the move the same color in the first place. Chow and Hennessy [45] rely on copy propagation to remove moves in the priority-based allocator. Briggs et al. [36] integrate biased coloring into graph-coloring allocation. Mössenböck and Wimmer [142] use "register hints" in their linear scan allocator to propagate copy information to the definition points of the variables. They gave also a technique based on register next use distances to assign caller-saved registers to local temporaries.

4.6 Experiments

The algorithms described earlier in the chapter were implemented in the back end of a production compiler developed by our industrial partner, STMicroelectronics for their commercial media processor based on the Lx architecture [64]. This static C compiler uses OPEN64 as the code generator, LAO as the register allocator, and OPEN64 for post-allocation optimization and assembly code emission. LAO can be used both in a static and dynamic compilation context. While the funding of those developments was motivated by dynamic compilation constraints, the industrial partner does not provide us with access to the dynamic compilation tool-chain. The target processor is 4-issue VLIW with 32 general-purpose registers, 8 of which are callee-saved. Compared to IA32, for example, the Lx architecture [64] has relatively few register constraints. That being said, our results show significant improvements compared to allocators that do not effectively handle register constraints; consequently, the disparity is likely to be even greater in our favor for target architectures with more constraints.

[1] Since it does not use SSA, this information is not directly available

Our experiments use a decoupled register allocation approach. The spilling algorithms used is described in [72]; the purpose of the experiments is to compare coalescing.

Our experiments use a subset of the Spec `CINT2000` benchmarks compiled using -O3 optimization level; our compiler cannot handle *eon*, which is written in C++, and *gcc*, which requires a frame pointer that our compiler does not support. To give a better idea on how the different configurations may apply to different targets, we made our experiments with three different numbers of allocatable registers: 32, 16 and 8 registers.

4.6.1 Graph Coloring and Repairing

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

- IRC: The IRC algorithm without live-range splitting, but no repairing; this algorithm is not guaranteed to find a k-coloring of the interference graph, so it is allowed to spill, when necessary.
- Split: The IRC algorithm with live-range splitting, but no repairing.
- Freeze, Conservative, and Dummy Nodes: The IRC algorithm without live-range splitting but with repairing implemented as described in Section 4.2.2.

Figure 4.7 reports the normalized execution time of the code generated by each configuration. Figure 4.8 reports the normalized number of vertices and number of edges of the interference graph for each benchmark.

Finally, Figure 4.9 reports the normalized number of dynamically executed copies for each configuration. We used frequency estimate [11] to find the number of times each basic block executed. For each copy operation occurring in basic block b, we use the weight assigned to b to estimate the number of times the copy executes. These numbers are then summed to produce a per function cost, and these costs are summed to produce a per benchmark cost. This metric is architecture agnostic, as it ignores, for example, the possibility to hide the copies by scheduling them in parallel with one another or with other operations, or to schedule them in a branch delay slot.

Due to the number of configurations, these Figures depict just geometric means. All numbers are normalized to IRC with 32 allocatable registers.

The baseline approaches are IRC and Split, denoted IRC Split in the Figures. Between those approaches, Split produces better quality code (Figures 4.7 and 4.9), but with a noticeable increase in the size of the interference graph (Figure 4.8). In its favor, Split is the only existing technique that can deal with register constraints in a decoupled register allocation context. Our goal is to identify a coalescing that achieves the code quality of Split but without increasing the size of the interference graph.

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

Dummy nodes is the naive approach to extend graph coloring to deal with negative affinities. It represents negative affinities using dummy nodes. As shown in Figures 4.7 and 4.9, this approach produces good quality code, but the dummy nodes that are added significantly increase the size of the interference graph. Although Dummy Nodes is not shown in Figure 4.8, its interference graphs are

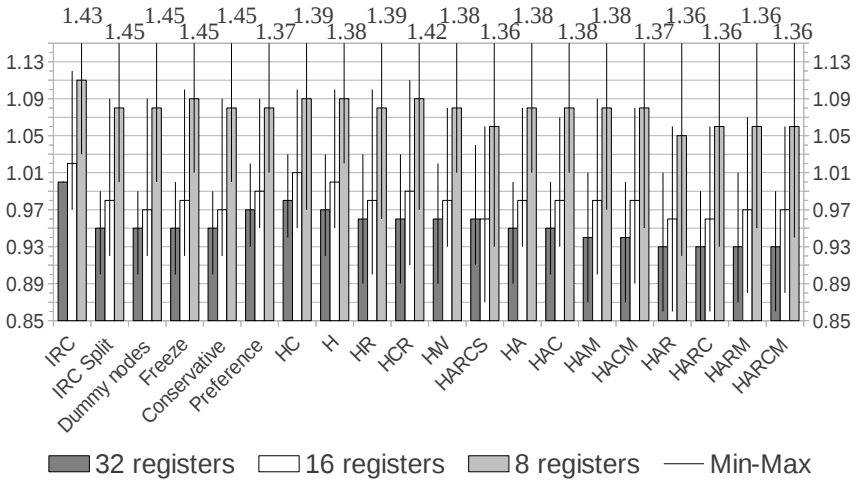


Figure 4.7: Geometric means over all benchmarks of the execution time of the generated code. Each bar represents the runtime for the given number of allocatable registers. The black lines in the middle of each bar represent the variation, i.e. minimum and maximum, over all benchmarks. All numbers are normalized to IRC with 32 registers (y=1). IRC stands for the iterated register coalescing in a decoupled fashion. IRC Split is IRC plus live-range splitting. The three next configurations are graph approaches with repairing as depicted in Section 4.2.2. Preference reports preference guided numbers. Then, letters stand for the mix of the bias coloring configurations applied to tree scan. H: Hints; R: Round-robin; C: Caller; M: Move related; A: Aggressive; W: Web; S: Split. For tree scan configurations, the results are sorted in increasing improvement with 32 registers. (Lower is better)

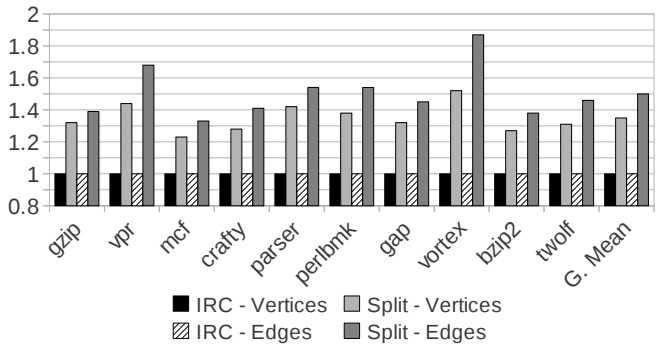


Figure 4.8: The normalized number of vertices and interference edges in the interference graph for each benchmark. For a given benchmark, the sizes of the interference graph of each function are summed. IRC, Freeze, and Conservative have the same interference graph sizes, while Split's interference graphs are noticeably larger. Dummy nodes is not a realistic solution, so we do not report its interference graph sizes, which would be large.

larger than Split in virtually all instances. For this reason, we do not consider Dummy Nodes to be a realistic approach.

Freeze only considers negative affinities during the biased coloring phase at the end of the coloring process. As shown in Figure 4.9, the quality of code generated by Freeze is inferior to that offer all other graph approaches: IRC, Split, Dummy Nodes, or Conservative. In particular, it has big worse cases (more than 28 times more moves than the IRC) when using with 32 and 16 registers. This becomes better when less choice are possible for the color of each variable, i.e. using only 8 registers, its quality competes with IRC. However, these bad performances on the number of moves barely show up in runtime numbers as reported in Figure 4.7. In that case, Freeze is slightly worse that Split but still better than IRC. Hence, inserting moves to repair is cheaper than having additional spill code. In terms of interference graph size, Freeze is comparable to IRC (Figure 4.8); the difference in size (due to a small number of negative-weighted affinity edges) is negligible, and is not shown in Figure 4.8.

Conservative converts negative affinities into interference edges using criteria similar to conservative coalescing. The quality of code generated by Conservative is comparable to Split, Dummy Nodes, and Freeze for runtime (Figure 4.7). It is one of the best regarding the dynamic number of moves (Figure 4.9), while the size of the interference graph is comparable to that of IRC, and is not shown in Figure 4.8. Among the three negative-affinity-based coalescing considered here, Conservative is the only one to achieve the code quality of Split with an interference graph size comparable to IRC.

4.6.2 Tree Scan

This section evaluates the allocation time, i.e., the compile time dedicated to register allocation, and the number of copy operations that execute dynamically when

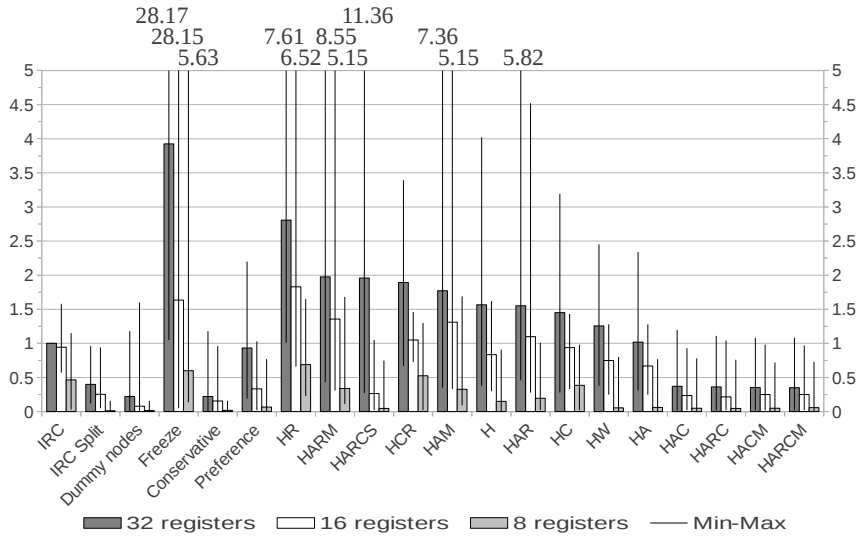


Figure 4.9: Geometric means of dynamic number of moves. See caption Figure 4.7 for the explanation of the configurations. The treescan configurations are sorted in increasing improvement with 32 registers. (Lower is better)

coalescing is performed by the tree scan algorithm with different biased color assignment techniques, as discussed in Section 4.4. As this technique is intended to be applied in a JIT context, but not restricted to, this section also reports its memory footprint. To ease the comparison with existing approaches, we included the most recent, to our knowledge, scan approach, the preference guided register allocator [33] to our results.

4.6.2.1 Allocation Time

Figure 4.10 reports the normalized compile time of the different color assignment approaches. The compile times reported include all memory allocation/deallocation, structure initialization/destruction, and liveness analysis; however, they do not include the time required to translate out of SSA, which is not part of the coloring process. For each benchmark, the runtime is the sum taken over all functions. Due to the number of configurations, only the geometric means over all benchmarks is reported.

As expected, the introduction of Register Hints to bias the color assignment process during the tree scan incurs no measurable overhead, while Round Robin color assignment incurs an overhead of 10%. Pre-coalescing comes at a higher price, 12% overhead for the Web strategy [39] and 27% for Aggressive coalescing [21]; Move Related coalescing costs an additional 11% as we use a pre-coalescing phase to know which variables are move related. The most expensive technique, however, is Caller, whose overhead is 50%; this overhead is due to the data flow analysis required to compute liveness information and a traversal of the CFG to identify variables that are live across calls. Lastly, we also report the allocation time of the tree scan with a Split strategy, where all live-ranges are split prior to constrained instructions [72]; the overhead of this technique is 71%.

Regarding the evolution of the different bias technique compile time with respect to the number of allocatable registers, we see that tree scan is more or less not impacted by this number. The slight gain with 8 registers comes from the way we chose the set of allocatable colors, here they are all callee-saved. Thus, repairing on call site never occur anymore. The same observation applies to Register Hints. For Round Robin, the compile time follows this number. Nothing surprising as it traverses this set to find the next available color. The pre-coalescing techniques depend on the program structure not the number of registers. Thus, no patterns come out. Regarding Caller, the number of call sites is not impacted by the number of registers. Thus the numbers are almost the same. The slight gain with 8 registers comes again from choice of the allocatable registers. When choosing the color for a variable having the caller flag, the operations which restrict the possible colors will never end in an empty set, thus error case is never reached. Finally, the Split strategy depends on the number of live variable, which is directly linked to the number of register in decoupled register allocation approach.

On average, the baseline tree scan runs 8.81 times faster than IRC, which respectively represents 4% and 26% of the whole back end compile time (17% for preference guided). In contrast, even the slowest-running variant of tree scan has a runtime of less than 2 times than of the baseline version. Compared to the preference guided allocator, tree scan is 4.72 times faster.

For a JIT compiler, it is clear that tree scan runs much more efficiently than register allocation based on graph coloring. Moreover, it also beats the preference guided allocator whatever the number of allocatable registers. However, the gap is smaller with few registers and tree scan is finally 2.96 times faster with 8 registers[2]). This is because the preference guided repairing process is faster when less variable are involved, whereas the tree scan is more or less linear in the number of instructions. Next, we look at the quality of the code generated by the coalescing.

Note that by adding the techniques overhead, you get the allocation time of the related composed method. For instance, caller plus web have composed overhead of $(1.5-1)+(1.12-1)=0.62$ on average. Thus, this composed method is 1.62 times slower than the baseline.

4.6.2.2 Number of Dynamically Executed Copies

Figure 4.9 reports the number of dynamically executed copy operations that result from different combinations of color assignment enhancements to the tree scan algorithm. See Section 4.6.1 to know how these numbers are computed. We consider that tree scan using register hints (H) is the most realistic baseline implementation for tree scan, due to its low runtime overhead. Thus, we did not test tree scan without any bias technique.

Let us first focus on the differences between the tree scan configurations, using register hints (H) as baseline. As the trends are the same whatever the number of registers, we comment the numbers with 32 allocatable registers. The impact of the caller heuristic (HC) in isolation is minimal: The compiler inserts less repairing code, but fewer copies are coalesced. In many cases, two move-related variables cannot be coalesced because one crosses a call and the other does not; as we will see, the caller heuristic becomes more effective when combined with better coalescing.

[2]The reported 2.82 is against the baseline tree scan with 32 registers as all numbers use the same base to normalize

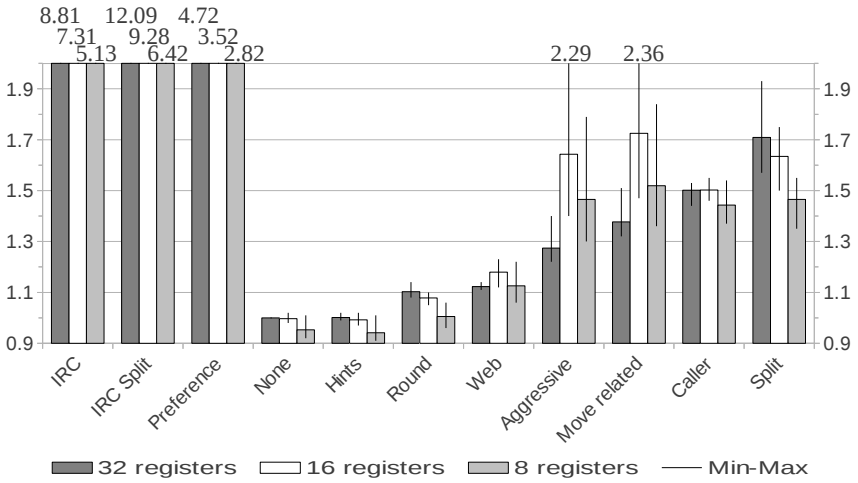


Figure 4.10: Normalized geometric means of allocation time. Numbers are normalized to tree scan baseline (None) with 32 registers. Preference reports preference guided numbers. Configurations to the right of None are tree scan algorithm with the related bias technique. (Lower is better)

Round-robin (HR) increases the number of dynamically executed copies by 79%. It does not have any information about future uses of variables, e.g.,, as operands of ϕs. Consequently, the likelihood of eliminating the copies that result during SSA destruction is quite low. Thus, the potential benefit of round-robin, is possible only with a control on how it spreads the allocation over the available colors. For instance, when combining round-robin with caller (HCR) the negative impact is reduced from 79% to 20%.

