Revisiting Out-of-SSA Translation for Correctness, Code Quality, and Efficiency

Benoit Boissinot, Alain Darte, and Fabrice Rastello
Compsys team, LIP
UMR 5668 CNRS—ENS Lyon—UCB Lyon—Inria
Lyon, France
Email: firstname.lastname@ens-lyon.fr

Benoit Dupont de Dinechin and Christophe Guillon
CEC compiler group
STMicroelectronics
Grenoble, France
Email: firstname.lastname@st.com

***Abstract*—Static single assignment (SSA) form is an intermediate program representation in which many code optimizations can be performed with fast and easy-to-implement algorithms. However, some of these optimizations create situations where the SSA variables arising from the same original variable now have overlapping live ranges. This complicates the translation out of SSA code into standard code. There are three issues to consider: correctness, code quality (elimination of copies), and algorithm efficiency (speed and memory footprint). Briggs et al. proposed patches to correct the initial approach of Cytron et al. A cleaner and more general approach was proposed by Sreedhar et al., along with techniques to reduce the number of generated copies. We propose a new approach based on coalescing and a precise view of interferences, in which correctness and optimizations are separated. Our approach is provably correct and simpler to implement, with no patches or particular cases as in previous solutions, while reducing the number of generated copies. Also, experiments with SPEC CINT2000 show that it is 2x faster and 10x less memory-consuming than the Method III of Sreedhar et al., which makes it suitable for just-in-time compilation.**

I. I

SSA form [1] is a popular intermediate code representation used in modern compilers. Each variable is defined once and ϕ-functions are used to merge the values at join points of the control flow graph. The properties of the underlying dominance tree [2] and the implied use-def chains make possible the use of efficient, simple, and fast algorithms for various code optimizations in SSA. However, designing a correct algorithm and developing a bug-free implementation to go out of general SSA is not so easy, especially when taking into account critical edges, branches that define variables, register renaming constraints, and the natural semantics of ϕ-functions as parallel copies. Some compilers restrict SSA to CSSA [3] (conventional SSA), as going out of it is straightforward. In CSSA, all SSA variables connected (possibly by transitivity) by ϕ-functions have non-overlapping live ranges. They can thus be all replaced by the same name without changing the semantics of the code, like with code obtained just after SSA construction. However, restricting to CSSA means either disabling many SSA optimizations (as in GCC or Jikes [1]) with a potential loss in code quality, or pushing the burden on the SSA optimizations designer, who must guarantee that CSSA and register renaming constraints are correctly maintained. A general mechanism to go out of SSA is preferable. Correctness and ease of implementation are the first issues to address.

[1] http://jira.codehaus.org/browse/RVM-254

In addition to correctness, it is important to design fast algorithms for SSA construction and destruction, and not only for static compilation. Indeed, as optimizations in SSA are fast and powerful, SSA is increasingly used in just-in-time (JIT) compilers that operate on a high-level target-independent program representation such as Java byte-code, CLI byte-code (.NET MSIL), or LLVM bitcode. Most existing JIT compilers also save time by including only the essential tasks of code generation: instruction selection, flow analyzes, register allocation [4]. These tasks are complemented by the binary encoding and the link editing required for creating executable native code. Register allocation often relies on "linear scan" techniques [5], [6], [7], [8] in order to save compilation time and space by avoiding interference graphs. Similarly, instruction scheduling is usually reduced to post-pass scheduling [9]. Pre-pass scheduling is applied only where predicted or found beneficial [10], [11].

For SSA, it is important to consider with care the cost of its construction, the increase of the universe of variable names, and the cost of out-of-SSA translation. For the construction, simple and fast algorithms exist [12], [13]. It is also possible to encode SSA directly in byte-code, with an acceptable code size increase [14]. Unfortunately, increasing the number of variable names has a negative impact on the computation/storage of the liveness sets and the interference graph (if used), especially if the latter is implemented as a bit matrix to support fast queries. A naive translation out of SSA further increases the number of new variables and the code size, because ϕ-functions are replaced by variables and copies from/to these variables. A solution is to introduce copies on the fly and only when needed during the out-of-SSA translation, as in the Method III of Sreedhar et al. [3]. In order to eliminate copies or to avoid introducing them, some interference information, and thus some liveness information, is required. Budimlić et al. [15] proposed an out-of-SSA mechanism more suitable for JIT compilation, based on the notion of dominance forest, which reduces the number of interference tests and does not require an interference graph. Finally, fast liveness checking [16] for SSA can also be used to avoid the expensive computation of liveness sets by data-flow analysis. However, no solution proposed so far integrates all these optimizations.

In light of previous work, our primary goal was to design a new out-of-SSA translation, suitable for JIT compilation, thus focused on **speed** and **memory footprint** of the algorithms, as previous approaches were not fully satisfactory. However,

to make this possible, we had to revisit the way out-of-SSA translation is conceptually modeled. We then realized that our framework also addresses **correctness** of the translation and **quality** of the generated code. The next section defines the needed SSA concepts more precisely, motivates the need for revisiting out-of-SSA translation, and gives an overview of our method and its various options. Beforehand, here is a summary of the contributions of this paper:

Coalescing-based formulation We propose a conceptually simple approach for out-of-SSA translation based on "coalescing" (a term used in register allocation when merging two non-interfering live ranges). Thanks to this formulation, our technique is provably-correct, generic, easy to implement, and can benefit from register allocation techniques. In particular, we handle register renaming constraints (dedicated registers, calling conventions, etc.).

Value-based interferences A unique feature of our method is that we exploit the fact that SSA variables are uniquely defined, thus have only one value, to define a more accurate definition of interferences, generalizing the techniques of Chaitin et al. [17] and of Sreedhar et al. [3]. All our algorithms can be applied with the traditional interference definition as well as our value-based definition.

Parallel copies Our technique exploits the semantics of ϕ-functions, i.e., with parallel copies. This makes the implementation easier and gives more freedom for coalescing. At some point, parallel copies must be converted as sequences of copies. We designed an optimal algorithm (in terms of number of copies) for this sequentialization.

Linear intersection check During the algorithm, we need to check interferences between two sets of coalesced variables. We propose an algorithm, linear in the number of variables, while previous algorithms were quadratic.

Speed/memory optimizations To reduce memory footprint, we can avoid the need for explicit liveness sets and/or interference graph. Also, as in Method III of Sreedhar et al. [3], our algorithm can be adapted to insert copies on the fly, only when needed, to speed up the algorithm.

II. W - -SSA ?

The translation out of SSA has already been addressed, so why a new method? First, we want to rely on a provably-correct method, generic, simple to implement, without special cases and patches, and in which correctness and code quality (performance and size) are conceptually separated. Second, we need to develop a technique that can be fast and not too memory-consuming, without compromising correctness and code quality. Let us first go back to previous approaches.

Translation out of SSA was first mentioned by Cytron et al. [1, Page 478]: "Naively, a k-input ϕ-function at entrance of a node X can be replaced by k ordinary assignments, one at the end of each control flow predecessor of X. This is always correct, but these ordinary statements sometimes perform a good deal of useless work. If the naive replacement is preceded by dead code elimination and then followed by coloring, however, the resulting code is efficient". In other words, copies are placed in predecessor blocks to emulate the ϕ-function semantics and Chaitin-style coalescing [17] (as in register allocation) is used to remove some of them.