The techniques that employ pre-coalescing (HW, HA) perform quite well. Web and aggressive strategies respectively reduce the number of dynamically executed copies by 20% and 35%. When combined with round-robin (HAR) and move-related (HAM), the negative impacts observed for the round-robin strategy, as described above, manifest themselves, but in a more limited fashion, as the pre-coalescing gives a better guide for assigning registers.

Combining the pre-coalescing and caller heuristic (HAC) is beneficial, because variables that are move-related to others that cross procedure call boundaries are biased using callee-saved registers. Compared to register hints alone (H), HAC reduces the number of dynamically executed copies by 76%. Augmenting HAC with the round-robin strategy (HARC) achieves an additional percent of improvement. Similarly, combining the move-related heuristic with the caller heuristic and a pre-coalescing (HACM) achieves 78% of improvement.

Lastly, we wish to establish that pre-splitting is not necessary when using repairing; the best result achieved with pre-splitting (HARCS) increasing the number of dynamically executed copies by 25% for 32 registers. This bad result comes from the way parallel copies inserted by split are handled in tree scan. The algorithm does not have any special care for such instruction. Thus, it assigns the result in a sequential order. If the first result should not reuse the color of the first argument because of

some bias information, like the caller flag[3], the used color may not be available to coalesce one of the other result. In the worse case, this error can propagate through all results of a given parallel copy, whereas it would have been limited to few variables in the repairing case that the bias information helps to avoid. However, with less registers, this problem is less likely to occur and this configuration competes with the best configurations. Nevertheless, the impact of pre-splitting on allocation time does not justify its use in a JIT compiler.

Compared to graph based and preference guided approaches, tree scan variants perform quite well. In particular, HAC, HARC, HACM and HARCM, are better than IRC using live-range splitting and are close to the best achieved quality: Conservative repairing strategy. For 32 registers, a tree scan using only an aggressive pre-coalescing (HA) achieves results almost as good as preference guided. Thus, it competes with preference guided whereas it is 3.72 times faster according to Figure 4.10. With 16 registers, this tree scan configuration has to be combined with at least the caller (HAC) technique to catch up the gap in code quality against preference guided. In this case, it is still 1.64 times faster. Finally, with 8 registers, the web pre-coalescing technique is sufficient for tree scan to beat the preference guided. In that configuration, it is 2.5 times faster.

4.6.2.3 Run Time Performance

We compare the quality of the execution of the code generated by tree scan using the different biasing techniques. Figure 4.7 reports these results.

Due to the advantage of a fully decoupled register allocation against IRC decoupled approach, all programs compiled with tree scan are always faster than their counterparts compiled with IRC. Although tree scan is approximately nine times faster in allocation time than IRC.

The base tree scan with register hints (H) generates code that is 3% faster than IRC. More surprisingly, with the additional caller heuristic (HC), code is only 2% faster than IRC even if the number of dynamic copies is less than register hints (H). This is because of the VLIW processor we use for evaluation. When the caller heuristic is *not* active, repairing often occurs at call sites. However, at call sites there are usually enough free slots in the VLIW bundles to hide the repairing code. Hence, this repairing code comes for free. If the caller heuristic *is* active, the repairing `move` instructions occur at different places where they are no longer easy to hide because of saturated VLIW bundles.

Round-robin (HR) gives an additional percent of improvement. This benefit comes from the additional freedom for the post scheduler. On our machine, post scheduling is very important because it places moves, spills, and reloads in unused slots of near bundles.

Using a pre-coalescing approach (HW, HA), tree scan achieves 4% of improvement. This is almost as good as IRC with splitting or repairing technique. Combining these approaches with caller heuristic (HAC), tree scan gets an additional percent of improvements and is as good as the best graph coloring algorithms reported here. We achieve an additional percent by combining pre coalescing with round robin (HAR), having tree scan generated code running faster than the best graph based approach.

[3]Without other bias techniques, since split points are just a renaming of the variable, it is always possible to reuse the color of the related argument.

Surprisingly, preference guided is just slightly better than tree scan with just register hints (H), despite the fact that it has far less dynamic moves than this configuration. The reasons are twofold. Preference guided biases his color using the same metric as the one used to count the dynamic number of moves. However, this metric is based on an heuristic of frequency estimate and may not reflect the actual runtime behavior. Moreover, like for the caller heuristic, the repairing code is placed on saturated VLIW bundles, in that case, the edges.

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

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

4.6.2.4 Footprint

So far we show that tree scan approach runs faster, produces faster code with a comparable number of dynamic moves than IRC decoupled approaches and preference guided. However, to be suitable for JIT, it also must have a small memory footprint. This is what we show in that section.

Figure 4.11 reports the normalized footprint of the main classes of approaches. The numbers are normalized to tree scan baseline (None) using 32 registers. The footprint measure all memory specifically allocated to perform the related algorithm. Thus, it does not take into account the footprint of the program representation, which is the same for all but IRC with live-range splitting approach, but it does take into account, for instance, the footprint of a liveness analysis and its results if this approach needs liveness sets information. For a given benchmarks, footprint is summed over all its functions. Due to the amount of reported configurations, only geometric means over all benchmarks are reported.

As expected, IRC with live-range splitting has the biggest memory footprint as its interference graph have more variables than IRC and uses far more memory, almost 30 times more for 32 registers, than a tree scan approach. Preference guided allocator consumes also more memory than tree scan. For 32 registers, this consumption is 5.59 bigger. It decreases rapidly, to 3.46 (3.84 against None with 8 registers), according to the number of registers, whereas tree scan is not much affected by the number of registers (1 to 0.9). Preference guided is sensible to these numbers because it has to record for each variable its preference set, which size depends on the number of registers. The other difference in size comes from the fact that it needs the liveness sets to build these preference sets.

For the different bias approaches as for tree scan baseline, the number of reg-

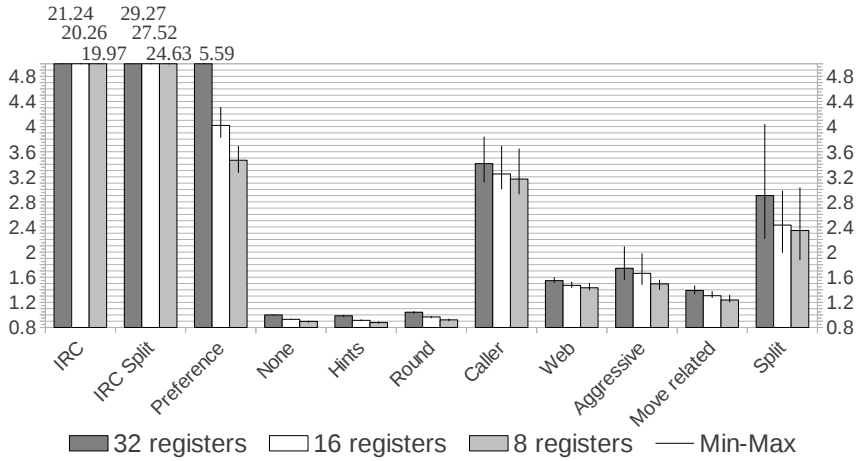


Figure 4.11: Normalized geometric means of memory footprint. Numbers are normalized to tree scan baseline (None) with 32 registers. Preference reports preference guided numbers. Configurations to the right of None are tree scan algorithm with the related bias technique. (Lower is better)

isters has a limited impact on the footprint. Caller heuristic is the most expensive bias technique, with 3.41 times the size of the base algorithm. It has to perform a liveness analysis to know which variables are crossing calls and then has to store that information. Split also has to perform that analysis but it does not need to store any additional information. As already stated, the move related technique uses a pre-coalescing heuristic to know which variable are move related. However, it uses less memory than the used coalescing technique, here aggressive, because it uses the result of the analysis, but does not have to store the color information per set nor which variables are in one set.

Like allocation time, to know the overhead of a composed bias technique, the overhead of each technique has to be added. Compared to IRC Split, tree scan has to use the HAC variant to achieve comparable coalescing quality, thus it uses 4.15 more memory than baseline. Consequently it uses still 7.05 times less memory than this technique for the same code quality. Compared to preference guided, we have to consider the number of registers. For 32 registers, HA variant is the closest. In that case, tree scan uses 3.21 times less memory than preference guided. For 16 registers, HAC is the closest cheapest variant. In that case, tree scan uses about the same amount of memory as preference guided. Finally, for 8 registers, HW is the closest cheapest variant and it uses 2.42 times less memory than preference guided.

4.7 Conclusion

This chapter has introduced repairing to handle register constraints during register coalescing. Repairing has been shown to be compatible with graph coloring-based coalescers and a new type of SSA-based coalescer called a tree scan, that does not build an interference graph and improves significantly upon past linear scan allocators. Our evaluation has shown that a graph coloring coalescer that employs re-

pairing can generate code whose quality is comparable to the most effective prior techniques that handle register constraints. The tree scan, moreover, runs more efficiently than the graph coloring-based coalescer with repairing because it does not require an interference graph, while producing code of comparable quality. Moreover, this is also true in the case of the recent scan allocator preference guided. Consequently, we believe that the most reasonable choice for JIT compilers having a decoupled register allocation is tree scan.

5

Revisiting Static Single Information

5.1 Introduction

The monotone data-flow framework is an old ally of compiler writers. Since the work of pionners like Prosser [114], Allen [4; 3], Kildall [86] and Hecht [78], data-flow analyses such as reaching definitions, available expressions and liveness analysis have made their way into the implementation of virtually every important compiler. The information acquired by data-flow analyses supports many classic compiler optimizations, such as common-subexpression and dead-code elimination, constant and copy propagation, register allocation and pointer analysis, among others. Furthermore, this framework provides a core theory grounding a profusion of developments in compiler research, both in the academia and in the industry.

The vast majority of data-flow analyses binds information to pairs formed by a variable and a program point [16; 1; 40; 56; 136; 83; 95; 102; 112; 120; 124; 127; 131; 133; 139]. For instance, for each program point p, and each integer variable v live at p, Stephenson *et al.*'s [131] bit-width analysis finds the size, in bits, of v at p. Although well studied in the literature, this approach has some drawbacks; in particular, it suffers from an excess of redundant information. For instance, a given variable v may be mapped to the same bit-width along many consecutive program points. Therefore, a natural way to reduce redundancies is to make these analyses *sparser*, increasing the granularity of the program regions that they manipulate. We identify two main design strategies to achieve this sparsity: the use of new data-structures that represent the program under analysis, or the use of new program representations which make it natural to associate information to larger code regions.

In terms of data-structures, the first, and best known method proposed to support sparse data-flow analyses is Choi *et al.*'s *Sparse Evaluation Graph* (SEG) [44]. The nodes of this graph represent program regions where information produced by the data-flow analysis might change. Choi *et al.*'s ideas have been further expanded, for example, by Johnson *et al.*'s *Quick Propagation Graphs* [83], or Ramalingan's *Compact Evaluation Graphs* [117]. Nowadays we have efficient algorithms that build such data-structures [110; 111; 82]. These data-structures improve many data-flow analyses in terms of runtime and memory consumption. Nevertheless, the elegance of SEGs and its successors have not, so far, been enough to attract the attention of mainstream compiler writers. Compilers such as gcc, LLVM or Java Hotspot rely, instead, on several types of program representations to provide support to sparse data-flow analyses.

The most famous among these representations is the Static Single Assignment form [55], which suits well forward flow analyses, such as reaching definitions. Other

representations, not as popular, yet more general than SSA form, exist too. For instance, Scott Ananian has introduced in the late nineties the *Static Single Information* (SSI) form, a program representation that supports both forward and backward analyses [6]. This representation was later discussed by Jeremy Singer [128] and revisited by Boissinot *et al.* [20]. Singer provided new algorithms plus examples of applications that benefit from the SSI form, and Boissinot *et al.* clarified a number of omissions in the related literature. A different program representation – the *Extended Static Single Assignment* (e-SSA) form – was introduced by Bodik *et al.* [16]. As opposed to SSI and SSA, the e-SSA form supports flow analyses that obtain information both from variable definitions and conditional tests. Another important representation, which supports data-flow analyses that acquire information from uses, is the *Static Single Use* form (SSU). As uses and definitions are not fully symmetric (the live-range can "traverse" a use while it cannot traverse a definition) there exists different variants of SSU (eg. [112; 69; 94]). For instance, the "strict" SSU form enforces that each definition reaches a single use, whereas SSI and other variations of SSU allow two consecutive uses of a variable on the same path. All these program representations are very effective, having seen use in a number of implementations of flow analyses; however, they only fit specific data-flow problems.

In this chapter we present a method to build program representations that support sparse data-flow analyses. We build these program representations by splitting the live ranges of variables, in such a way that the information associated to variables is invariant along their entire live ranges. Our technique is more general than the program representations that we have mentioned before. It can be parametrized according to the direction(s) of the flow problems, i.e., forward and/or backward, and according to the program points where data-flow information is produced. Usually these points contain variable definitions, uses or conditional tests. In order to build these program representations, we use an algorithm that is as powerful as the method that Singer has used to convert a program to the SSI form [128]. However, our algorithm is simpler: as we show in Section 5.3, for all unidirectional and all non-truly bidirectional data-flow analysis we can avoid iterating the live range splitting process in order to build intermediate representations.

Our method subsumes Choi *et al.*'s sparse evaluation graphs, as we demonstrate in 5.6; however, we improve on SEGS in a number of ways. Firstly, SEGs best suit a class of data-flow analyses that Zadeck defines as *Partitioned Variable Problems* [144] (PVP). Reaching definitions and liveness analysis are examples of PVPs. For these problems we can build intermediate program representations isomorphic to SEGs. However, as we explain in Section 5.2, many data-flow problems do not fit into this category; nevertheless, we can handle them sparsely. Secondly, we improve on SEGs in terms of space: this data-structure keeps - for each program variable - a mapping from SEG vertices to Control Flow Graph (CFG) edges that is linear on the size of the CFG. We do not keep this map. Instead, we can replace it with the fast liveness check described in Chapter 2. Thus, whenever necessary we can map the information related to a variable to the program points where this variable is live.

We have implemented our framework on top of the LLVM compiler [88], and have used it to provide intermediate representations to two well known compiler optimizations: Wegman *et al.*'s [139] conditional constant propagation, and Bodik *et al.*'s [16] algorithm for array bounds check elimination. We have also built the SSI form as defined by Singer, and compare it with the other program representations that we produce. The intermediate program representations that we derive from our framework increase the size of the original program by less than 5%. This is

one order of magnitude less than Singer's SSI form. Furthermore, our experiments indicate that the time to build these program representations is less than 2% of the time taken by the standard suite of optimizations used in the LLVM compiler.