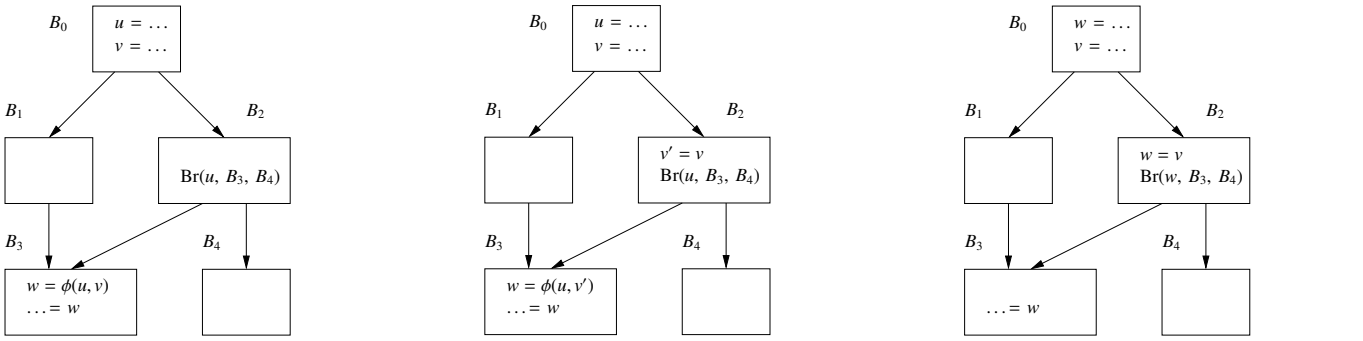
Although this naive translation seems, at first sight, correct, Briggs et al. [12] pointed subtle errors due to parallel copies and/or critical edges in the control flow graph. Two typical situations are identified, the "lost copy problem" and the "swap problem", some patches are proposed to handle them correctly, and a "more complicated algorithm that includes liveness analysis and a pre-order walk over the dominator tree" (Page 880) is quickly presented for the general cases, but with neither a discussion of complexity, nor a correctness proof. Nevertheless, according to the authors, this solution "cures the problems that (they) have seen in practice" (Page 879).

The first solution, both simple and correct, was proposed by Sreedhar et al. [3]. In addition to the copies at the end of each control flow predecessor, they insert another copy at the entry of the basic block for each ϕ-function. This simple mechanism, detailed hereafter, is sufficient to make the translation always correct, except for the special cases described later. Several strategies are then proposed to introduce as few copies as possible, including a special rule to eliminate more copies than with standard coalescing so that "copies that it places cannot be eliminated by the standard interference graph based coalescing algorithm" [3, Page 196]. This last (also unproved) claim turns out to be correct, but only for the very particular way copies are inserted, i.e., always after the previously-inserted copies in the same block. Also, the way coalescing is handled is again more a patch, driven by implementation considerations, than a conceptual choice. We will come back to this point later. Nevertheless, our technique is largely inspired by the various algorithms of Sreedhar et al.

In other words, these previous approaches face some conceptual subtleties that make them sometimes incorrect, incomplete, overly pessimistic, or too expensive. This is mostly due to the fact that a clean definition of interference for variables involved in a ϕ-function is missing, while it is needed both for correctness (for adding necessary copies) and for code quality (for coalescing useless copies). Our first contribution, beyond algorithmic improvements, is to address this key point. Thanks to this interference definition, we develop a clean out-of-SSA translation approach, in which correctness and optimization are not intermixed. The resulting implementation is much simpler, has no special cases, and we can even develop fast algorithms for each independent phase, without compromising the quality of results. Before detailing our contributions, we first explain the basics of out-of-SSA translation, with copy insertion and coalescing, and its intrinsic subtleties.

A. Correctness of ϕ-functions elimination with copy insertion

Consider a ϕ-function $a_0 = \phi(a_1, \ldots, a_n)$ placed at entry of a block B_0: a_0 takes the value of a_i if the control-flow comes from the i-th predecessor block of B_0. If $a_0, \ldots, a_n$ can be given the same name without changing the semantics of the program, the ϕ-function can be eliminated. When this property is true, the SSA form is said to be conventional



a) Initial SSA code b) Insufficient copy insertion c) Incorrect out-of-SSA translation

Fig. 1. Considering liveout sets may not be enough.

(CSSA) [3]. This is not always the case, in particular after copy propagation or code motion, as some of the a_i may "interfere". The technique of Sreedhar et al. [3] consists in three steps: a) translate SSA into CSSA, thanks to the introduction of copies; b) eliminate redundant copies; c) eliminate ϕ-functions and leave CSSA. In their Method I, the translation into CSSA is as follows. For each ϕ-function $a_0 = \phi(a_1, \ldots, a_n)$ at entry of block B_0, with predecessor blocks B_i, $1 \leq i \leq n$:

- $n + 1$ new variables $a'_0, \ldots, a'_n$ are introduced;
- a copy $a'_i = a_i$ is placed at the end of B_i;
- a copy $a_0 = a'_0$ is placed just after all ϕ-functions in B_0;
- the ϕ-function is replaced by $a'_0 = \phi(a'_1, \ldots, a'_n)$.

If, because of different ϕ-functions, several copies are introduced at the same place, they should be viewed as parallel copies. This is what we propose, as Leung and George do in [18]. However, as far as correctness is concerned, copies can be sequentialized in any order, as they concern different variables. This is what Sreedhar et al. do in all their methods.

Lemma 1: If copies are placed in the predecessor blocks *after* any definition in them, then the introduction of the new variables a'_i and the corresponding copies, for all ϕ-functions, transform the code in CSSA form. In other words, replacing all variables a'_i by a new unique variable for each ϕ-function and removing all ϕ-functions is a correct out-of-SSA translation.

Proof: After insertion of copies, the code semantics is preserved. The variable a_i is copied (after its definition) into a'_i, then fed into the new ϕ-function to create a'_0, which is finally copied into a_0. All names are different, thus do not create any definition conflict. To show that the code is in CSSA, note that the variables a'_i have very short live ranges. The variables a'_i, for $i > 0$, are defined at the very end of disjoint blocks B_i, thus none is live at the definition of another: they do not interfere. The same is true for a'_0 whose live range is located at the very beginning of B_0, even if B_0 may be equal to B_i for some i. The $n+1$ variables a'_i are never simultaneously live on a given execution path, so they can share the same variable name. ■

Lemma 1 explains why the proposal of Cytron et al. was wrong. 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.

So, the trick is to split the definition of the ϕ-function itself with one new variable at the block entry, in addition to copies traditionally inserted at the end of predecessor blocks. Then, in the methods of Sreedhar et al., the copy involving a'_i is considered useless depending on the intersection of its live range with the liveout set of the block B_i. However, there is a first subtlety. Depending on the branch instruction, the copies cannot always be inserted at the very end of the block, i.e., after all variables uses and definitions. For example, for a ϕ-function after a conditional branch that uses a variable u, the copies are inserted *before* the use of u. Thus, the intersection check must be done with u also, otherwise some incorrect code can be generated. Consider the SSA code in Figure 1(a), which is not CSSA. As u is not liveout of block B_2, the optimized algorithm (Method III) of Sreedhar et al. considers that it is sufficient to insert a copy v' of v at the end of B_2. But the copy has to be inserted before the branch, so before the use of u (Figure 1(b)) and the code is still not CSSA since u and v' interfere. Removing the ϕ-function, i.e., giving the same name to w, u, and v' leads to the incorrect code of Figure 1(c). This problem is never mentioned in the literature. Fortunately, it is easy to correct it by considering the intersection with the set of variables live *just after the point of copy insertion* (here the liveout set plus u) instead of just the liveout set of the block.

There is a more 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. This point has never been mentioned before: out-of-SSA translation by copy insertion alone is not always possible, depending on the branch instructions and the particular case of interferences. We give such an example in [19].

These different situations illustrate again why out-of-SSA translation must be analyzed with care, to address correctness even before thinking of code optimization. Aggressive SSA optimizations can indeed make out-of-SSA translation tricky.

B. Going out of CSSA: a coalescing problem

Once the copies are inserted as in Section II-A, the code is in CSSA, except for the special cases of branch with definition explained above. Then, going out of CSSA is straightforward: all variables involved in a ϕ-function can be given the same name and the ϕ-functions can be removed. This solves the correctness aspect. To improve the code however, it is important to remove as many copies as possible. This can be treated with classic coalescing as CSSA is equivalent to standard code: liveness and interferences can be defined as for regular code (with parallel copies). The difference is that the number of introduced copies and of new variables can be artificially large, which can be too costly especially if an interference graph is used. Sreedhar et al. proposed several improvements: introducing copies only when variables interfere and updating conservatively the interference graph (Method II), a more involved algorithm that uses and updates liveness information (Method III), a special SSA-based coalescing, useful to complement Method I and Method II but useless after Method III. All these techniques rely on the explicit representation of **congruence classes** that partition the program variables into sets of variables coalesced together.

But why relying on special coalescing rules depending on the method? Actually, once the code is in CSSA, the optimization problem is a standard aggressive coalescing problem (i.e., with no constraints on the number of target variables) and heuristics exist for this NP-complete problem [20], [21]. The fact that the code is in SSA does not make it simpler or special. Also, Method III, even though it was primarily designed for speed, turns out to give better results than Method I followed by coalescing. This is because Sreedhar et al. rely on a too conservative definition of interferences to decide if two variables can be coalesced. As Section III-A will show, it is better to exploit the SSA properties to identify when two variables have the same **value**: in this case, they do not interfere even if their live ranges intersect. Then, with this intrinsic definition of interferences, there is no point to compare, in terms of quality of results, a method that introduces all copies first as in Method I or on the fly as in Method III. They should be equivalent. Furthermore, this definition of interferences is more accurate, thus more copies can be removed.

Another weakness in Sreedhar et al. model is that copies are inserted in a particular sequential order at the end or entry of basic blocks. We prefer to stick to the SSA semantics, i.e., to use parallel copies (all uses are read before any write occurs). We then sequentialize these copies, once we know which remain. The interest is twofold. First, with sequential copies, some additional interferences between the corresponding variables appear, which hinders coalescing, especially in case of additional register constraints. Second, with parallel copies, we avoid a tricky update of liveness information: copies are handled in a uniform way. This is fundamental to reduce the engineering effort. In [19], we illustrate this coalescing mechanism with the classic swap and lost copy problems.

In conclusion, with a more accurate interference definition, the use of parallel copies, a standard coalescing algorithm to remove copies, we get what we need: a conceptually simple approach, provably correct, in which correctness and optimization are separated. This is of high importance for implementing SSA without bugs in an industrial compiler.

III. K - -SSA

We can now give an overview of the general process before detailing each individual step. Conceptually, our out-of-SSA translation process comprises four successive phases:

1) Insert parallel copies for all ϕ-functions as in Method I of Sreedhar et al. and coalesce all a'_i together.
2) Build the interference graph with an accurate definition of interference, using the "SSA value" of variables..
3) Coalesce aggressively, maybe with renaming constraints.
4) Sequentialize parallel copies, possibly with one more variable and some additional copies, in case of swaps.

Step 1 was presented in Section II-A. We now detail Steps 2, 3, and 4 in Sections III-A, III-B, and III-C respectively. Also, thanks to the independence between correctness (Step 1) and optimization (Step 3), we propose algorithms that make the whole process fast enough for just-in-time compilation. They are described in Section IV: fast live range intersection test (Section IV-A), fast interference test and node merging (Section IV-B), "virtualization" of initial copy insertion (Section IV-C), i.e., copy insertion on the fly as in Method III of Sreedhar et al. With these techniques, we can even avoid to build the liveness sets and the interference graph, for a gain in memory footprint too.

A. Live range intersection and equal values

It is common to find in the literature the following definition of interference "two variables interfere if their live ranges intersect" (e.g. in [22], [15], [23]) or its refinement "two variables interfere if one is live at a definition point of the other" (e.g. in [24]). 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" [17]: 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 [2]. In SSA with the dominance property – in which each use is

dominated by its unique definition, so it is defined – upward exposed use analysis is sufficient. The notion of value is even harder, but may be approximated using data-flow analysis on specific lattices [25], [26]. 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 [2].

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. [17] proposed a counting mechanism, rediscovered in [27], to update the interference graph, but it was considered to be too space consuming. Recomputing it from time to time was preferred [17], [24]. Since then, most coalescing techniques based on graph coloring use either live range intersection graph [3], [15] or Chaitin's interference graph with reconstructions [22], [28].

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 an equivalence relation. This property is used in SSA dominance-based copy folding and global value numbering [29]. The **value** of an equivalence class 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 **interfere with** b **if live**(a) **intersects live**(b) **and** $V(a) \neq V(b)$. (The first part reduces to def(a) $\in$ live(b) or def(b) $\in$ live(a) thanks to the dominance property [15].) In the previous example, a, b, and c have the same value $V(c) = V(b) = V(a) = a$, thus they do not interfere.

It should be clear now why we advocate the out-of-SSA translation previously introduced: introduce copies to ensure the correctness, exploit the SSA properties to identify variables that have the same value, and coalesce variables that do not interfere. Because of our more accurate notion of interference, there is no need to rebuild or update the interference graph, no need to develop a special SSA-based coalescing algorithm as in [3], no need to make a distinction between variables that can be coalesced with Chaitin's approach or not. What is important is just to know if they interfere or not.

To make the interference definition complete, it remains to define precisely when variables are live. In SSA, the status of the ϕ-function and its arguments is unclear because they live beyond the dominance tree. However, after Step 1 (the introduction of the variables a'_i) and their coalescing into one unique node, the code is in CSSA and could be translated directly into standard code. The liveness of this unique node is thus precisely defined: its live range is the union of the live range of its constituting elements, using traditional liveness definition for standard code. In other words, it lives from the output of each parallel copy in the predecessor block to the input of the parallel copy where the ϕ-function exists. Also, to check the intersection with other variables, it is sufficient to check the intersection at the parallel copies locations.