5.2 Sparse Data-flow Analyses

In this section we quickly review some concepts related to flow analyses. For a more in depth overview of this topic we recommend Nielson *et al.* [103]. The monotone data-flow framework associates *information* with *program points*. We define a *Program Point* as any minimum region in the program code where a data-flow analysis can acquire information. The algorithms that we describe in this chapter consider as program points the instructions in the source code, and the regions between consecutive instructions. A *transfer function* determines how information flows between adjacent program points. This information is an element of an algebraic body called a *lattice*. For instance, liveness analysis is a flow problem in which the challenge is to determine which variables are *live*[1] in and out of each CFG node. The regions of interest, in this case, are *program points between instructions*. A variable v is *live out* of an instruction inst if there is a path from inst to another instruction inst' that uses v, and v is not re-defined along this path. A variable is *live in* at an instruction inst if it is live out at inst, and it is not defined by inst. The result of liveness analysis is a mapping that gives, for each instruction, its IN and OUT sets. We will focus on liveness analysis for a single variable v, e.g., either IN = Live, or IN = Dead; same for OUT. Normally we find a solution to a data-flow problem by continuously solving a set of data-flow equations associated with each program region until a fix point is reached. Given a transfer function f_v^{inst}, and a meet operator $\wedge$ that we will define further, in our example these equations are:

$$\begin{cases} \mathsf{IN}[\mathsf{inst}] &= f_v^{inst}(\mathsf{OUT}[\mathsf{inst}]) \\ \mathsf{OUT}[\mathsf{inst}] &= \bigwedge_{S \in succ(inst)} \mathsf{IN}[S] \end{cases} \tag{5.1}$$

Because liveness analysis combines information that flows out of a node to find the information that flows into it, we call it a *backward* analysis. Forward analyses are the opposite: the meet operator combines the information that comes from the predecessors of a region to produce the information that flows to the successors of this region. Different types of program instructions are associated with different transfer functions. In the case of liveness analysis, we have three types of transfer functions, which depend on the instruction either using or defining the variable v:

Type of instruction inst	Transfer function
inst uses v	$f_v^{inst} = \lambda x.\text{Live}$
inst defines v and does not use v	$f_v^{inst} = \lambda x.\text{Dead}$
inst neither uses nor defines v	$f_v^{inst} = \lambda x.x$

Some program points are considered *meet nodes*, because they combine the information that comes from two or more regions. In the case of liveness analysis,

[1] Since "live" is a technical term, we use it, instead of "alive", even as a predicate adjective, e.g., "the variable is live".

conditional branches are meet nodes, because they are source of two different program paths. The variable of interest v may be live in one of these paths, and dead along the other. Information, in this case, is combined via the meet operator $\wedge$. For liveness analysis, this operator is defined by the table below, which says, for instance, that if a variable is dead along a path and live along the other, then it is live past that meet point:

$\wedge$	Dead	Live
Dead	Dead	Live
Live	Live	Live

Some transfer functions are identities. For instance, in liveness analysis, an instruction that neither defines nor uses any variable is associated with an identity transfer function. The goal of sparse data-flow analysis is to shortcut these functions, a task that we accomplish by grouping contiguous program points bound to identities into larger regions. Sometimes it is possible to perform this grouping more efficiently via a customized program representation [83; 44; 117; 60]. In particular, the class of Partitioned Data-flow Analyses (PDA), defined by Zadeck [144], greatly benefits from sparsity. These analyses, which include live variables, reaching definitions and forward/backward printing, can be decomposed into a set of sparse data-flow problems – usually one per variable – each independent on the other. For completeness, we re-state Zadeck's definition, as the sum of two notions: Partitioned Variable Problem (PVP) and Partitioned Variable Lattice (PVL).

Property 5.1 (PVP/PVL)**.** PARTITIONED VARIABLE PROBLEM*:*
Let $\mathcal{V} = \{v_1, \ldots, v_n\}$ be the set of program variables. We consider, without loss of generality, a forward data-flow analysis. This data-flow analysis is an equation system that associates, with each program point i, an element from a lattice $\mathcal{L}$, given by the equation $a^i = \bigwedge_{s \in pred(i)} F^s(a^s)$, where a^i denotes the abstract state associated with program point i, and F^s is the transfer function associated with program point s. The analysis can be written[2] *as a constraint system that binds to each program point i and each $s \in pred(i)$ the equation $a^i = a^i \wedge F^s(a^s)$ or, equivalently, the inequation $a^i \sqsubseteq F^s(a^s)$. The corresponding Maximum Fixed Point (MFP) problem is said to be a* Partitioned Variable Problem *iff:*

[PVL]: *$\mathcal{L}$ can be decomposed into the product of $\mathcal{L}_{v_1} \times \cdots \times \mathcal{L}_{v_n}$ where each $\mathcal{L}_{v_i}$ is the lattice associated with program variable v_i.*

[PVP]: *each transfer function F^s can also be decomposed into a product $F^s_{v_1} \times F^s_{v_2} \times \cdots \times F^s_{v_n}$ where $F^s_{v_j}$ is a function from $\mathcal{L}_{v_j}$ to $\mathcal{L}_{v_j}$.*

Let us from now, for any PVL problem, denote the abstract state associated with variable v at program point i, $[v]^i$.

Live analysis is a partitioned variable problem: the liveness information (lattice of Boolean values $\mathcal{B}$) can be computed for each *individual* (PVL property) variable *independently* (PVP property): the overall lattice can be written as a cross product $\mathcal{L} = \mathcal{B}^n$. The liveness information for variable v at program point i, e.g., $[v]^i$, can be expressed in term of its state at the successors s of i: $[v]^i = [v]^i \wedge F^s_v([v]^s)$ with F^s_v from $\mathcal{B}$ to $\mathcal{B}$.

[2] As far as we are concerned with finding its maximum solution. See for example Section 1.3.2 of [103].

Many data-flow analyses do *not* provide the PVP property; however, most of them *do* fulfill the PVL property. Consider a problem as simple as constant propagation as an example: if we denote by $\mathscr{C}$ the lattice of constants, the overall lattice can be written as $\mathscr{L} = \mathscr{C}^n$ with n the number of variables (PVL property); as opposed to liveness information, the constant value of some variable v at program point i has to be expressed in term of the constant value of *some other* variables (not only v) at the predecessors S of i: $[v]^i = [v]^i \wedge F_v^s([v_1]^s, \ldots, [v_n]^s)$ with F_v^s from $\mathscr{L}$ to $\mathscr{C}$ (and not from $\mathscr{C}$ to $\mathscr{C}$). Notice that there are data-flow analyses that do not meet the PVL property, such as those that rely on relations between variables [98].

If the information associated with a variable is invariant along its entire live range, then we can bind this information to the variable itself. In other words, we can replace all the constraint variables $[v]^i$ by a single constraint variable $[v]$, for each variable v and every $i \in \text{live}(v)$. In the context of constant propagation, at the program points $s \in \text{live}(v)$ that do not redefine a variable v, $[v]^i = [v]^i \wedge F_v^s([v_1]^s, \ldots, [v_n]^s) = [v]^i \wedge [v]^s$ simplifies into $[v] = [v]$. On the other hand, $F_v^{def(v)}$ simplifies to a function that depends only on some $[u]$ where each u is an argument of the instruction defining v. This gives the intuition on why a propagation engine along the def-use chains of a SSA-form program can be used to solve the constant propagation problem in an equivalent, yet "sparser", manner. This also paves the way toward a formal definition of the Static Single Information property.

Property 5.2 (SSI)**.** STATIC SINGLE INFORMATION: *Consider a forward (resp. backward) monotone PVL problem E_{dense} stated as a set of constraints $[v]^i \sqsubseteq F_v^s([v_1]^s, \ldots, [v_n]^s)$ for every variable v, each program point i, and each $s \in pred(i)$ (resp. $s \in succ(i)$). A program representation fulfills the Static Single Information property iff:*

[SPLIT]: *for each variable v, each $i \in live(v)$ with a unique predecessor (resp. successor) s, such that $F_v^s \neq \lambda x.\bot$ is non-trivial, i.e. is not the simple projection on $\mathscr{L}_v$ (see Definition 5.7 in 5.7), then s should contain a definition (resp. last use) of v; Let $(Y_v^i)_{(v,i) \in variables \times prog\_points}$ be a maximum solution to E_{dense}. Each join (resp. split) node i for which $F_v^s(Y_{v_1}^s, \ldots, Y_{v_n}^s)$ has different values on its incoming edges (s,i) (resp. outgoing edges (i,s)) should have a ϕ-function at the entry of i (resp. σ function at the exit of i) for v as defined in Section 5.2.1.*

[INFO]: *each program point $i \notin live(v)$ should be bound to an undefined (see Definition 5.7) transfer function, e.g., $F_v^i = \lambda x.\bot$.*

[LINK]: *each instruction* inst *for which F_v^{inst} depends on some $[u]^s$ (see Definition 5.7) should contain a (potentially pseudo) use (resp. def) of u live-out (resp. live-in) of s.*

[VERSION]: *for each variable v, $live(v)$ is a connected component of the CFG.*

These properties allows us to attach the information to variables, instead of program points. The SPLIT property forces the information related to a variable to be invariant along its entire live-range. INFO forces this information to be irrelevant outside the live range of the variable. The LINK property forces the def-use chains to reach the points where information is available for a transfer function to be evaluated. The VERSION property provides an one-to-one mapping between variable names and live ranges.

We must split live ranges to provide the SSI properties. If we split them between each pair of consecutive instructions, then we would automatically provide these

properties, as the newly created variables would be live at only one program point. However, this strategy would lead to the creation of many trivial program regions, and we would lose sparsity. In Section 5.3 we provide a sparser way to split live ranges that fit Property 5.2. Possibly, we may have to extend the live-range of a variable to cover every program point where the information is relevant. We accomplish this last task by inserting into the program pseudo-uses and pseudo-definitions of this variable.

5.2.1 Special instructions used to split live ranges

We group program points in three kinds: interior nodes, branches and joins. At each place we use a different notation to denote live range splitting.

Interior nodes are program points that have a unique predecessor and a unique successor. At these points we perform live range splitting via copies. If the program point already contains another instruction, then this copy *must* be done *in parallel* with the existing instruction. The notation,

$$inst \parallel v_1 = v_1' \parallel \ldots \parallel v_m = v_m'$$

denotes m copies $v_i = v_i'$ performed in parallel with instruction $inst$. This means that all the uses of $inst$ plus all v_i' are read simultaneously, then $inst$ is computed, then all definitions of $inst$ plus all v_i are written simultaneously.

We call *joins* the program points that have one successor and multiple predecessors. For instance, two different definitions of the same variable v might be associated with two different constants; hence, providing two different pieces of information about v. To avoid that these definitions reach the same use of v we merge them at the earliest program point where they meet. We do it via special instructions called ϕ-functions, which were introduced by Cytron *et al.* to build SSA-form programs [55]. The assignment

$$v_1 = \phi(v_1^1 : l^1, \ldots, v_1^q : l^q) \parallel \ldots \parallel v_m = \phi(v_m^1 : l^1, \ldots, v_m^q : l^q)$$

contains m ϕ-functions to be performed in parallel. The ϕ symbol works as a multiplexer. It will assign to each v_i the value in v_i^j, where j is determined by l^j, the basic block last visited before reaching the ϕ assignment. The above statement encapsulates m parallel copies: all the variables $v_1^j, \ldots, v_m^j$ are simultaneously copied into the variables $v_1, \ldots, v_m$.

In backward analyses the information that emerges from different uses of a variable may reach the same *branch point*, which is a program point with a unique predecessor and multiple successors. To ensure Property 5.2, the use that reaches the definition of a variable must be unique, in the same way that in a SSA-form program the definition that reaches a use is unique. We ensure this property via special instructions that Ananian has called σ-functions [6]. The σ-functions are the dual of ϕ-functions, performing a parallel assignment depending on the execution path taken. The assignment

$$(v_1^1 : l^1, \ldots, v_1^q : l^q) = \sigma(v_1) \parallel \ldots \parallel (v_m^1 : l^1, \ldots, v_m^q : l^q) = \sigma(v_m)$$

represents m σ-functions that assign to each variable v_i^j the value in v_i if control flows into block l^j. These assignments happen in parallel, i.e., the m σ-functions

encapsulate m parallel copies. Also, notice that variables live in different branch targets are given different names by the σ-function that ends that basic block.

5.2.2 Propagation engine

As mentioned earlier for any program that fulfills the SSI property of a given PVL problem, a propagation engine along the def-use chains can be used to solve it sparsely. Let us consider a unidirectional forward (resp. backward) PVL problem E^{ssi}_{dense} stated as a set of equations $[v]^i \sqsubseteq F^s_v([v_1]^s,\ldots,[v_n]^s)$ (or equivalently $[v]^i = [v]^i \wedge F^s_v([v_1]^s,\ldots,[v_n]^s)$ for every variable v, each program point i, and each $s \in pred(i)$ (resp. $s \in succ(i)$).

We see two ways to handle ϕ and σ-functions during the data-flow analysis. Either we consider each of them as a whole and get only one equation per ϕ or σ-function, or we consider them as a set of copies and then have as many equations as the number of predecessors, in the case of ϕ-functions, or successors, in the case of σ-functions. We have opted for the second choice, because it simplifies our notation. Any ϕ-function $a = \phi(a_1 : l^1,\ldots,a_m : l^m)$ (resp. σ-function $(a_1 : l^1,\ldots,a_m : l^m) = \sigma(a)$) at program point i leads to as many constraints as the set of predecessors (resp. successors) l^j of i. In other words, a ϕ-function such as $a = \phi(a_1 : l^1,\ldots,a_m : l^m)$, gives us n constraints such as $[a]^i \sqsubseteq F^{l^j}_v([v_1]^{l^j},\ldots,[v_n]^{l^j})$, which usually simplifies into $[a]^i \sqsubseteq [a_j]^{l^j}$, and then can write equivalently into the classical meet $[a]^i \sqsubseteq \bigwedge_{l^j \in pred(i)} [a_j]^{l^j}$. Similarly, a σ-function $(l^1 : a_1,\ldots,l^m : a_m) = \sigma(a)$ at program point i yields n constraints such as $[a_j]^{l_j} \sqsubseteq F^{l^j}_v([v_1]^{l^j},\ldots,[v_n]^{l^j})$, which usually simplifies into $[a_j]^{l_j} \sqsubseteq [a]^i$.

Given a program that fulfills the SSI property for E^{ssi}_{dense} and the set of transfer functions F^s_v, we show here how to build an equivalent sparse constrained system.

Definition 5.3 (SSI constrained system)**.** Consider that a program in SSI form gives us a constraint system that associates with each variable v the constraints $[v]^i \sqsubseteq F^s_v([v_1]^s,\ldots,[v_n]^s)$. We define a system of sparse equations E^{ssi}_{sparse} as follows:

- For each instruction at a program point i that defines (resp. uses) a variable v, we let $a\ldots b$ be its set of used (resp. defined) variables. Because of the LINK property, F^i_v depends only on some $[a]^i\ldots[b]^i$. Thus, there exists a function G^i_v defined as the restriction of F^i_v on $\mathscr{L}_a \times \cdots \times \mathscr{L}_b$, i.e. $G^i_v([a],\ldots,[b]) = F^i_v([v_1],\ldots,[v_n])$.
- The sparse constrained system associates with each variable v, and each definition (resp. use) point i of v, the corresponding constraint $[v] \sqsubseteq G^i_v([a],\ldots,[b])$ where $a,\ldots,b$ are used (resp. defined) at i.

The SSI constrained system might have several inequations for the same left-hand-side. This is due to the way that we handle ϕ and σ functions but not only. Indeed, our definition of the SSI property, as opposed to the original ones [128; 6], does not ensure the SSA or the SSU properties, because such a guarantee is not necessary to every sparse analysis. It is a common assumption in the compiler's literature that "data-flow analysis (...) can be made simpler when each variable has only one definition", as stated in Chapter 19 of Appel's textbook [10]. A naive interpretation of the above statement could lead one to conclude that data-flow analyses become simpler as soon as the program representation enforces a single source of information per live-range: SSA for forward propagation, SSU for backward, and the

```
1   function back_propagate(transfer_functions 𝒢)
2       worklist = ∅
3       foreach v ∈ vars: [v] = ⊤
4       foreach i ∈ insts: worklist += i
5       while worklist ≠ ∅:
6           let i ∈ worklist; worklist −= i
7           foreach v ∈ i.uses():
8               [v]_new = [v] ∧ G_v^i([i.defs()])
9               if [v] ≠ [v]_new:
10                  stack += v.defs()
11                  [v] = [v]_new
```

Figure 5.1: Backward propagation engine under SSI

```
1   function forward_propagate(transfer_functions 𝒢)
2       worklist = ∅
3       foreach v ∈ vars: [v] = ⊤
4       foreach i ∈ insts: worklist += i
5       while worklist ≠ ∅:
6           let i ∈ worklist; worklist −= i
7           foreach v ∈ i.defs():
8               [v]_new = [v] ∧ G_v^i([i.uses()])
9               if [v] ≠ [v]_new:
10                  stack += v.uses()
11                  [v] = [v]_new
```

Figure 5.2: Forward propagation engine under SSI

original SSI for bi-directional analyses. This premature conclusion is contradicted by the example of dead-code elimination, a backward data-flow analysis that the SSA form simplifies. In fact, the SSA form fulfills our definition of the SSI property for dead-code elimination. Nevertheless, the corresponding constraint system has several inequations (one per variable use) for the same left-hand-side (one for each variable). It is well known that such a system can be solved using chaotic iteration such as the worklist algorithm [103, Sec 6.1] given in Figures 5.1 and 5.2: replace G_v^i in Figure 5.1 by "*i is a useful instruction or one of its definitions is marked as useful*" and one obtains the classical algorithm for dead-code elimination.