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.

B. Coalescing

As discussed earlier, the out-of-SSA translation is nothing but a traditional aggressive coalescing problem, i.e., with no constraints on the number of colors. If all copies are initially inserted, as in Method I of Sreedhar et al., any sophisticated technique can be used. In particular, it is possible to use weights to treat in priority the copies placed in inner loops: this reduces the number of static and of dynamically-executed copies. Sreedhar et al. do not use weights. We use classic profile information to get basic block frequencies. Note however that this weight may be slightly under-estimated: in some cases, there may be an additional copy if a swap is needed when sequentializing a parallel copy (see Section III-C).

If copies are inserted on the fly, as in Method III of Sreedhar et al., the copy variables a'_i are created only when needed, but reasoning as if they were available. We call this process *virtualization* (see Section IV-C). To make this possible, ϕ-functions are considered one after the other, thus copies are somehow coalesced in this particular ϕ-function by ϕ-function order. Also, Sreedhar et al. use a deferred copy insertion mechanism that, even if not expressed in these terms, amounts to build some maximal independent set of variables (i.e., that do not interfere), which are then coalesced. This gives indeed slightly better results than reasoning one copy at a time. In our virtualized version, we also process one ϕ-function at at time, but simply consider its related copies by decreasing weight.

We also address the problem of **copy sharing**. Consider again the example of two successive copies $b = a$ and $c = a$. 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. Similar practical situations, due to calling convention constraints, are given in [18]. 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 (note: 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). Section III-E measures the effects of this important post-optimization, which is a direct by-product of our value-based interference definition.

C. 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 [30]. For completeness, we give in our companion technical report [19] a detailed description of this algorithm with its complete pseudo-code.

D. Handling of register renaming constraints

Register renaming constraints, such as calling conventions or dedicated registers, are treated with **pinned variables** [18]. A pinned variable is a SSA variable pre-coalesced to another variable or pre-allocated to an architectural register [20]. To avoid interferences, we first ensure that a pinned variable has a short live range spanning no more than the constraining instruction. This is achieved by splitting the live ranges of pinned variables with parallel copies inserted just before and after the constraining instructions. These parallel copies are then coalesced just like the copies implied by the ϕ-functions.

Pre-allocated variables require a special treatment, as two variables pre-allocated to different architectural registers must not be coalesced. So all variables pre-allocated to a given register are first pre-coalesced together, and the corresponding congruence class is labeled by this register. Next, when checking the interference between two congruence classes, we first check if they are labeled with two different registers. If yes, they are considered to be interfering.

E. Qualitative experiments

The experiments were done on the SPEC CINT2000 benchmarks (with the exception of the C++ benchmark eon) compiled at aggressive optimization level, using the Open64-based production compiler for the STMicroelectronics ST200 VLIW family. This compiler was directly connected to the STMicroelectronics JIT compiler for CLI [9], which implements the out-of-SSA techniques proposed in this paper, the techniques of Sreedhar et al. [3], and also the fast liveness checking for SSA [16]. This experimental setup ensures that algorithms are implemented in the context of a real JIT compiler, yet the code they process is highly optimized C code.

First, we evaluated how the accuracy of interference impacts the quality of coalescing by implementing seven variants of coalescing. Below $a \mapsto b$ is a copy to be removed. X and Y are the congruence classes of a and b, i.e., the set of coalesced variables that contain a and b (see Section II-B). In case of coalescing, X and Y will be merged into a larger class.

Intersect X and Y can be coalesced if no variables $x \in X$ and $y \in Y$ have intersecting live ranges.

Sreedhar I This is Sreedhar et al. SSA-based coalescing: X and Y can be coalesced if there is no pair of variables $(x, y) \in (X \times Y) \setminus \{(a, b)\}$ whose live ranges intersect: (a, b) is not checked as a and b have the same value.

Chaitin X and Y can be coalesced if no variables $x \in X$ and $y \in Y$ interfere following Chaitin's definition, i.e., x is live at the definition of y and this definition is not a copy $x \mapsto y$ (or the converse).

Value X and Y can be coalesced if no variables $x \in X$ and $y \in Y$ interfere following our value-based interference definition, i.e., their live ranges intersect and have a different value, as explained in Section III-A.

Sreedhar III This is the virtualization mechanism used in Method III of Sreedhar et al. Copies are inserted, considering one ϕ-function at a time, as explained in Section III-B. We added the SSA-based coalescing of Method I, which is useless for ϕ-related copies, but not for copies due to register renaming constraints.

Value + IS This is Value, extended with a quick search for an independent set of variables, for each ϕ-function, as in Sreedhar III.

Sharing This is Value + IS, followed by our copy sharing mechanism, see Section III-B. If c is live just after the copy $a \mapsto b$ and $V(c) = V(a)$, i.e., a and c have the same value, then, denoting Z the congruence class of c, 1) if $Y = Z$ and $Y \neq X$, the copy $a \mapsto b$ is redundant and can be removed; 2) if X, Y, and Z are all different, and if Y and Z can be coalesced (following the Value rule), the copy $a \mapsto b$ can be removed after coalescing Y and Z because c has already the right value.

Figure 2 gives, for each variant, the ratio of number of remaining static copies compared to the less accurate technique (Intersect). Comparing the cost of remaining "dynamic" copies, computed with a static estimate of the basic block frequencies, gives similar results. The first four variants show what is gained when using a more and more accurate definition of interferences (from Intersect to Value). It is interesting to note, again, that Sreedhar I is quite inefficient as, for

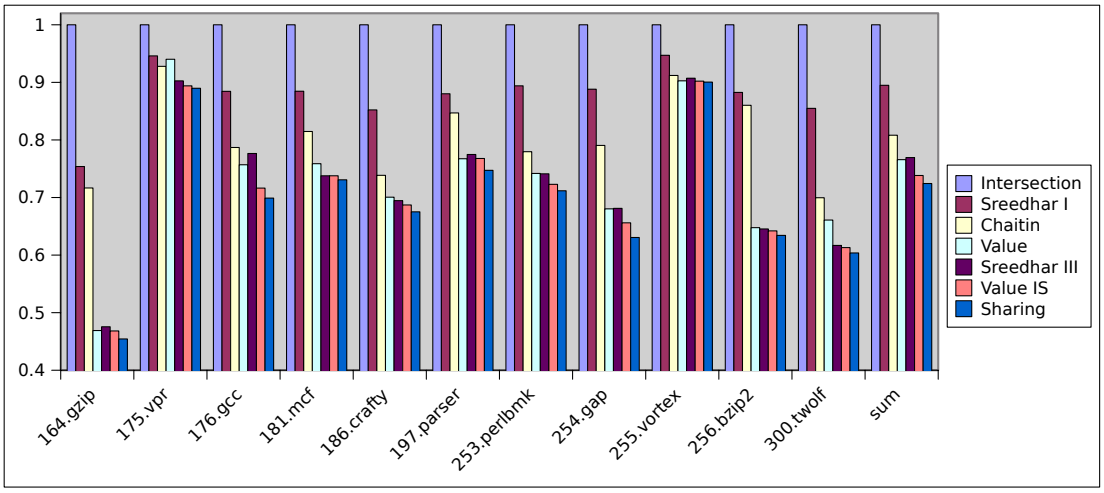


Fig. 2. Impact of interference accuracy and coalescing strategies on remaining number of moves.

example, it cannot coalesce two congruence classes X and Y if $X \times Y$ contains two pairs of intersecting copy-related variables. Introducing variables on the fly as in Method III avoids this problem as only copies that cannot be removed by the SSA-based coalescing are introduced (but, in [3], it is not tuned to optimize weighted moves). Also, the independent set search integrated in this method improves the results compared to Value, which is the basic version with our value-based interference. If this independent set trick is also added to Value, our technique outperforms Sreedhar III (version Value + IS). The last variant, Sharing, shows that we can go even further with our additional sharing mechanism.