The following theorem proved in 5.8 states the equivalence between sparse and dense analyses.

Theorem 5.4 (sparse ≡ dense)**.** *Consider a program in SSI-form that gives origin to a constraint system E^{ssi}_{dense} associating with each variable v the constraints $[v]^i = [v]^i \wedge F^s_v([v_1]^s,\ldots,[v_n]^s)$. Suppose that each F^s_v is a monotone function from $\mathscr{L}^n$ to $\mathscr{L}$ where $\mathscr{L}$ is of finite height. Let $(Y_v)_{v\in variables}$ be the maximum solution of the corresponding sparse constraint system.*

Then, $(X^i_v)_{(v,i)\in variables\times prog\_points}$ with $\begin{cases} X^i_v = Y_v & for\ i \in live(v) \\ X^i_v = \bot & otherwise \end{cases}$ is the maximum solution to E^{ssi}_{dense}.

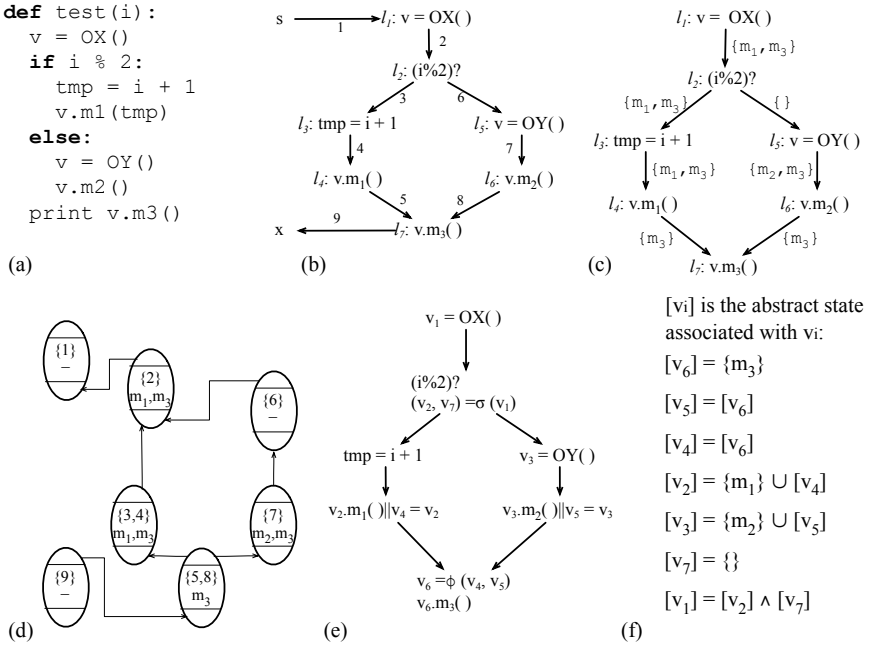


Figure 5.3: Class inference analysis as an example of backward data-flow analysis that takes information from the uses of variables.

5.2.3 Examples of sparse data-flow analyses

As we have mentioned before, many data-flow analyses can be classified as PVP/PVL problems. In this section we present some meaningful examples.

Class Inference Some dynamically typed languages, such as Python, JavaScrip, Ruby or Lua, represent objects as tables containing methods and fields. It is possible to improve the execution of programs written in these languages if we can replace these simple tables by actual classes with virtual tables [43]. A class inference engine tries to assign a class to a variable v based on the ways that v is used. The Python program in Figure 5.3(a) illustrates this optimization. Our objective is to infer the correct suite of methods for each object bound to variable v. Figure 5.3(b) shows the control flow graph of the program, and Figure 5.3(c) shows the results of a dense implementation of this analysis. Notice that each program instruction is associated with a transfer function, and that some of these functions, such as that in label l_3, are trivial, having no influence on the data-set that they create. Choi *et al.* [44] would perform the class inference analysis on the SEG in Figure 5.3(d). Each node of this graph is labeled with the edges of the CFG that it groups together. All the CFG edges grouped by a node have the same data-flow information, as one can verify in Figure 5.3(c). We show this information – a set of methods – in each SEG node. The SEG edges point in the direction that information flows between program regions. Instead of using a separate data-structure, like Choi *et al.* do, we work directly on the program, producing the representation given in Figure 5.3(e). Because type inference is a backward analysis that extracts information from use sites, we

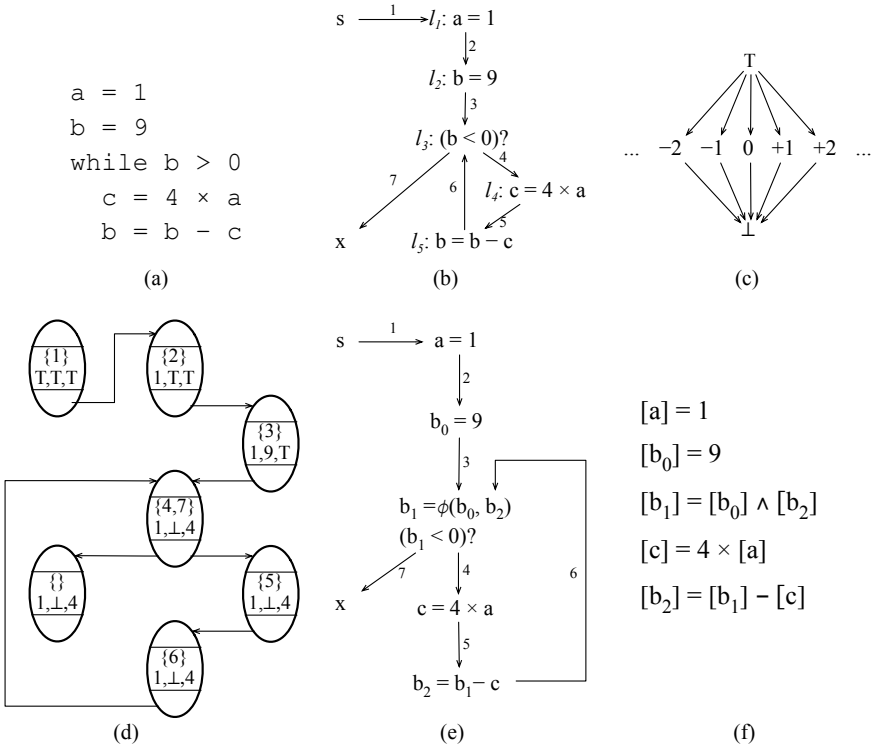


Figure 5.4: Constant propagation as an example of forward data-flow analysis that takes information from the definitions of variables.

split live ranges at these program points, and rely on σ-functions to merge them back. We want to stay in SSA-form; hence, we must also insert ϕ-function to join the live ranges that denote the same variable definition. The use-def chains that we derive from the program representation lead naturally to a constraint system, which we show in Figure 5.3(f), where $[v_j]$ is the information associated with variable v_j. A fix-point to this constraint system is a solution to our data-flow problem. Class inference is a Partitioned Variable Problem (PVP)[3], because the data-flow information associated to a variable v can be computed independently from the other variables. In the words of Choi *et al.*, SEGs are “specially attractive" for this kind of problem.

Constant Propagation There exist many data-flow analyses that are not Partitioned Variable Problems. Constant propagation is an example: in this analysis, the abstract state of a variable v is determined by the abstract state of the variables used to define v. Figure 5.4 illustrates constant propagation. We want to find out which variables in the program of Figure 5.4(a) can be replaced by constants. The CFG of this program is given in Figure 5.4(b). Constant propagation has a very simple lattice, which we show in Figure 5.4(c). The SEG created for this instance of constant propagation is given in Figure 5.4(d). Every instruction in this example either gen-

[3]Actually class inference is no more a PVP as soon as we want to propagate the information through copies.

erates information, or merges it; thus, the SEG contains a node representing each instruction. We have augmented each SEG node with the edges that it represent in the CFG, plus the final result of the constant propagation problem in that region. Because we have three variables, each node is associated to a three dimensional vector $([a],[b],[c])$, where $[x]$ is the abstract state of variable x, as given by the lattice in Figure 5.4(c). Our approach, to this kind of problem is sparser, because we bind a lattice value directly to each live range, instead of having to associate product lattices to program regions. In constant propagation, information is produced at the program points where variables are defined. Thus, in order to provide Property 5.2, we must guarantee that each program point is dominated by a single definition of a variable. Figure 5.4(e) shows the intermediate representation that we create for the program in Figure 5.4(b). In this case, our intermediate representation is equivalent to the SSA form. The def-use chains implicit in our program representation lead to the constraint system shown in Figure 5.4(f).

Taint analysis The objective of taint analysis [120] is to find program vulnerabilities. In this case, a harmful attack is possible when input data reaches sensitive program sites without going through special functions called sanitizers. Figure 5.5 illustrates this type of analysis. We have used ϕ and σ-functions to split the live ranges of the variables in Figure 5.5(a) producing the program in Figure 5.5(b). Lets assume that *echo* is a sensitive function, because it is used to generate web pages. For instance, if the data passed to *echo* is a JavaScript program, then we could have an instance of cross-site scripting attack. Thus, the statement *echo* v_1 may be a source of vulnerabilities, as it outputs data that comes directly from the program input. On the other hand, we know that *echo* v_2 is always safe, for variable v_2 is initialized with a constant value. The call *echo* v_5 is always safe, because variable v_5 has been sanitized; however, the call *echo* v_4 might be tainted, as variable v_4 results from a failed attempt to sanitize v. The def-use chains that we derive from the program representation leads naturally to a constraint system, which we show in Figure 5.5(c). The intermediate representation that we create in this case is equivalent to the *Extended Single Static Assignment* (e-SSA) form [16]. It also suits the ABCD algorithm for array bounds-checking elimination [16], Su and Wagner's range analysis [133] and Gawlitza *et al.*'s range analysis [67].

Null pointer analysis The objective of null pointer analysis is to determine which references may hold null values. Nanda and Sinha have used a variant of this analysis to find which method dereferences may throw exceptions, and which may not [102]. This analysis allows compilers to remove redundant null-exception tests and helps developers to find null pointer dereferences. Figure 5.6 illustrates this analysis. Because information is produced at use sites, we split live ranges after each variable is used, as we show in Figure 5.6(b). For instance, we know that the call $v_2.m()$ cannot result in a null pointer dereference exception, otherwise an exception would have been thrown during the invocation $v_1.m()$. On the other hand, in Figure 5.6(c) we notice that the state of v_4 is the meet of the state of v_3, definitely not-null, and the state of v_1, possibly null, and we must conservatively assume that v_4 may be null.

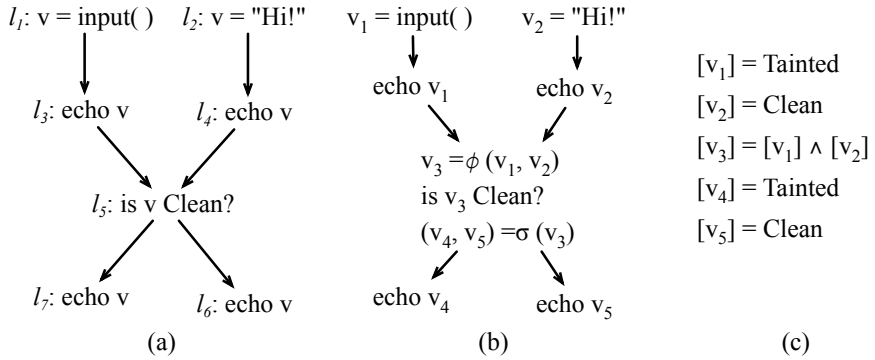


Figure 5.5: Taint analysis as an example of forward data-flow analysis that takes information from the definitions of variables and conditional tests on these variables.

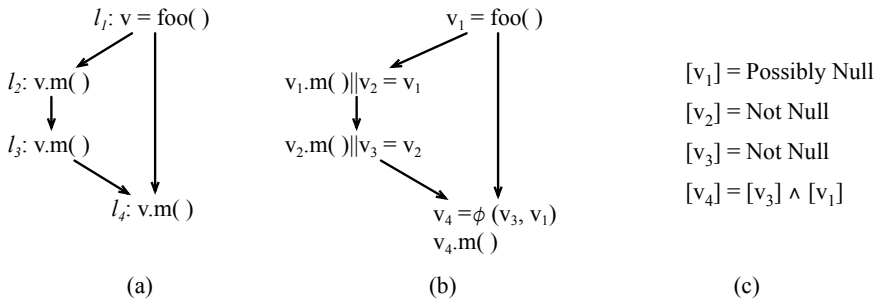


Figure 5.6: Null pointer analysis as an example of forward data-flow analysis that takes information from the definitions and uses of variables.

5.3 Building the Intermediate Program Representation

A *live range splitting strategy* $\mathscr{P}_v = I_\uparrow \cup I_\downarrow$ over a variable v consists of a set of "oriented" program points. We let $I_\downarrow$ denote a set of points i with forward direction. Similarly, we let $I_\uparrow$ denote a set of points i with backward direction. The live-range of v must be split at least at every point in $\mathscr{P}_v$. Going back to the examples from Section 5.2.3, we have the live range splitting strategies enumerated below. The list in Figure 5.7 gives further examples of live range splitting strategies.

- Class inference is a backward analysis that takes information from the uses of variables; thus, for each variable, the live-range splitting strategy is characterized by the set $Uses_\uparrow$ where *Uses* is the set of use points. For instance, in Figure 5.3(e), we have that $\mathscr{P}_v = \{l_4, l_6, l_7\}_\uparrow$.
- Constant propagation is a forward analysis that takes information from definition sites. Thus, for each variable v the live-range splitting strategy is characterized by the set $Defs_\downarrow$ where *Defs* is the set of definition points. For instance,

Client	Splitting strategy $\mathscr{P}$
Alias analysis, reaching definitions cond. constant propagation [139]	$Defs_{\downarrow}$
Partial Redundancy Elimination [6; 128]	$Defs_{\downarrow} \bigcup LastUses_{\uparrow}$
ABCD [16], taint analysis [120], range analysis [133; 67]	$Defs_{\downarrow} \bigcup Out(Conds)_{\downarrow}$
Stephenson's bitwidth analysis [131]	$Defs_{\downarrow} \bigcup Out(Conds)_{\downarrow} \bigcup Uses_{\uparrow}$
Mahlke's bitwidth analysis [95]	$Defs_{\downarrow} \bigcup Uses_{\uparrow}$
An's type inference [80], Class inference [43]	$Uses_{\uparrow}$
Hochstadt's type inference [136]	$Uses_{\uparrow} \bigcup Out(Conds)_{\uparrow}$
Null-pointer analysis [102]	$Defs_{\downarrow} \bigcup Uses_{\downarrow}$

Figure 5.7: Live range splitting strategies for different data-flow analyses. We use *Defs* (*Uses*) to denote the set of instructions that define (use) the variable; *Conds* to denote the set of instructions that apply a conditional test on a variable; Out(*Conds*) the exits of the corresponding basic blocks; *LastUses* to denote the set of instructions where a variable is used, and after which it is no longer live.

in Figure 5.4, we have that $\mathscr{P}_b = \{l_2, l_5\}_{\downarrow}$.

- Taint analysis is a forward analysis that takes information from points where variables are defined, and conditional tests that use these variables. For instance, in Figure 5.5, we have that $\mathscr{P}_v = \{l_1, l_2, \text{Out}(l_5)\}_{\downarrow}$ where $\text{Out}(l_i)$ denotes the exit of l_i.
- Nanda *et al.*'s null pointer check [102] is a forward analysis that takes information from definitions and uses. For instance, in Figure 5.6, we have that $\mathscr{P}_v = \{l_1, l_2, l_3, l_4\}_{\downarrow}$.

The algorithm SSIfy in Figure 5.8 implements a live range splitting strategy in three steps. Firstly, it splits live ranges, inserting new definitions of variables into the program code. Secondly, it renames these newly created definitions; hence, ensuring that the live ranges of two different re-definitions of the same variable do not overlap. Finally, it removes dead and "unborn" definitions from the program code. We describe each of these phases in the rest of this section.

```
1  function SSIfy(var v, Splitting_Strategy P_v)
2      split(v, P_v)
3      rename(v)
4      clean(v)
```

Figure 5.8: Split the live ranges of v to convert it to SSI form

Splitting live ranges through the creation of new definitions of variables In order to implement $\mathscr{P}_v$ we must split the live ranges of v at each program point listed by

$\mathscr{P}_v$. However, these points are not the only ones where splitting might be necessary. As we have pointed out in Section 5.2.1, we might have, for the same original variable, many different sources of information reaching a common program point. For instance, in Figure 5.4(b), there exist two definitions of variable b – l_2 and l_5 – that reach the use of b at l_3. The information that flows forward from l_2 and l_5 collides at l_3, the merge point of the if-then-else. Hence the live-range of b has to be split immediately before l_3 – at $\text{In}(l_3)$ –, leading, in our example, to a new definition b_1. In general, the set of program points where information collides can be easily characterized by join sets [53]. The join set of a set of nodes P contains the CFG nodes that can be reached by two or more nodes of P through disjoint paths. Join sets created by the forward propagation of information can be over-approximated via the notion of iterated dominance frontier [140]. This concept is the basics of SSA construction, and for completeness we recall its definition below:

- **Dominance**: a CFG node n dominates a node n' if every program path from the entry node of the CFG to n' goes across n. If $n \neq n'$, then we say that n *strictly* dominates n'.

- **Dominance frontier** (DF): a node n' is in the dominance frontier of a node n if n dominates a predecessor of n', but does not strictly dominate n'.

- **Iterated dominance frontier** (DF^+): the iterated dominance frontier of a node n is the limit of the sequence:

$$\begin{aligned} DF_1 &= DF(n) \\ DF_{i+1} &= DF_i \cup \{DF(z) \mid z \in DF_i\} \end{aligned}$$

Similarly, split sets created by the backward propagation of information can be over-approximated by the notion of *iterated post-dominance frontier* (pDF^+), which is the dual of DF^+ [10]. That is, the post-dominance frontier is the dominance frontier in a CFG where direction of edges have been reversed.

Figure 5.9 shows the algorithm that we use to create new definitions of variables. This algorithm has three main phases. First, in lines 3-9 we create new definitions to split the live ranges of variables due to backward collisions of information. These new definitions are created at the iterated post-dominance frontier of points that originate information. If a program point is a join node, then each of its predecessors will contain the live range of a different definition of v, as we ensure in line 6 of our algorithm. Notice that these new definitions are not placed parallel to an instruction, but in the region immediately after it, which we denote by Out(...). In lines 10-16 we perform the inverse operation: we create new definitions of variables due to the forward collision of information. Our starting points, in this case, include also the original definitions of v, as we see in line 11, because we want to stay in SSA form in order to have access to a fast liveness check. Finally, in lines 17-23 we actually insert the new definitions of v. These new definitions might be created by σ functions (due to $\mathscr{P}_v$ or to the splitting in lines 3-9); by ϕ-functions (due to $\mathscr{P}_v$ or to splitting in lines 10-16); or by parallel copies. Contrary to Singer's algorithm, originally designed to produce SSI form programs, we do not iterate between the insertion of ϕ and σ functions. Nevertheless, as we show in the Appendix, our method is enough to ensure the SSI properties for any combination of unidirectional problems.

```
1  function split(var v, Splitting_Strategy P_v = I_↓ ∪ I_↑)
2      "compute the set of split points"
3      S_↑ = ∅
4      foreach i ∈ I_↑:
5          if i.is_join:
6              foreach e ∈ incoming_edges(i):
7                  S_↑ = S_↑ ⋃ Out(pDF+(e))
8          else:
9              S_↑ = S_↑ ⋃ Out(pDF+(i))
10     S_↓ = ∅
11     foreach i ∈ S_↑ ⋃ Defs(v) ⋃ I_↓:
12         if i.is_branch:
13             foreach e ∈ outgoing_edges(i)
14                 S_↓ = S_↓ ⋃ In(DF+(e))
15         else:
16             S_↓ = S_↓ ⋃ In(DF+(i))
17     S = P_v ⋃ S_↑ ⋃ S_↓
18     "Split live range of v by inserting φ, σ, and copies"
19     foreach i ∈ S:
20         if i does not already contain any definition of v:
21             if i.is_join: insert "v = φ(v,...,v)" at i
22             elseif i.is_branch: insert "(v,...,v) = σ(v)" at i
23             else: insert a copy "v = v" at i
```

Figure 5.9: Live range splitting

Variable Renaming The algorithm in Figure 5.10 builds def-use and use-def chains for a program after live range splitting. This algorithm is similar to the standard algorithm used to rename variables during the SSA construction [10, Algorithm 19.7]. To rename a variable v we traverse the program's dominance tree, from top to bottom, stacking each new definition of v that we find. The definition currently on the top of the stack is used to replace all the uses of v that we find during the traversal. If the stack is empty, this means that the variable is not defined at this point. The renaming process replaces the uses of undefined variables by $\perp$ (line 3). We have two methods, stack.set_use and stack.set_def to build the chain relations between the variables. Notice that sometimes we must rename a single use inside a ϕ-function, as in lines 19-20 of the algorithm. For simplicity we consider this single use as a simple assignment when calling stack.set_use, as one can see in line 20. Similarly, if we must rename a single definition inside a σ-function, then we treat it as a simple assignment, like we do in lines 15-16 of the algorithm.

Dead and Undefined Code Elimination The algorithm in Figure 5.11 eliminates ϕ-functions that define variables not actually used in the code, σ-functions that use variables not actually defined in the code, and parallel copies that either define or use variables that do not reach any actual instruction. We mean by "actual" instructions, those instructions that already existed in the program before we transformed it with split. In line 3 we let "web" be the set of versions of v, so as to restrict the cleaning process to variable v, as we see in lines 4-6 and lines 10-12. The set "active" is initialized to actual instructions in line 4. Then, during the loop in lines 5-8 we add to active ϕ-functions, σ-functions, and copies that can reach actual definitions through use-def chains. The corresponding version of v is then marked as *defined*

```
1  function rename(var v)
2      "Compute use-def & def-use chains"
3      "We consider here that stack.peek() = ⊥ if stack.isempty(),
4       and that def(⊥) = entry"
5      stack = ∅
6      foreach CFG node n in dominance order:
7          if exists v = φ(v : l^1, ..., v : l^q) in In(n):
8              stack.set_def(v = φ(v : l^1, ..., v : l^q))
9          foreach instruction u in n that uses v:
10             stack.set_use(u)
11         if exists instruction d in n that defines v:
12             stack.set_def(d)
13         foreach instruction (...) = σ(v) in Out(n):
14             stack.set_use((...) = σ(v))
15         if exists (v : l^1, ..., v : l^q) = σ(v) in Out(n):
16             foreach v : l^i = v in (v : l^1, ..., v : l^q) = σ(v):
17                 stack.set_def(v : l^i = v)
18         foreach m in successors(n):
19             if exits v = φ(..., v : l^n, ...) in In(m):
20                 stack.set_use(v = v : l^n)
21 function stack.set_use(instruction inst):
22     while def(stack.peek()) does not dominate inst: stack.pop()
23     v_i = stack.peek()
24     replace the uses of v by v_i in inst
25     if v_i ≠ ⊥: set Uses(v_i) = Uses(v_i) ⋃ inst
26 function stack.set_def(instruction inst):
27     let v_i be a fresh version of v
28     replace the defs of v by v_i in inst
29     set Def(v_i) = inst
30     stack.push(v_i)
```

Figure 5.10: Versioning

(line 8). The next loop, in lines 11-14 performs a similar process, this time to add to the active set, instructions that can reach actual uses through def-use chains. The corresponding version of v is then marked as *used* (line 14). Each non live variable (see line 15), i.e. either undefined or dead (non used) is replaced by $\perp$ in all ϕ, σ, or copy functions where it appears in. This is done by lines 15-18. Finally every useless ϕ, σ, or copy functions are removed by lines 19-20. As a historical curiosity, Cytron *et al.*'s procedure to build SSA form produced what is called *the minimal representation* [53]. Some of the ϕ-functions in the minimal representation define variables that are never used. Briggs *et al.* [36] remove these variables; hence, producing what compiler writers normally call *pruned SSA-form*.

5.3.1 Implementing parallel copies, ϕ and σ-functions

Traditional instruction sets do not provide ϕ-functions nor σ-functions. Thus, before producing an executable program, the compiler must implement these instructions somehow. Normally, ϕ-functions and parallel copies are replaced by ordinary copy instructions, as discussed by Briggs *et al.* [36] or Boissinot *et al* [21]. There exists ways to implement the semantics of parallel copies via simple copies without increasing the register pressure in the source program [109]. The implementation of

```
1   clean(var v)
2       let web = {v_i | v_i is a version of v}
3       let defined = ∅
4       let active = { inst | inst actual instruction and web ∩ inst.defs ≠ ∅}
5       while ∃inst ∈ active s.t. web ∩ inst.defs\defined ≠ ∅:
6           foreach v_i ∈ web ∩ inst.defs\defined:
7               active = active ∪ Uses(v_i)
8               defined = defined ∪ {v_i}
9       let used = ∅
10      let active = { inst | inst actual instruction and web ∩ inst.uses ≠ ∅}
11      while ∃inst ∈ active s.t. inst.uses\used ≠ ∅:
12          foreach v_i ∈ web ∩ inst.uses\used:
13              active = active ∪ Def(v_i)
14              used = used ∪ {v_i}
15      let live = defined ∩ used
16      foreach non actual inst ∈ Def(web):
17          foreach v_i operand of inst s.t. v_i ∉ live:
18                      replace v_i by ⊥
19          if inst.defs = {⊥} or inst.uses = {⊥}
20              remove inst
```

Figure 5.11: Dead and undefined code elimination. Original instructions not inserted by split are called *actual* instruction. We let inst.defs denote the set of variable(s) defined by inst, and inst.uses denote the set of variables used by inst.

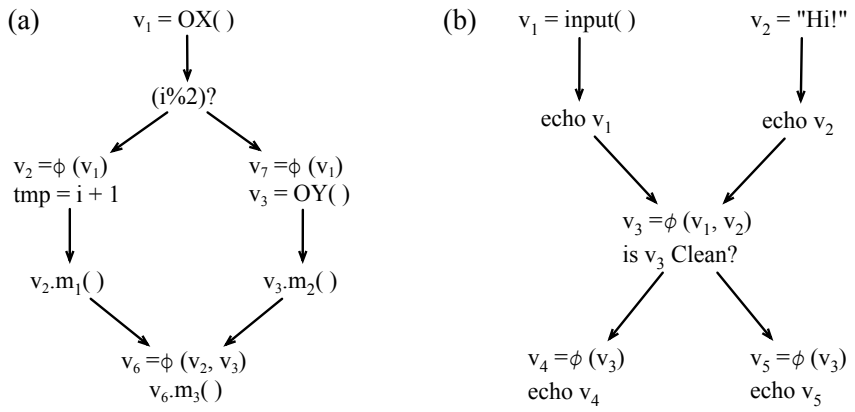


Figure 5.12: (a) getting rid of copies and σ-functions; (b) implementing σ-functions via single arity ϕ-functions.

σ-functions; however, has not been discussed in the literature. A possible solution is to get rid of copies and σ-functions by simply copy-propagating them; thus, leaving for the SSA elimination module the task of replacing σ-functions with special instructions. As an example, Figure 5.12(a) shows the result of copy folding applied on Figure 5.3(e).

Alternatively, σ-functions can be implemented as single arity ϕ-functions. As an example, Figure 5.12(b) shows how we would represent the σ-functions in Figure 5.5(b). If l is a branch point with n successors that would contain a σ-function

$(v_1 : l^1, \ldots, v_n : l^n) = \sigma(v)$, then, for each successor l_i of l, we insert at the beginning of l_i an instruction $v_i = \phi(v : l)$. Notice that it is possible that l_i already contains a ϕ-function for v. This case happens when the control flow edge $l \rightarrow l_i$ is *critical*. A critical edge links a basic block with several successors to a basic block with several predecessors. If l_i already contains a ϕ-function $v' = \phi(\ldots, v_i, \ldots)$, then we rename v_i to v.

5.3.2 Deriving dense information from sparse analyses

We can use our sparse data-flow analysis framework to solve even some data-flow problems that demand information at every program point, such as bitwidth analysis [95; 131; 67; 132]. There exist clients of bit-width analyses that need to know the bit sizes of the variables at particular program points. For instance, Barik *et al.* [13] have designed a bit-width aware register allocator. In this setting, the register pressure at a program point p is the sum of the bit sizes of all the variables live at p. We can support the register allocator of Barik *et al.* by coupling the result of the sparse bit-width analysis with live range information. We can perform this coupling efficiently, because algorithm SSIfy preserves the single static assignment property.

Preserving the SSA properties is key due to two reasons. First, liveness analysis has a non-iterative implementation for SSA-form programs linear on the program size [10, p.429]. Second, if we only need liveness information for some specific variables, at some specific program points, then there is a fast liveness check for SSA-form programs. The problem of answering the question "is variable v live at program point p" has an algorithm that is $O(U)$, where U is the number of times that v is used in the program code [22]. Over 95% of variables found in common benchmarks are used less than 5 times [22, p.42]; thus, this asymptotic complexity is constant in practice.

5.4 Experimental Results

This section describes experiments that we have performed to probe the size and the runtime efficiency of the algorithms that we use to build intermediate representations. Our experiments were conducted on a dual core `Intel Pentium D` of 2.80GHz of clock, 1GB of memory, running `Linux Gentoo`, version 2.6.27. Our framework runs in LLVM 2.5 [88], and it passes all the tests that LLVM does. The LLVM test suite consists of over 1.3 million lines of C code. In this chapter we will be showing only the results of compiling SPEC CPU 2000. In order to compare different live range splitting strategies we generate program representations to three different LLVM analyses:

1. *SSI*: We use Ananian's Static Single Information form [6] as a baseline for our experiments. We build the SSI program representation via Singer's iterative algorithm.

2. *ABCD*: $(\{def, cond\}_{\downarrow})$. This live range splitting strategy generalizes the ABCD algorithm for array bounds checking elimination [16]. An example of this live range splitting strategy is given in Figure 5.5.