These experiments confirm that the decomposition of the problem into the insertion of parallel copies followed by coalescing with an accurate identification of values is sufficient to obtain the best code quality so far. In addition, it is also a clean and flexible solution because, with our intrinsic value-based interference definition, the fact that two variables can be coalesced does not depend on the way we introduce copies, wether before coalescing as in Method I or on the fly as in Method III. Also, with less programming effort, we can do slightly better than Sreedhar III in terms of quality of results. More importantly, since our approach separates the correctness and the quality of results from how coalescing is implemented, we can focus on algorithm speed and memory footprint. These two points are addressed in the next section.

IV. M -

Implementing the technique of Section III may be too costly. First, it inserts many instructions before realizing most are useless, and copy insertion is 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 a working graph and of an interference graph, relying nevertheless on classic liveness sets; b) replacing the quadratic time complexity interference check between congruence classes by a linear complexity algorithm; c) replacing classic liveness set computation by fast liveness checking for SSA; d) emulating ("virtualizing") the introduction of all initial copies, as in Method III of Sreedhar et al.

If an interference graph is available, it is not clear whether using an additional working graph is much more expensive or not, but, in the context of aggressive coalescing, both Chaitin [24] and Sreedhar et al. [3] preferred not to use one. To get rid of it, Sreedhar et al. manipulate congruence classes, that is, sets of variables that are already coalesced together. Then, two variables can be coalesced if their corresponding congruence classes do not contain two interfering variables, one in each congruence class. This quadratic number of variable-to-variable interference tests might be expensive. In Section IV-B, we propose a linear time complexity algorithm for interference detection between two congruence classes.

When an interference graph is not available, a variable cannot directly access the list of variables that interfere with it, so queries are typically restricted to interference *checking*, i.e., existence of an interference. The classic approach consists in computing the interference relation, which is stored as a bit matrix. Building such interference graph implies a costly traversal of the program and requires the liveness sets. Then, interference queries are $O(1)$. A second approach is to perform value tests, dominance checks, and liveness checks, without relying on any pre-computation of liveness sets. Queries are more time-consuming, but this avoids the need for an interference graph. Section IV-A details such methods.

Finally, Section IV-C explains how to adapt the virtualization mechanism used in Method III of Sreedhar et al., which inserts copies only when needed, to avoid the introduction of many new variables and of useless copies that will be removed.

A. Live-range intersection tests

Remember Section III-A. What we need is a double test, test of live range intersection and test of equality of values. The next section explains how to check, with this interference notion, whether two congruence classes (sets of variables already coalesced) **interfere** or not. Before, we need an algorithm to decide if two live ranges **intersect**. Several methods exist for testing live range intersection, as we briefly recall here.

The classic method builds live-in and live-out sets for each basic block, with data-flow analysis. A refinement is to build the sets only for the global variables live along some control-flow edge. The live range intersection graph can then be built, either lazily or completely, by traversing backward each basic block. This has the disadvantage that the computation is fairly expensive and its results are easily invalidated by program transformations. For example, in Method III of Sreedhar et al., some effort must be done to update live-in and live-out sets. The fact that sequential copies are used instead of parallel copies makes the update even more complicated.

Budimlić et al. [15] proposed an intersection test that avoids the use of the interference graph. It uses SSA properties and liveness information at basic block boundaries. The important property is that two SSA variables intersect if and only if the variable whose definition dominates the definition of the other is live at this second definition point. Thus, either it is live-in for this block or defined earlier in the same block. If it is live-out, the two live ranges intersect, otherwise a backwards traversal of the block is needed to decide. Thus, this test avoids the use of an interference graph but requires the storage of the liveness sets and, in some cases, performs block traversals. These traversals can be avoided if def-use chains are available.

Recently, a SSA liveness check has been proposed by Boissinot et al. [16], which can answer whether a given variable is live at a given program location. The technique does not require any pre-computed liveness sets but relies on a pre-computation that only depends on the structure of the control flow graph. In other words, it is still valid even if some instructions are moved, introduced, or removed. For checking if two SSA variables interfere, we can use this test by checking if one is live at the definition of the other. According to [16], this technique is a **fast** liveness check for SSA programs.

We will not detail these different intersection tests any further. In the next section, they are used as a black box for developing a linear time algorithm that checks interference between two congruence classes.

B. Linear interference test between two congruence classes (with extension to value-based interferences)

In Sections III-A and III-C, we presented our two first main contributions, the notion of value-based interference (taking into account equality of values) and the method to sequentialize parallel copies. We now develop our third main contribution: how to efficiently perform an interference test between two sets of already-coalesced variables (congruence classes in Sreedhar et al. terminology). Suppose that the two tests needed to decide if two SSA variables interfere – the live range intersection test (Section IV-A) and the "has-the-same-value" test (Section III-A) – are available as black boxes. To replace the quadratic number of tests by a linear number of tests, we simplify and generalize the dominance-forest technique proposed by Budimlić et al. [15]. Our contributions are: a) we avoid constructing explicitly the dominance forest; b) we are also able to check for interference between *two* sets; c) we extend it to support the notion of equality of value.

Given a set of variables, Budimlić et al. define its *dominance forest* as a graph forest where ancestors of a variable are exactly the variables of the set that dominate it (i.e., whose definition point dominates the definition point of the other). The key idea of their algorithm is that the set contains two intersecting variables if and only if it contains a variable that intersects with its parent in the dominance forest. The trick is then to traverse the dominance forest and to check the live range intersection for each of its edges. Instead of constructing explicitly the dominance forest, we propose to represent each congruence class as a list of variables ordered according to a pre-DFS order $\prec$ of the dominance tree (i.e., a depth-first search where each node is ordered before its successors). Then, because querying if a variable is an ancestor of another one can be achieved in $O(1)$ (simple dominance test), simulating the stack of a recursive traversal of the dominance forest is straightforward. Thus, as in [15], we can derive a linear-time intersection test for a set of variables (**Algorithm 1**).