3. *CCP*: $(\{def, cond_{eq}\}_{\downarrow})$. This live range splitting strategy, which supports Wegman *et al.*'s [139] conditional constant propagation, is a subset of the previous

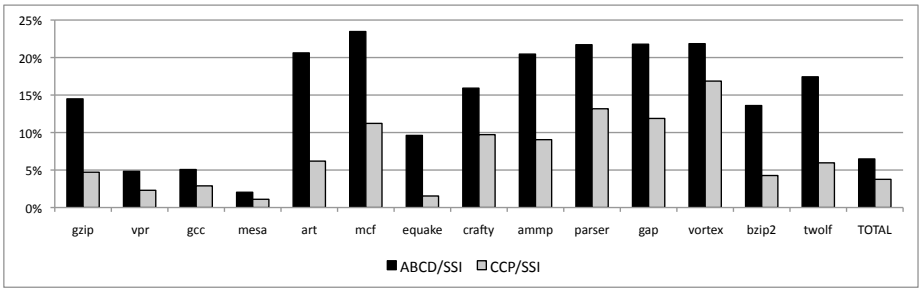


Figure 5.13: Comparison of the time taken to produce the different program representations. 100% is the time of using the SSI live range splitting strategy. The shorter the bar, the faster the live range splitting strategy. The SSI conversion took 1315.2s in total, the ABCD conversion took 85.2s, and the CCP conversion took 49.4s.

strategy. Differently of the ABCD client, this client requires that only variables used in equality tests, e.g., ==, undergo live range splitting. That is, $cond_{eq}(v)$ denotes the conditional tests that check if v equals a given value.

For an explanation about the sets *defs*, *uses* and *conds*, see Figure 5.7.

5.4.1 Runtime

The chart in Figure 5.13 compares the execution time of the three live range splitting strategies. We show only the time to perform live range splitting. The time to execute the optimization itself, removing array bounds check or performing constant propagation, is not shown. The bars are normalized to the running time of the *SSI* live range splitting strategy. On the average, the ABCD client runs in 6.8% and the CCP client runs in 4.1% of the time of SSI. These two forward analyses tend to run faster in benchmarks with sparse control flow graphs, which present fewer conditional branches, and therefore fewer opportunities to restrict the ranges of variables.

In order to put the time reported in Figure 5.13 in perspective, Figure 5.14 compares the running time of our live range splitting algorithms with the time to run the other standard optimizations in our baseline compiler[4]. In our setting, LLVM -O1 runs 67 passes, among analysis and optimizations, which include partial redundancy elimination, constant propagation, dead code elimination, global value numbering and invariant code motion. We believe that this list of passes is a meaningful representative of the optimizations that are likely to be found in an industrial strength compiler. The bars are normalized to the optimizer's time, which consists of the time taken by machine independent optimizations plus the time taken by one of the live range splitting clients, e.g, ABCD or CCP. The ABCD client takes 1.48% of the optimizer's time, and the CCP client takes 0.9%. To emphasize the speed of these passes, we notice that the bars do not include the time to do machine dependent optimizations such as register allocation.

[4]To check the list of LLVM's target independent optimizations try `llvm-as < /dev/null | opt -std-compile-opts -disable-output -debug-pass=Arguments`

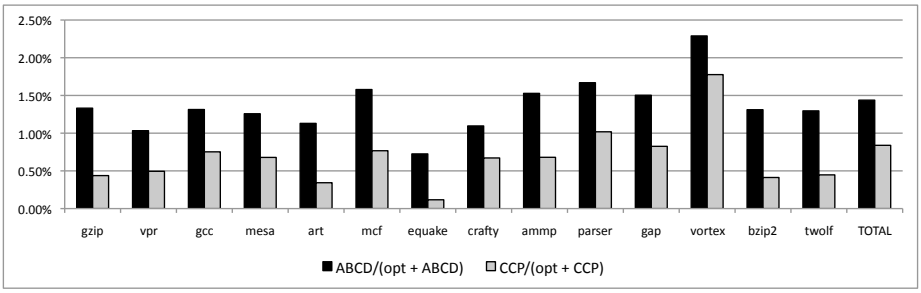


Figure 5.14: Execution time of two different live range splitting strategies compared to the total time taken by machine independent LLVM optimization passes (`opt`). 100% is the total time taken by `opt`. The shorter the bar, the faster the conversion.

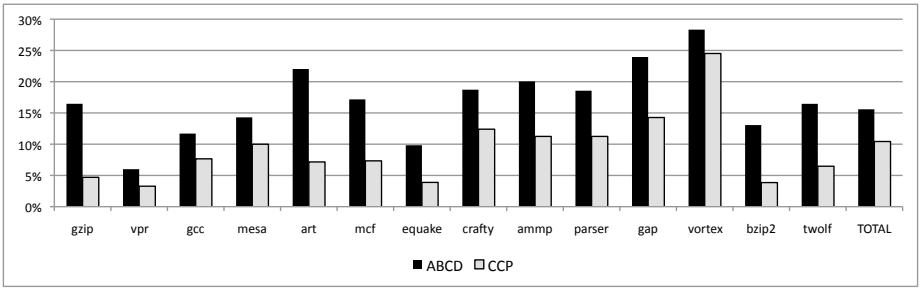


Figure 5.15: Number of ϕ and σ-functions produced by different live range splitting strategies. 100% is the number of instructions inserted by the SSI conversion.

5.4.2 Space

Figure 5.15 compares the number of ϕ and σ-functions inserted by each live range splitting strategy. The bars are the sum of these instructions, as inserted by each conversion, divided by the number of σ and ϕ-functions inserted by the SSI live range splitting strategy. The CCP client created 67.3K σ-functions, and 28.4K ϕ-functions. The ABCD client created 98.8K σ-functions, and 42.0K ϕ-functions. The SSI conversion inserted 697.6K σ-functions, and 220.6K σ-functions.

The chart in Figure 5.16 shows the number of σ and ϕ-functions that each live range splitting strategy inserts per variable. The denominator includes only variables that have lead to the creation of special instructions. That is, variables that are live only inside one basic block are not taken into consideration. The figure emphasizes the difference between the conversion required by the two forward analyses and the SSI conversion. On the average, for each variable whose conversion is requested by either the ABCD or the CCP client, we will create 0.6 ϕ-functions, and 1.3 σ-functions. On the other hand, SSI will insert 6.1 σ-functions and 2.7 ϕ-functions per variable.

Finally, Figure 5.17 outlines how much each live range splitting strategy increases program size. We show results only to the ABCD and CCP clients, to keep the chart easy to read. The SSI conversion increases program size in 17.6% on average. This is an absolute value, i.e., we sum up every ϕ and σ function inserted, and divide

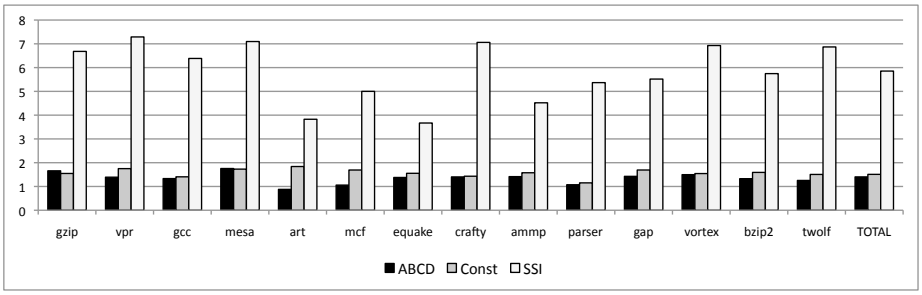


Figure 5.16: Average number of ϕ and σ-functions produced per variable.

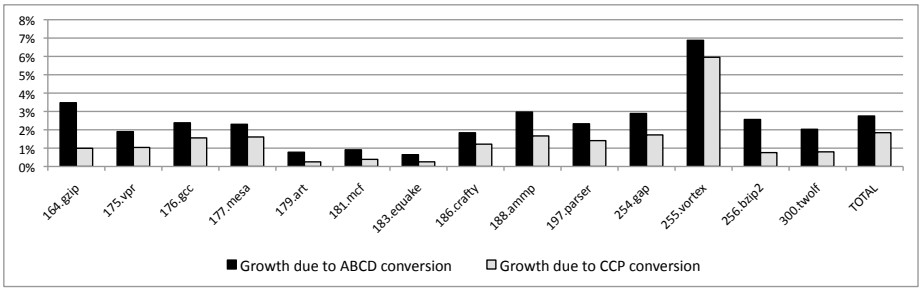


Figure 5.17: Growth in program size due to the insertion of new ϕ and σ functions to perform live range splitting.

it by the number of bytecode instructions in the original program. This compiler already uses the SSA-form by default, and we do not count as new instructions the ϕ-functions originally used in the program. The ABCD client increases program size by 2.75%, and the CCP client increases program size by 1.84%.

An interesting question that deserves attention is "What is the benefit of using a sparse data-flow analysis in practice?" We have not implemented dense versions of the ABCD or the CCP clients. However, previous works have shown that sparse analyses tend to outperform equivalent dense versions in terms of time and space efficiency [44; 117]. In particular, the e-SSA format used by the ABCD and the CCP optimizations is the same program representation adopted by the tainted flow framework of Rimsa *et al.* [120], which has been shown to be faster than a dense implementation of the analysis, even taking the time to perform live range splitting into consideration.

5.5 Conclusion

This chapter has presented a systematic way to build program representations that suit data-flow analyses. We build different program representations by splitting the live ranges of variables. The way in which we split live ranges depends on two factors. First, which program points produce new information, e.g., uses, definitions, tests, etc. Second, how this information propagates along the variable live range: forwardly or backwardly. We have used an implementation of our framework in

LLVM to convert programs to the Static Single Information form [6], and to provide intermediate representations to the ABCD array bounds-check elimination algorithm [16] and to Wegman *et al.*'s Conditional Constant Propagation algorithm [139]. This very implementation has been used by Couto *et al.* [59] to provide the program representation required to implement Gawlitza *et al.*'s [67] range analysis algorithm. We have also used our live range splitting algorithm, implemented in the phc PHP compiler [15], to provide the Extended Static Single Assignment form necessary to solve the tainted flow problem [120].

5.6 Appendix. Isomorphism to Sparse Evaluation Graphs

Given a control flow graph G, Choi *et al.* define a sparse evaluation graph as a tuple $\langle N_{SG}, E_{SG}, M\rangle$, such that:

- N_{SG} is a set of nodes defined as follows:
  1. N_{SG} contains a node n_s representing the entry point $s \in G$;
  2. N_{SG} contains a node n_p for each point $p \in G$ that is associated with a non-identity transfer function.
  3. N_{SG} contains a node n_m for each point m in the iterated dominance frontier of the points of G used to build the nodes in step (1) and (2). These are called *meet* nodes.
- We let P denote the set of points $p \in G$ used in step 2 above, plus the point $s \in G$ used in step 1 above; we let M denote the set of points $m \in G$ used in step 3 above; if we let $S = P \bigcup M$ then we define E_{SG} as follows:
  1. there is an edge $(n_q, n_m) \in N_{SG}^2$ whenever $m \in M$ and q is, among all the nodes in S, the immediate dominator of one of the CFG predecessors of m. See `search(3b)` and `link(2b)` in Choi *et al* [44];
  2. there is an edge $(n_q, n_p) \in N_{SG}^2$ whenever $p \in P$, and q is, among all the nodes in S, the immediate dominator of p. See `search(1)` and `link(2b)` [44];
- The mapping function $M : E_G \mapsto N_{SG}$ associates to each edge (u, v) of the CFG the node $n_q \in N_{SG}$, whenever $q \in S$ is the immediate dominator of $u \in G$. See `search(3a)` [44]. This is done through the recursive function `search` that performs a topological traversal of the CFG (DFS of the dominance tree; See `search(4)` [44]).

Theorem 5.6 states that, for forward partitioned variable data-flow problems (PVP), the algorithm in Figure 5.8 can build program representations isomorphic to Sparse Evaluation Graphs. The proof that this result holds for backward data-flow problems, is analogous, and we omit it.

Lemma 5.5 (CFG cover)**.** *Let `Prog` be a program with its corresponding CFG G with start node s, and exit node x. Let `Prog`′ be the program that we obtain from `Prog` by:*

1. *adding a pseudo-definition of each variable to s;*
2. *adding a pseudo-use of each variable to x;*
3. *placing a pseudo-use of a variable v at each point where v is defined;*

4. *converting the resulting program into SSA form.*

If v is a variable in $Prog$, then the live ranges of the different names of v in $Prog'$ completely partition the program points of G. In other words, each program point of G belongs to exactly one live range of v in $Prog'$.

Proof. First, v is alive at every point of G, due to transformations (1), (2) and (3). Therefore, if V is the set of the different names of v after the conversion to SSA form in step (4), then any program point of G belongs to the live range of at least one $v' \in V$. The result follows from a well-know property of Cytron's SSA-form conversion algorithm [55], which, as observed by Sreedhar *et al.* [130], creates variables with non-intersecting live ranges. In other words, after the SSA renaming, two different names of v cannot be simultaneously alive at a program point p. □

Theorem 5.6 (Equivalence SSI/SEG)**.** *Given a forward Sparse Evaluation Graph (SEG) that represents a variable v in a program representation $Prog$ with CFG G, there exits a live range splitting strategy that once applied on v builds a program representation that is isomorphic to SEG.*

Proof. We argue that the SEG of v is isomorphic to the representation of v in $Prog'$, the program representation that we derive from $Prog$ by applying the transformations 1-3 listed in Lemma 5.5 in addition to a pass of SSIfy. If we let P, as before, be defined as the set of CFG points associated with non-identity transfer functions, plus the start node s of the CFG, then after we apply the splitting strategy $P_!$, we have that:

1. there will be exactly one definition per node of P and one definition per node of $DF^+(P)$. So there is an one-to-one correspondence between SSA definitions and SG nodes.

2. From Lemma 5.5 the live-ranges of the different names of v provides a partitioning of the points of G. If v' is a new name of v, then each program point where v' is alive is dominated by v''s definition[5]. Each program point belongs to the live-range of the name of v whose definition immediately dominates it (among all definitions). Thus, live ranges give origin to a function that maps SSA definitions to program points. Consequently, there is an isomorphism between the live-ranges and the mapping function M.

3. def-use chains on $Prog'$ are isomorphic to the edges in E_{SG}: indeed a SEG node n_p is linked to n_q whenever (i) n_p immediately dominates n_q if $q \in P$; or (ii) n_q is in the dominance frontier of n_p if $q \in M$. In the former case the definition of v at p reaches the (pseudo-)use of v at q. In the latter this definition reaches the use of v at the ϕ-function placed at q by SSIfy$(v, P_!)$.

□

In the proof of Theorem 5.6 we had to augment the program with a pseudo-definition of v at the CFG's entry point and a pseudo-use at every actual definition of v and at the CFG's exit point. The difference between a code with or without pseudo uses/defs is related to the necessity to compute data-flow information beyond the live-ranges of variables or not. This necessity exists for optimizations such as partial redundancy elimination, which may move, create or delete code.

[5] This is a classical result of SSA-form. See Budimlic *et al.* [39] for a proof

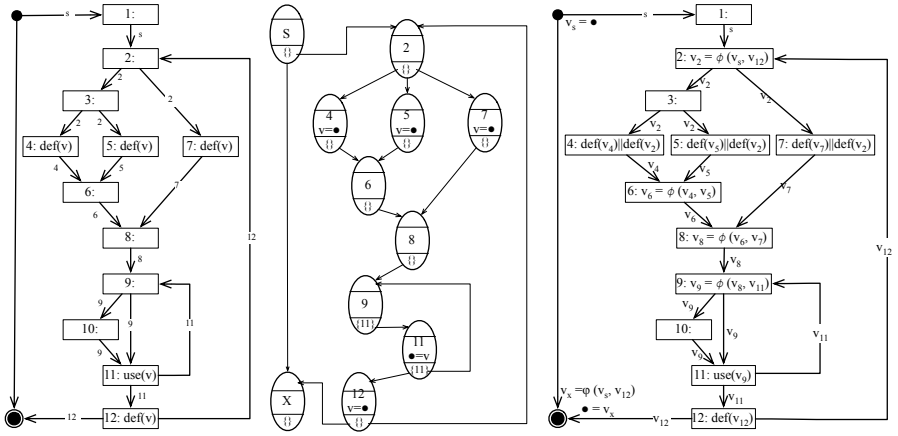


Figure 5.18: Example of equivalence between SEGs and our live range splitting strategy for reaching uses.

Figure 5.18 compares SEG and the forward live range splitting strategy in the example taken from Figure 11 of Choi *et al.* [44], which shows the reaching uses analysis. In the left we see the original program, and in the middle the SEG built for a forward flow analysis that extracts information from uses of variables. We have augmented the edges in the left CFG with the mapping M of SEG nodes to CFG edges. In the right we see the same CFG, augmented with pseudo defs and uses, after been transformed by SSIfy applied on the points $\{S,4,5,7,11,12\}_{\downarrow}$. The edges of this CFG are labeled with the definitions of v live there.

5.7 Appendix. Correctness of our SSIfication

In this section we consider a unidirectional forward (resp. backward) PVL problem stated as a set of equations $[v]^i = [v]^i \wedge F_v^s(\ldots)$ for every variable v, each program point i, and each $s \in pred(i)$ (resp. $s \in succ(i)$). We rely on the concepts introduced by Definition 5.7 in order to prove Theorem 5.11.

Definition 5.7 (Trivial/constant transfer functions. Dependencies)**.** Let $\mathscr{L}_{v_1} \times \mathscr{L}_{v_2} \times \cdots \times \mathscr{L}_{v_n}$ be the decomposition per variable of lattice $\mathscr{L}$, where $\mathscr{L}_{v_i}$ is the lattice associated with variable v_i. Let F_v be a transfer function from $\mathscr{L}$ to $\mathscr{L}_v$. We say that F_v is *trivial* if:

$$\forall x = ([v_1], \ldots, [v_n]) \in \mathscr{L},\ F_v(x) = x_v$$

We say that F_v is *constant with value* $C \in \mathscr{L}$ if:

$$\forall x \in \mathscr{L},\ F_v(x) = C$$

If F_v is constant with value $\bot$, e.g., $F_v(x) = \bot$, then we say that F_v is *undefined*.
We say that F_v *depends on variable* v_j if:

$$\begin{array}{l} \exists x = ([v_1], \ldots, [v_n]) \neq ([v_1]', \ldots, [v_n]') = x' \text{ in } \mathscr{L} \\ \text{such that } \left[\forall k \neq j,\ [v_k] = [v_k]' \wedge F_i(x) \neq F_i(x')\right] \end{array}$$

Lemma 5.8 (Live range preservation)**.** *If variable v is live at a program point i, then there is a version of v live at i after we run* SSIfy.

Proof. Split cannot remove any live range of v, as it only inserts "copies" from v to v, e.g., each copy has the same source and destination. Rename removes live ranges of v, but it replaces them with the live ranges of new versions of this variable whenever a use of v is renamed. Clean only removes "copies"; hence, all the original instructions remain in the code. □ □

Lemma 5.9 (Non-Overlapping)**.** *Two different versions of v, e.g., v_j and v_k cannot be simultaneously live at a program point i transformed by* SSIfy.

Proof. The only algorithm that creates new versions of v is rename. Each new version of v is unique, as we ensure in line 27-29 of the algorithm. If rename changes the use of v to v_k at i, then there exists a definition of v_k at some program point i' that dominates i, as we ensure in line 22 of the algorithm. Lets assume that we have two versions of v, e.g., v_k and v_j, live at a program point i, in order to derive a contradiction. In this case, there exist program points i_i where v_k is used, and i_j where v_j is used, reachable from i. There exists also a program point i'_i where v_k is defined, and a program point i'_j where v_j is defined, so that i'_i dominates i_i, and i'_j dominates i_j. Now, if neither i'_i dominates i'_j nor vice-versa, then we have a contradiction, because, given that i'_i reaches i_j and i'_j reaches i_i, then neither i'_i would dominates i_i, nor i'_j would dominates i_j. Without loss of generality, lets assume that i'_i dominates i'_j. In this case, rename visits i'_i first, and upon visiting i'_j, places the definition of v_j on top of the definition of v_k in the stack in line 30. Thus, i'_j cannot dominate i_i, or we would have, at that program point, a use of v_j, instead of v_k. In this case, i_j is live past the dominance frontier of i'_i, forcing split (line 14) to create a ϕ-function that dominates i_i, at a program point that is dominated by i'_i; hence, creating a new definition v_ϕ of v. Therefore, at i_i we would have a use of v_ϕ instead of v. □ □

Theorem 5.10 (Semantics)**.** SSIfy *maintains the following property: if a value n written to variable v at program point i' is read at a program point i in the original program, then the same value assigned to a version of variable v at program point i' is read at program point i after transformation.*

Proof. For simplicity, we will extend the meaning of "copy" to include not only the parallel copies placed at interior nodes, but also ϕ and σ-functions. Split cannot create new values, as it only inserts "copies". Clean cannot remove values, as it only removes "copies". From the hypothesis we know that the definition of v that reaches i is live at i. From Lemma 5.8 we know that there is a version of v live at i. From Lemma 5.9 we know that only one version of v can be live at i, and so rename cannot send new values to i. □ □

Now suppose that the program, not necessarily under SSI form, fulfills INFO and LINK as defined in Property 5.2 for a system of monotone equations E_{dense}, given as a set of constraints $[v]^i \sqsubseteq F^s_v([v_1]^s, \ldots, [v_n]^s)$. Consider a live range splitting strategy $\mathscr{P}_v$ that *includes* for each variable v the set of program points $I_\downarrow$ (resp. $I_\uparrow$) where F^s_v is non-trivial. The following theorem states that Algorithm SSIfy creates a program form that fulfills the Static Single Information property.

Theorem 5.11 (Correctness of SSIfy)**.** *Given the conditions stated above, Algorithm* SSIfy*$(v, \mathscr{P}_v)$ creates a new program representation such that:*

1. *there exists a system of equations E^{ssi}_{dense}, isomorphic to E_{dense} for which the new program representation fulfills the SSI property.*

2. *if E_{dense} is monotone then E^{ssi}_{dense} is also monotone.*

Proof. We derive from this new program representation a system of equations isomorphic to the initial one by associating trivial transfer functions with the newly created "copies". The INFO and LINK properties are trivially maintained. As only trivial and constant functions have been added, monotonicity is maintained.

To show that we provide SPLIT, we must first show that each $i \in \text{live}(v)$ where F^s_v is non-trivial leads to a definition (resp. last use) of v at s. The function split separates these points in lines 9 and 16, and later, in line 23, inserts definitions in those points. Second, we must show that each join (resp. split) node for which E_{dense} has possibly different values on its incoming edges should have a ϕ-function (resp. σ-function) for v. These points are separated in lines 7 and 14 of split. To see why this is the case, notice that line 7 separates the points in the iterated dominance frontier of points that originate information that flows forward. These are, as a direct consequence of the definition of iterated dominance frontier, the points where information collide. Similarly, line 14 separates the points in the post-dominance frontier of regions which originate information that flows backwardly.

We ensure VERSION as a consequence of the SSA conversion. All our program representations preserve the SSA representation, as we include the definition sites of v in line 11 of split. Function rename ensures the existence of only one definition of each variable in the program code (line 27), and that each definition dominates all its uses (consequence of the traversal order). Therefore, the newly created live ranges are connected on the dominance tree of the source program. Function rename also creates a new program representation for which it is straightforward to build a system of equations E^{ssi}_{dense} isomorphic to E_{dense}: Firstly, the constraint variables are renamed in the same way that program variables are. Secondly, for each program variable, new system variables bound to $\perp$ are created for each program point outside of its live-range. □

□

5.8 Equivalence between sparse and dense analyses.

We have shown that SSIfy transforms a program P into another program P^{ssi} with the same semantics. Furthermore, this representation provides the SSI property for a system of equations E^{ssi}_{dense} that we extract from P^{ssi}. This system is isomorphic to the system of equations E_{dense} that we extract from P. From the so obtained program under SSI for the constrained system E^{ssi}_{dense}, Definition 5.3 shows how to construct a sparse constrained system E^{ssi}_{sparse}. When transfer functions are monotone and the lattice has finite height, Theorem 5.4 states the equivalence between the sparse and the dense systems. The purpose of this section is to prove this theorem. We start by introducing the notion of *coalescing*. Let E be a constraint system that associates with each $1 \leq i \leq n$ the constraint $a_i \sqsubseteq H_i(a_1, \ldots, a_n)$, where each a_i is an element of a lattice $\mathscr{L}$ of finite height, and H_i is a monotone function from $\mathscr{L}^n$ to $\mathscr{L}$. Let $(A_1, \ldots, A_n)$ be the maximum solution to this system, and let $1 \leq m \leq n$ such that $\forall i, 1 \leq i \leq m, A_i = A_m$. We define a "coalesced" con-

straint system E_{coal} in the following way: for each $1 \leq i \leq m$ we create the constraint $b_m \sqsubseteq H_i(b_m,\ldots,b_m,b_{m+1},\ldots,b_n)$; for each $m < i \leq n$ we create the constraint $b_i \sqsubseteq H_i(b_m,\ldots,b_m,b_{m+1},\ldots,b_n)$. Lemma 5.12 shows that coalescing preserves the maximum solution of the original system.

Lemma 5.12 (Equivalence with coalescing)**.** *If E is a constraint system with maximum solution $(A_1,\ldots,A_m,\ldots,A_n)$; if for any i,j such that $1 \leq i,j \leq m$ we have that $A_i = A_j$; and if E_{coal} is the "coalesced" system that we derive from E by coalescing $a_1 \ldots a_m$. Then the maximum solution of E_{coal} is $(A_m,\ldots,A_n)$.*

Proof. Both system have a (unique) maximum solution (see e.g. [103]), although the solution of the "coalesced" system has smaller cardinality, i.e.., $n-m+1$. Now, as $(A_m,\ldots,A_m,A_{m+1},\ldots,A_n)$ is a solution to E, by definition of E_{coal}, $(A_m,\ldots,A_n)$ is a solution to E_{coal}. Let us prove that this solution is maximum, i.e. for any solution $(B_m,\ldots,B_n)$ of E_{coal}, we have $(B_m,\ldots,B_n) \sqsubseteq (A_m,\ldots,A_n)$. By definition of E_{coal}, we have that $(B_m,\ldots,B_m,B_{m+1},\ldots,B_n)$ is a solution to E. As $(A_1,\ldots,A_n)$ is maximum, we have $(B_m,\ldots,B_m,B_{m+1},\ldots,B_n) \sqsubseteq (A_1,\ldots,A_n)$. So $(B_m,\ldots,B_n) \sqsubseteq (A_m,\ldots,A_n)$. □
□

We now prove Theorem 5.4, which states that there exists a direct mapping between the maximum solution of a dense constraint system associated with a SSI-form program, and the sparse system that we can derive from it, according to Definition 5.3.

Proof. The constraint systems E^{ssi}_{dense} and E^{ssi}_{sparse} have a maximum unique solution, because the transfer functions are monotone and $\mathscr{L}$ has finite height

The idea of the proof is to modify the constraint system E^{ssi}_{dense} into a system equivalent to E^{ssi}_{sparse}. To accomplish this transformation, we (i) replace each F^s_v by G^s_v, where G^s_v is constructed as in Definition 5.3; (ii) for each v, coalesce $\left([v]^i\right)_{i \in live(v)}$ into $[v]$; (iii) coalesce all other constraint variables into $[v_\perp]$.

The LINK property allows us to replace F^s_v by G^s_v. Due to SPLIT, a new variable is defined at each point where information is generated, and due to VERSION there is only one live range associated with each variable. Hence, $([v]^i)_{i \in live(v)}$ is invariant. Due to INFO, we have that $([v]^i)_{i \notin live(v)}$ is bound to $\perp$. Due to Lemma 5.12, we know that this new constraint system has a maximum solution $(Y_v)_{v \in variables \cup \perp}$: X^i_v equals Y_v for all $i \in live(v)$, and $Y_\perp$ otherwise.

Each constraint $[v]^i \sqsubseteq F^s_v([v_1]^s,\ldots,[v_n]^s)$, in the original system, translates to a constraint in the "coalesced" one in the following way:

$$\begin{cases} \text{if } i \in \mathsf{live}(v): & \text{if } s \in \mathsf{defs}(v) & : [v] \sqsubseteq G^s_v([a],\ldots,[b]) & (1) \\ & \text{else} & : [v] \sqsubseteq [v] & (2) \\ \text{otherwise} & & : [v_\perp] \sqsubseteq \perp & (3) \end{cases}$$

Case (1) follows from LINK, case (2) follows from SPLIT, and case (3) follows from INFO. By ignoring $y_\perp$ that appears only in (3), and by removing the constraints produced by (2), which are useless, we obtain E^{ssi}_{sparse}. □

Bibliography

[1] W. B. Ackerman. *Efficient Implementation of Applicative Languages*. PhD thesis, MIT, 1984.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[3] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–146, 1976.

[4] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5:1–19, 1970.

[5] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *15th Symposium on Principles of Programming Languages (POPL'88)*, pages 1–11. ACM, 1988.

[6] Scott Ananian. The static single information form. Master's thesis, MIT, September 1999.

[7] Fernando Magno Quint ao Pereira and Jens Palsberg. Register allocation by puzzle solving. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 216–226, New York, NY, USA, 2008. ACM.

[8] Andrew w. Appel. *Modern compiler implementation in ML*. Cambridge University Press, 1998.

[9] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, pages 243–253. ACM Press, 2001.

[10] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, second edition, 2002.

[11] Thomas Ball and James R. Larus. Branch prediction for free. *SIGPLAN Notices*, 28(6):300–313, 1993.

[12] R. Barik. *Efficient optimization of memory accesses in parallel programs*. PhD thesis, Rice University, 2009.

[13] Rajkishore Barik, Christian Grothoff, Rahul Gupta, Vinayaka Pandit, and Raghavendra Udupa. Optimal bitwise register allocation using integer linear programming. In *LCPC*, volume 4382 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2006.

[14] Michael Bender and Martín Farach-Colton. The LCA problem revisited. In Gastón Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.

[15] Paul Biggar. *Design and Implementation of an Ahead-of-Time Compiler for PHP*. PhD thesis, Trinity College Dublin, 2009.

[16] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.

[17] B. Boissinot, A. Darte, B. Dupont de Dinechin, C. Guillon, and F. Rastello. Revisiting out-of-SSA translation for correctness, code quality, and efficiency. In *International Symposium on Code Generation and Optimization (CGO'09)*, pages 114–125. IEEE Computer Society Press, March 2009.

[18] Benoit Boissinot. *Towards an SSA-Based Compiler Back-End: Some Interesting Properties of SSA and its Extensions*. PhD thesis, ENS-Lyon, September 2010.

[19] Benoit Boissinot, Florian Brandner, Alain Darte, Benoit Dupont de Dinechin, and Fabrice Rastello. A non-iterative data-flow algorithm for computing liveness sets in strict ssa programs. In *9th Asian Symposium on Programming Languages and Systems (APLAS'11)*. Springer Verlag, December 2011.

[20] Benoit Boissinot, Philip Brisk, Alain Darte, and Fabrice Rastello. SSI properties revisited. *ACM Transactions on Embedded Computing Systems*, 2010. Special Issue on Software and Compilers for Embedded Systems.

[21] Benoit Boissinot, Alain Darte, Benoît Dupont de Dinechin, Christophe Guillon, and Fabrice Rastello. Revisiting out-of-SSA translation for correctness, code quality, and efficiency. In *International Symposium on Code Generation and Optimization (CGO'09)*, pages 114–125. IEEE Computer Society Press, March 2009. Best paper award.