Algorithm 1: Check intersection in a set of variables

```
   Data: List of variables list sorted according to a
         pre-DFS order of the dominance tree
   Output: Returns        if the list contains an interference
1  dom ← empty_stack ;            /* stack of the traversal */
2  i ← 0 ;
3  while i < list.size() do
4      current ← list(i++) ;
5      other ← dom.top() ;        /*       if dom is empty */
6      while (other ≠       ) and dominate(other, current) =
           do
7          dom.pop() ;  /* not the desired parent, remove */
8          other ← dom.top() ;    /* consider next one */
9      parent ← other ;
10     if (parent ≠       ) and (intersect(current, parent) =
           ) then return      ;   /* intersection detected */
11     dom.push(current) ;    /* otherwise, keep checking */
12 return      ;
```

Suppose now that we have two intersection-free sets (two congruence classes of non-intersecting variables) `blue` and `red`. To coalesce them, we need to check if there is an intersection between both. We do as if the two sets were merged and we apply the previous technique. The only difference is that we can save intersection tests when we compare two variables in the same set: in Line 10 of Algorithm 1, the `intersect` test should check if parent and current belong to a different list before running an expensive intersection test. Also, because each set is represented as an ordered list, traversing two lists

in order is straightforward. We just have to progress in the right list, according to the pre-DFS order $\prec$ of the dominance tree. We now use two indices i_r and i_b and we replace Lines 2-4 of the previous algorithm by the following lines:

```
i_r ← 0 ; i_b ← 0 ;
while i_r < red.size() or i_b < blue.size() do
    if i_r = red.size() or (i_r < red.size() and i_b < blue.size()
    and blue(i_b) ≺ red(i_r)) then current ← blue(i_b++) ;
    else current ← red(i_r++) ;
```

The last step is to extend our intersection technique to an interference test that takes equalities into account. Suppose that b is the parent of a in the dominance forest. In the previous algorithm, the induction hypothesis is that the subset of already-visited variables does not contain any intersection. Then, if c is an already-visited variable, the fact that b and a do not intersect guarantees that c and a do not intersect, otherwise the intersection of b and c would have already been detected. However, for interferences with equalities, this is no longer true. The variable c may intersect b but if they have the same value, they do not interfere. The consequence is that, now, a and c may intersect even if a and b do not intersect. However, if a does not intersect b and any of the variables it intersects, then a does not intersect any of the already-visited variables. To speed up such a test and to avoid checking intersection between variables in the same set, we keep track of one additional information: for each variable a, we store the nearest ancestor of a that has the same value and that intersects it. We call it the "equal intersecting ancestor" of a. We assume that the equal intersecting ancestor is pre-computed within each set, denoted by `equal_anc_in`(a), and we will compute the equal intersecting ancestor in the opposite set, denoted by `equal_anc_out`(a). The skeleton of the complete algorithm for interference test with equality is the same as Algorithm 1, with the patch to progress along the lists `Red` and `Blue`, and where the call Line 10 is now an **interference** test (Function `interference`). The principles of the algorithm are given in the codes themselves. Two equal intersecting ancestors, `in` and `out`, are used to make sure that the test `intersect`(a, b), which runs a possibly expensive intersection test, is performed only if a and b belong to different sets.

Note that once a list is empty and the stack does not contain any element of this list, there is no more intersection or updates to make. Thus, the algorithm can be stopped, i.e., the while loop condition in Algorithm 1 can be replaced by:

```
while (i_r < red.size() and n_b > 0) or (i_b < blue.size() and
n_r > 0) or (i_r < red.size() and i_b < blue.size()) do
```

where n_r (resp. n_b) are variables that must be implemented to count the number of elements of the stack that come from the list `Red` (resp. `Blue`). Finally, if a coalescing of the two sets occurs, it remains to store the two lists as a unique ordered list (in linear time, in a similar joint traversal) and to update the equal intersecting ancestor `equal_anc_in`(a) for the combined set as the maximum (following the pre-DFS order $\prec$) of `equal_anc_in`(a) and `equal_anc_out`(a).

Function `update_equal_anc_out`(a, b)

Data: Variables a and b, same value, but in different sets
Output: Set nearest intersecting ancestor of a, in other set, with same value (if does not exist)

```
1 tmp ← b ;
2 while (tmp ≠ ) and (intersect(a, tmp) = ) do
3     tmp ← equal_anc_in(tmp) ;  /* follow the chain of
      equal intersecting ancestors in the other set */
4 equal_anc_out(a) ← tmp ;  /* tmp intersects a or  */
```

Function `chain_intersect`(a, b)

Data: Variables a and b, different value, in different sets
Output: Returns if a intersects b or one of its equal intersecting ancestors in the same set

```
1 tmp ← b ;
2 while (tmp ≠ ) and (intersect(a, tmp) = ) do
3     tmp ← equal_anc_in(tmp) ;  /* follow the chain of
      equal intersecting ancestors */
4 if tmp = then return else return ;
```

Function `interference`(a, b)

Data: A variable a and its parent b in the dominance tree
Output: Returns if a interferes (i.e., intersects and has a different value) with an already-visited variable. Also, update `equal_anc` information

```
/* a and b are assumed to not be equal to  */
1 equal_anc_out(a) ←  ;  /* initialization */
2 if a and b are in the same set then
3     b ← equal_anc_out(b) ; /* check/update in other set */
4 if value(a) ≠ value(b) then
5     return chain_intersect(a, b) ;  /* check with b and its
      equal intersecting ancestors in the other set */
6 else
7     update_equal_anc_out(a, b) ;  /* update equal
      intersecting ancestor going up in the other set */
8     return ;  /* no interference */
```

C. Virtualization of ϕ-nodes

Implementation of the whole procedure, as described in Section III, starts by introducing many new variables a'_i (one for each argument of a ϕ-function, plus its result) and many copies in the block where the ϕ occurs and in its predecessors. These variables are immediately coalesced together, in what we call a ϕ-node, and stored in a congruence class. Nevertheless, in the data structures used (interference graph, liveness sets, variable name universe, parallel copy instructions, congruence classes), these variables exist as data items and consume memory and time, even if at the end, after coalescing, they may disappear.

To avoid the introduction of these initial variables and copies, the technique is to emulate the whole process, as does Method III of Sreedhar et al., which introduces necessary copies on the fly, when they appear to be needed. We want our implementation to be clean and able to handle all special cases without tricks. For that purpose, we use exactly the same

algorithms than for our solution without virtualization. We use a special location in the code, identified as a "virtual" parallel copy, where the real copies, if any, will be placed. The original arguments (resp. definition) of a ϕ-function are then assumed, initially, to have a "use" (resp. "def") in the parallel copy but are not considered as live-out (resp. live-in) along the corresponding control flow edge. Then, the algorithm selects copies to coalesce, following some order, either a real copy or a virtual copy. If it turns out that a virtual copy $a_i \mapsto a'_i$ (resp. $a'_0 \mapsto a_0$) cannot be coalesced, the copy is materialized in the parallel copy and a'_i (resp. a'_0) becomes explicit in its congruence class. The corresponding ϕ-operand is replaced and the use of a'_i (resp. def of a'_0) is now assumed to be on the corresponding control flow edge. This way, only copies that the first approach would finally leave uncoalesced are introduced.