[22] Benoit Boissinot, Sebastian Hack, Daniel Grund, Benoît Dupont de Dinechin, and Fabrice Rastello. Fast liveness checking for SSA-form programs. In *CGO'08: proceedings of the sixth annual ieee/acm international symposium on code generation and optimization*, pages 35–44, New York, NY, USA, 2008. ACM Press.

[23] Florent Bouchez. *A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases*. PhD thesis, ENS Lyon, France, apr 2009.

[24] Florent Bouchez, Quentin Colombet, Alain Darte, Christophe Guillon, and Fabrice Rastello. Parallel copy motion. In *13th International Workshop on Software & Compilers for Embedded Systems (SCOPES'10)*, pages 1–10, St. Goar, Germany, June 2010. ACM Press.

[25] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation and spill complexity under SSA. Technical Report RR2005-33, LIP, ENS-Lyon, France, August 2005.

[26] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the NP-completeness proof of chaitin et al. really prove? In *Workshop on Duplicating, Deconstructing and Debunking (WDDD'06), held in conjunction with the International Symposium on Computer Architecture (ISCA'33)*, July 2006.

[27] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? or revisiting register allocation: Why and how. In *19th International Workshop on Languages and Compilers for Parallel Computing (LCPC'06)*, volume 4382 of *LNCS*, pages 283–298, New Orleans, USA, November 2006. Springer Verlag.

[28] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 102–114, Washington, DC, USA, mar 2007. IEEE Computer Society Press. Best paper award.

[29] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of spill everywhere under ssa form. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, pages 103 – 112. ACM Press, 2007.

[30] Florent Bouchez, Alain Darte, and Fabrice Rastello. Advanced conservative and optimistic register coalescing. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'08)*, pages 147–156. ACM Press, 2008.

[31] Florian Brandner and Quentin Colombet. Copy elimination on data dependence graphs. In *Symposium on Applied Computing (SAC'12)*. Elsevier, 2012. To appear.

[32] Matthias Braun and Sebastian Hack. Register Spilling and Live-Range Splitting for SSA-Form Programs. In *Compiler Construction 2009*, volume 5501 of *Lecture Notes In Computer Science*, pages 174–189. Springer, 2009.

[33] Matthias Braun, Christoph Mallon, and Sebastian Hack. Preference-Guided Register Assignment. In *Compiler Construction 2010*, volume 6011 of *Lecture Notes In Computer Science*, pages 205–223. Springer, 2010.

[34] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software — Practice and Experience*, 28(8):859–881, July 1998.

[35] Preston Briggs. *Register allocation via graph coloring*. PhD thesis, Rice university, April 1992.

[36] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, May 1994.

[37] Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2:59–69, March 1993.

[38] Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial-time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*. ACM Press, 2005.

[39] Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast copy coalescing and live-range identification. In *PLDI*, pages 25–32. ACM, 2002.

[40] Robert Cartwright and Mattias Felleisen. The semantics of program dependence. *SIGPLAN Not.*, 24(7):13–27, 1989.

[41] G. J. Chaitin. Register allocation and spilling via graph coloring. *Symposium on Compiler Construction*, 17(6):98–105, 1982.

[42] Gregory J. Chaitin, Mark A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[43] Craig Chambers and David Ungar. Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. *SIGPLAN Not.*, 24(7):146–160, 1989.

[44] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL*, pages 55–66. ACM, 1991.

[45] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(4):501–536, Oct. 1990.

[46] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. In *ACM SIGPLAN Workshop on Intermediate Representations*, pages 35–49. ACM, January 1995.

[47] Jean-Francois Collard. *Reasoning about Program Transformations*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[48] Quentin Colombet, Benoit Boissinot, Philip Brisk, Sebastian Hack, and Fabrice Rastello. Graph coloring and treescan register allocation using repairing. In *International Conference on Compilers, Architectures, and Synthesis of Embedded Systems (CASES'11)*. IEEE Computer Society, October 2011.

[49] Quentin Colombet, Florian Brandner, and Alain Darte. Studying optimal spilling in the light of ssa. In *International Conference on Compilers, Architectures, and Synthesis of Embedded Systems (CASES'11)*, Taipei, Taiwan, October 2011. IEEE Computer Society.

[50] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[51] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. An empirical study of iterative data-flow analysis. In *15th International Conference on Computing (ICC'06)*, pages 266–276, Washington, DC, USA, 2006. IEEE Computer Society.

[52] Ron Cytron and Jeanne Ferrante. What's in a name? Or the value of renaming for parallelism detection and storage allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27. IEEE Computer Society Press, August 1987.

[53] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *POPL*, pages 25–35, 1989.

[54] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, October 1991.

[55] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[56] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, New York, NY, USA, 1982. ACM.

[57] Boubacar Diouf, Albert Cohen, and Fabrice Rastello. A polynomial spilling heuristic: Layered allocation. In *International Symposium on Code Generation and Optimization (CGO'13)*. IEEE Computer Society Press, February 2013. to be published.

[58] Boubacar Diouf, Albert Cohen, Fabrice Rastello, and John Cavazos. Split register allocation: Linear complexity without the performance penalty. In *International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC'10)*, volume 5952 of *Lecture Notes in Computer Science*, pages 66–80. Springer Verlag, January 2010.

[59] Douglas do Couto Teixeira and Fernando Magno Quintao Pereira. The design and implementation of a non-iterative range analysis algorithm on a production compiler. In *SBLP*, pages 45–59. SBC, 2011.

[60] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Reducing the cost of data flow analysis by congruence partitioning. In *Compiler Construction, 5th International Conference (CC'94)*, volume 786 of *Lecture Notes in Computer Science*, pages 357–373. Springer, 1994.

[61] Benoît Dupont de Dinechin, Fran cois de Ferrière, Christophe Guillon, and Arthur Stoutchinin. Code generator optimizations for the ST120 DSP-MCU core. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'00)*, pages 93 – 103. ACM Press, 2000.

[62] Benoît Dupont de Dinechin, Christophe Monat, and Fabrice Rastello. Parallel execution of the saturated reductions. In *Workshop on Signal Processing Systems (SIPS'2001)*, pages 373–384. IEEE Computer Society Press, 2001.

[63] Janet Fabri. Automatic storage optimization. In *Proceedings of the SIGPLAN symposium on Compiler construction (CC'79)*, pages 83–91. ACM Press, 1979.

[64] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 203–213. ACM, June 2000.

[65] Martin Farach-Colton and Vincenzo Liberatore. On local register allocation. *Journal of Algorithms*, 37(1):37–65, 2000.

[66] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103, 1996.

[67] Thomas Gawlitza, Jerome Leroux, Jan Reineke, Helmut Seidl, Gregoire Sutre, and Reinhard Wilhelm. Polynomial precise interval analysis revisited. *Efficient Algorithms*, 1:422 – 437, 2009.

[68] Lal George and Andrew W. Appel. Iterated register coalescing. *TOPLAS*, 18(3):300–324, 1996.

[69] Lal George and Blu Matthias. Taming the ixp network processor. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation (PLDI'03)*, pages 26–37. ACM, 2003.

[70] M. Gerlek, M. Wolfe, and E. Stoltz. A Reference Chain Approach for Live Variables. Technical Report CSE 94-029, Oregon Graduate Institute of Science & Technology, 1994.

[71] Christophe Guillon, Fabrice Rastello, Thierry Bidault, and Florent Bouchez. Procedure placement using temporal-ordering information: Dealing with code size expansion. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04)*, pages 268–27. ACM Press, April 2004.

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

[73] Sebastian Hack and Gerhard Goos. Copy coalescing by graph recoloring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 227–237. ACM, 2008.

[74] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *International Conference on Compiler Construction (CC'06)*, volume 3923 of *LNCS*, pages 247–262. Springer-Verlag, March 2006.

[75] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, May 1984.

[76] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems*, 19(4):557–567, 1997.

[77] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, July 1974.

[78] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, 1977.

[79] Matthew S. Hecht and Jeffrey D. Ullman. Analysis of a simple algorithm for global data flow problems. In *1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'73)*, pages 207–217, New York, NY, USA, 1973. ACM.

[80] Jong hoon An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for ruby. In *POPL*, pages 459–472. ACM, 2011.

[81] Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. *ACM Transactions on Programming Languages and Systems*, 19:1031–1052, November 1997.

[82] R. Johnson, D. Pearson, and K. Pingali. The program tree structure. In *PLDI*, pages 171–185. ACM, 1994.

[83] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *PLDI*, pages 78–89. ACM, 1993.

[84] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976.

[85] K. W. Kennedy. Node listings applied to data flow analysis. In *2nd ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL'75)*, pages 10–21. ACM, 1975.

[86] Gary A. Kildall. A unified approach to global program optimization. In *Symposium on Principles of Programming Languages (POPL'73)*, pages 194–206. ACM Press, 1973.

[87] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5:1–32, May 2008.

[88] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.

[89] Allen Leung and Lal George. Static single assignment form for machine code. In *International Conference on Programming Language Design and Implementation (PLDI'99)*, pages 204–214. ACM Press, 1999.

[90] Lian Li, Jingling Xue, and Jens Knoop. Scratchpad memory allocation for data aggregates via interval coloring in superperfect graphs. *ACM Trans. Embedded Comput. Syst.*, 10(2):28, 2010.

[91] LibFirm: A library that provides an intermediate representation and optimisations for compilers. `http://pp.info.uni-karlsruhe.de/firm`.

[92] LLVM: The LLVM compiler infrastructure. `http://llvm.org`.

[93] R. Lo, F. Chow, R. Kennedy, S. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Conference on Programming Language Design and Implementation (PLDI'(*)*, pages 26–37, 1998.

[94] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation (PLDI'98)*, pages 26–37. ACM, 1998.

[95] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(11):1355–1371, Nov 2001.

[96] Cathy May. The parallel assignment problem redefined. *IEEE Trans. Software Eng.*, 15(6):821–824, 1989.

[97] David McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49:512–537, July 2002.

[98] Antoine Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19:31–100, 2006.

[99] Michel Minoux. LTUR: A simplified linear-time unit resolution algorithm for Horn formulae and computer implementation. *Information Processing Letters*, 29:1–12, September 1988.

[100] Mono: Cross platform, open source .NET development framework. `http://www.mono-project.com`.

[101] Hanspeter Mössenböck and Michael Pfeiffer. Linear scan register allocation in the context of SSA form and register constraints. In *International Conference on Compiler Construction (CC'02)*, volume 2304 of *LNCS*, pages 229–246. Springer, 2002.

[102] Mangala Gowri Nanda and Saurabh Sinha. Accurate interprocedural null-dereference analysis for java. In *ICSE*, pages 133–143, 2009.

[103] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2005.

[104] Rei Odaira, Takuya Nakaike, Tatsushi Inagaki, Hideaki Komatsu, and Toshio Nakatani. Coloring-based coalescing for graph coloring register allocation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 160–169, New York, NY, USA, 2010. ACM.

[105] David A. Padua. Parallelization, automatic. In *Encyclopedia of Parallel Computing*, pages 1442–1450. 2011.

[106] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT'98)*, pages 196–204. IEEE Press, 1998.

[107] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 26(4), 2004.

[108] F. M. Q. Pereira and J. Palsberg. Register allocation via coloring of chordal graphs. In *Proceedings of Asian Symposium on Programming Languages and Systems (APLAS'05)*, volume 3780 of *LNCS*, pages 315–329. Springer, November 2005.

[109] Fernando Magno Quintao Pereira and Jens Palsberg. SSA elimination after register allocation. In *CC*, pages 158 – 173, 2009.

[110] Keshav Pingali and Gianfranco Bilardi. APT: A data structure for optimal control dependence computation. In *PLDI*, pages 211–222. ACM, 1995.

[111] Keshav Pingali and Gianfranco Bilardi. Optimal control dependence computation and the roman chariots problem. In *TOPLAS*, pages 462–491. ACM, 1997.

[112] John Bradley Plevyak. *Optimization of Object-Oriented and Concurrent Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

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

[114] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Eastern joint IRE-AIEE-ACM computer conference*, pages 133–138. ACM, 1959.

[115] G. Ramalingam. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems*, 21(2):175–188, March 1999.

[116] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Transactions on Programming Languages and Systems*, 24(5):455–490, 2002.

[117] G. Ramalingam. On sparse evaluation representations. *Theoretical Computer Science*, 277(1-2):119–147, 2002.

[118] Fabrice Rastello, François de Ferrière, and Christophe Guillon. Optimizing translation out of SSA using renaming constraints. In *International Symposium on Code Generation and Optimization (CGO'04)*, pages 265–278. IEEE Computer Society Press, 2004.

[119] Lakshminarayanan Renganarayana, U. Ramakrishna, and Sanjay Rajopadhye. Combined ilp and register tiling: Analytical model and optimization framework. In Eduard Ayguadé, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *Languages and Compilers for Parallel Computing*, volume 4339 of *Lecture Notes in Computer Science*, pages 244–258. Springer, 2006.

[120] Andrei Alves Rimsa, Marcelo D'Amorim, and Fernando M. Q. Pereira. Tainted flow analysis on e-SSA-form programs. In *CC*, pages 124–143. Springer, 2011.

[121] Hongbo Rong. Tree Register Allocation. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 67–77. ACM, 2009.

[122] Hongbo Rong, Alban Douillet, and Guang R. Gao. Register allocation for software pipelined multidimensional loops. *ACM Trans. Program. Lang. Syst.*, 30(4):23:1–23:68, August 2008.

[123] B. K. Rosen, F. K. Zadeck, and M. N. Wegman. Global value numbers and redundant computations. In *POPL*, pages 12–27. ACM Press, 1988.

[124] Subhajit Roy and Y. N. Srikant. The hot path ssa form: Extending the static single assignment form for speculative optimizations. In *CC*, pages 304–323, 2010.

[125] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[126] Vivek Sarkar and Rajkishore Barik. Extended linear scan: an alternate foundation for global register allocation. In *LCTES/CC*, pages 141–155. ACM, 2007.

[127] Bernhard Scholz, Chenyi Zhang, and Cristina Cifuentes. User-input dependence analysis via graph reachability. Technical report, Sun, Inc., 2008.

[128] Jeremy Singer. *Static Program Analysis Based on Virtual Register Renaming*. PhD thesis, University of Cambridge, 2006.

[129] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288. ACM, 2004.

[130] Vugranam C. Sreedhar, Roy Dz ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *SAS*, pages 194–210. Springer-Verlag, 1999.

[131] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. In *PLDI*, pages 108–120. ACM, 2000.

[132] Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In *TACAS*, pages 280–295, 2004.

[133] Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computer Science*, 345(1):122–138, 2005.

[134] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9(3):355–365, December 1974.

[135] André Tavares, Quentin Colombet, Mariza Bigonha, Christophe Guillon, Fernando M. Q. Pereira, and Fabrice Rastello. Decoupled graph-coloring register allocation with hierarchical aliasing. In *14th International Workshop on Software and Compilers for Embedded Systems (SCOPES'11)*, pages 1–10, St. Goar, Germany, 2011. ACM Press.

[136] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. *POPL*, pages 395–406, 2008.

[137] Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 142–151. ACM Press, 1998.

[138] W. T. Trotter. *Combinatorics and Partially Ordered Sets: Dimension Theory*. The John Hopkins University Press, Baltimore and London, 1992.

[139] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *TOPLAS*, 13(2), 1991.

[140] Michael Weiss. The transitive closure of control dependence: the iterated join. *TOPLAS*, 1(2):178–190, 1992.

[141] Christian Wimmer and Michael Franz. Linear scan register allocation on SSA form. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*, pages 170–179. ACM, 2010.

[142] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *1st ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 132–141, 2005.

[143] B. Yang, S. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman. Latte: A java vm just-in-time compiler with fast and efficient register allocation. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:128, 1999.

[144] Frank Kenneth Zadeck. *Incremental Data Flow Analysis in a Structured Program Editor*. PhD thesis, Rice University, 1984.