The only key point to make the emulation of copy insertion possible is that one should never have to test an interference with a variable that is not yet materialized or coalesced. For that, ϕ-nodes are treated one by one, and all virtual copies that imply a variable of the ϕ-node are considered (either coalesced or materialized) before examining any other copies. The weakness of this approach is that a global coalescing algorithm cannot be used because only a partial view of the interference structure is available during the algorithm. However, the algorithm can still be guided by the weight of copies, i.e., the dynamic count associated to the block where it would be placed if not coalesced. The rest is only a matter of accurate implementation, but once again intrinsically this is nothing else than emulating these copies and variables.

D. Results in terms of speed and memory footprint

To measure the potential of our different contributions, in terms of speed-up and memory footprint reduction, we implemented a generic out-of-SSA translation that enables us to evaluate different combinations. We selected the following:

`Us I` Simple coalescing with no virtualization but different techniques for checking interferences and liveness.

`Sreedhar III` Method III of Sreedhar et al. (thus with virtualization) complemented by their SSA-based coalescing for non ϕ-related copies. Both use an interference graph stored as a bit-matrix and liveness sets as ordered sets.

`Us III` Our implementation of ϕ-nodes coalescing with virtualization followed by coalescing of other copies. This implementation is generic enough to support various options: with parallel or sequential copies, with/without interference graph, with/without liveness sets. Hence, its implementation is less tuned than `Sreedhar III`.

By default, `Us III` and `Us I` use an interference graph and classic liveness sets. The options are:

`InterCheck` No interference graph: intersections are checked using a dominance test and liveness sets as in [15].

`InterCheck+LiveCheck` No interference graph and no liveness sets: intersections are checked with the fast liveness checking algorithm of [16], see Section IV-A.

`Linear+InterCheck+LiveCheck` In addition, our linear intersection check is used instead of the quadratic one.

When an interference graph, liveness sets, or liveness checking are used, timings include their construction. Figure 3 shows the timings for these different variants versus `Sreedhar III` as a baseline. `InterCheck` always slows down the execution, while `LiveCheck` and `Linear` always fasten the execution with a significant ratio. A very interesting result is that the simple SSA-based coalescing algorithm without any virtualization is as fast as the complex algorithm with virtualization. Indeed, when using `Linear+InterCheck+LiveCheck`, adding first all copies and corresponding variables before coalescing them, does *not* have the negative impact measured by Sreedhar et al. any longer. Hence `Us I+Linear+InterCheck+LiveCheck` provides a quite attractive solution, which is at least twice faster than `Sreedhar III`. Also, as mentioned before, the quality (in terms of moves) of the generated code does not depend on the virtualization, unlike in Sreedhar et al. methods, thanks to our interference definition with equality of values.

Figure 4 shows the memory footprint used for the interference graph and the liveness sets. The variable universe used for liveness and interference information is restricted to ϕ-related and copy-related variables.

Interference graph is stored using a half-size bit-matrix. `Measured` provides the measured footprint from the statistics provided by our memory allocator. In `Sreedhar III` or `Us III`, variables are added incrementally so the bit-matrix grows dynamically. This leads to a memory footprint slightly higher than for a perfect memory. The behavior of such a perfect memory is evaluated in `Evaluated` using the formula $\lceil \frac{\#\text{variables}}{8} \rceil \times \#\text{variables}/2$.

Liveness sets are stored as ordered sets. `Measured` provides the measured footprint of the livesets, without counting those used in liveness construction. As for the interference graph, livesets are modified by `Sreedhar III` or `Us III`. Since the number of simultaneous live variables does not change, their sizes remain roughly the same. Because the use of ordered sets instead of bit-sets is arguable, we evaluated the corresponding footprint of liveness sets, for a perfect memory, by counting the size of each liveset. For bit-sets, we evaluated the footprint using the formula $\lceil \frac{\#\text{variables}}{8} \rceil \times \#\text{basicblocks} \times 2$.

Liveness checking uses 2 bit-sets per basic block, plus a few other sets during construction. These sets are measured in the memory footprint. A perfect memory is evaluated using the formula $\lceil \frac{\#\text{basicblock}}{8} \rceil \times \#\text{basicblock} \times 2$.

The results show that the main gain comes from the removal of the interference graph. We would like to point out that in practice the memory usage for liveness sets construction is difficult to optimize and might lead to a very large memory footprint in practice. On the other hand, the liveness checking objects depend only on the control flow graph. Our statistics excessively favors the classic liveness sets: the memory usage for their construction has been removed from the statistics, while the memory usage for the liveness checking has been kept. As an illustration, in our compiler, the memory footprint

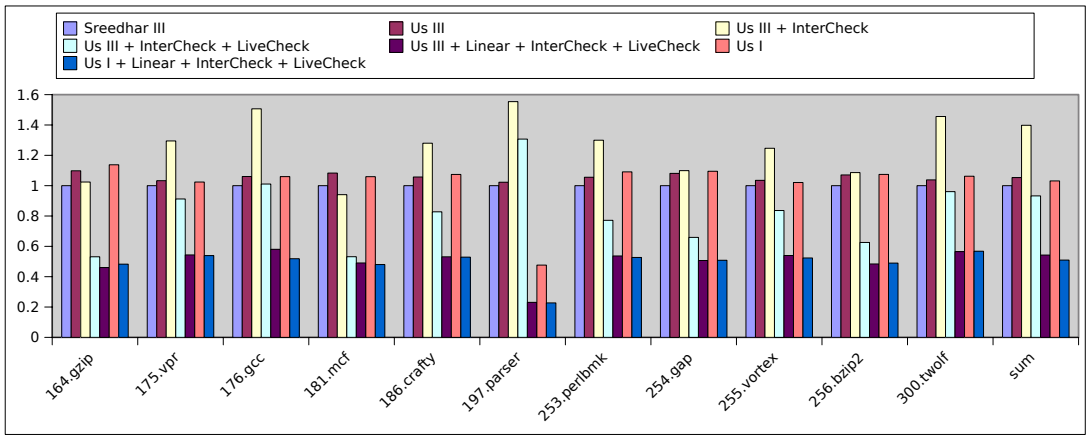


Fig. 3. Performance results in terms of speed (time to go out of SSA).

for liveness sets construction has the same order of magnitude than the memory footprint for the interference graph.

In conclusion, `Us I+Linear+InterCheck+LiveCheck` is a simple and clean solution (it avoids the complexity of the implementation of virtualization), yet it leads to a memory footprint at least 10 times smaller than `Sreedhar III`.

V. C

We revisited the out-of-SSA translation techniques for the purposes of ensuring correctness, quality of generated code, and efficiency (speed and memory footprint) of algorithms. This work is motivated by the use of the SSA form in JIT compilers for embedded processors. The techniques proposed by Sreedhar et al. [3] fix the correctness issues of previous algorithms, are insensitive to the program flow graph, and produce code of good quality. However, their optimized version (Method III) is hard to implement correctly when dealing with branch instructions that use or define variables. The technique proposed by Budimlić et al. [15] is geared towards speed and introduces fast intersection of SSA live ranges and dominance forests for finding intersections in a set of SSA variables in linear time. This technique does not allow critical edges and is difficult to implement correctly. Still, the idea to optimistically coalesce variables with a rough but cheap filtering, then decoalesce interfering variables within the obtained congruence classes, is interesting. This coalescing scheme is orthogonal to and compatible with our approach.

We significantly advanced the understanding of out-of-SSA translation by reformulating it as an aggressive coalescing problem over the transformed SSA program resulting from Method I of Sreedhar et al. Our key insight is that interferences must be considered as intersection refined with value equivalence for any out-of-SSA translation to be effective, and this is supported by our experiments. Thanks to the SSA structural properties, computing the value equivalence comes at no cost. This leads to a provably-correct solution, generic, easy to implement, that can benefit from register allocation techniques. In particular, we can handle register renaming constraints (dedicated registers, calling conventions, etc.).

Then, we generalized the idea of dominance forests of Budimlić et al., first to enable interference checking between two congruence classes, then to integrate the check for equal values. In addition, our implementation is much simpler as we do not explicitly build the dominance forest. The reduced number of SSA variable intersection tests that results from this technique enables more costly intersection checks that do not rely on liveness sets and explicit interference graph.

Last, we proposed an algorithm for the virtualization of copies that preserves the correctness and quality of our approach while providing a very efficient method for out-of-SSA translation regarding both speed and memory consumption.

Our experiments performed on the SPEC CINT2000 benchmarks show that a simple SSA-based coalescing algorithm that uses our previously described techniques but without any virtualization is as fast as the complex algorithm with virtualization. It yields an out-of-SSA translation algorithm that outperforms the speed of Method III of Sreedhar et al. by a factor of 2, that reduces the memory footprint by a factor of 10, while keeping, at least better, its coalescing abilities.

R

[1] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451 – 490, 1991.

[2] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java*, 2nd ed. Cambridge University Press, 2002.

[3] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam, "Translating out of static single assignment form," in *Static Analysis Symposium (SAS'99)*, Italy, 1999, pp. 194 – 204.

[4] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth, "Fast, effective code generation in a just-in-time java compiler," in *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'98)*. Montreal, Quebec, Canada: ACM Press, 1998, pp. 280–290.

[5] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 5, pp. 895–913, 1999.

[6] O. Traub, G. Holloway, and M. D. Smith, "Quality and speed in linear-scan register allocation," in *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'98)*. Montreal, Quebec, Canada: ACM Press, 1998, pp. 142–151.

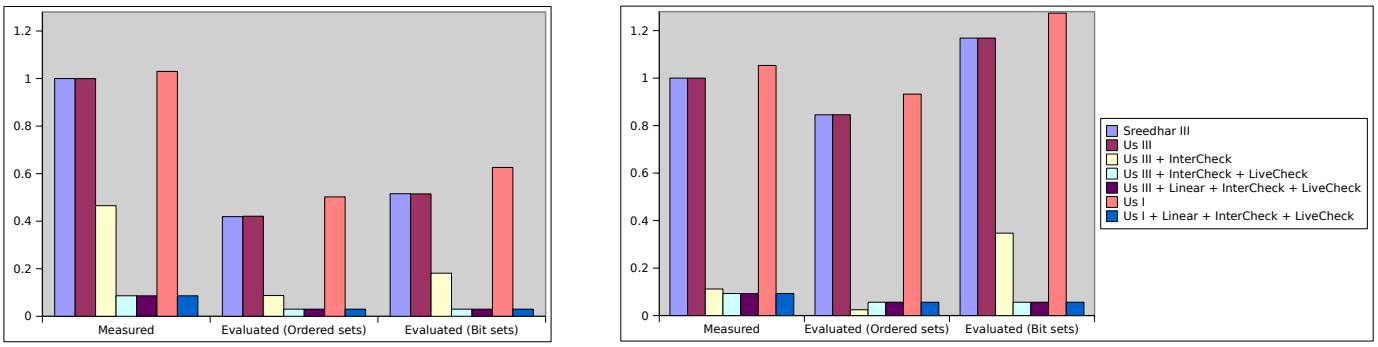


Fig. 4. Performance results in terms of memory footprint (maximum and total).

[7] C. Wimmer and H. Mössenböck, "Optimized interval splitting in a linear scan register allocator," in *ACM/USENIX International Conference on Virtual Execution Environments (VEE'05)*. Chicago, IL, USA: ACM, 2005, pp. 132–141.

[8] V. Sarkar and R. Barik, "Extended linear scan: An alternate foundation for global register allocation," in *International Conference on Compiler Construction (CC'07)*, ser. Lecture Notes in Computer Science, vol. 4420. Braga, Portugal: Springer Verlag, Mar. 2007, pp. 141–155.

[9] B. Dupont de Dinechin, "Inter-block scoreboard scheduling in a JIT compiler for VLIW processors," in *Euro-Par 2008 - Parallel Processing, 14th International Euro-Par Conference*, ser. Lecture Notes in Computer Science, vol. 5168. Las Palmas de Gran Canaria, Spain: Springer, Aug. 2008, pp. 370–381.

[10] J. Cavazos and J. E. B. Moss, "Inducing heuristics to decide whether to schedule," in *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'04)*. Washington, DC, USA: ACM Press, 2004, pp. 183–194.

[11] V. Tang, J. Siu, A. Vasilevskiy, and M. Mitran, "A framework for reducing instruction scheduling overhead in dynamic compilers," in *Conference of the Center for Advanced Studies on Collaborative Research (CASCON'06)*. Toronto, Ontario, Canada: ACM, 2006, p. 5.

[12] 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*, vol. 28, no. 8, pp. 859–881, Jul. 1998.

[13] V. C. Sreedhar and G. R. Gao, "A linear time algorithm for placing ϕ-nodes," in *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*. ACM, 1995, pp. 62–73.

[14] A. Gal, C. W. Probst, and M. Franz, "Structural encoding of static single assignment form," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 2, pp. 85–102, dec 2005. [Online]. Available: http://www2.imm.dtu.dk/pubdb/p.php?4150

[15] Z. Budimlić, K. D. Cooper, T. J. Harvey, K. Kennedy, T. S. Oberg, and S. W. Reeves, "Fast copy coalescing and live-range identification," in *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'02)*. ACM Press, June 2002, pp. 25–32.

[16] B. Boissinot, S. Hack, D. Grund, B. D. de Dinechin, and F. Rastello, "Fast liveness checking for SSA-form programs," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO'08)*. ACM, 2008, pp. 35–44.

[17] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, pp. 47–57, Jan. 1981.

[18] A. Leung and L. George, "Static single assignment form for machine code," in *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'99)*. ACM Press, 1999, pp. 204–214. [Online]. Available: citeseer.ist.psu.edu/leung99static.html

[19] B. Boissinot, A. Darte, B. Dupont de Dinechin, C. Guillon, and F. Rastello, "Revisiting out-of-SSA translation for correctness, code quality, and efficiency," LIP, ENS-Lyon, France, Tech. Rep. RR2008-40, Dec. 2008.

[20] F. Rastello, F. de Ferrière, and C. Guillon, "Optimizing translation out of SSA using renaming constraints," in *International Symposium on Code Generation and Optimization (CGO'04)*. IEEE Computer Society Press, 2004, pp. 265–278.

[21] F. Bouchez, A. Darte, and F. Rastello, "On the complexity of register coalescing," in *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE Computer Society Press, Mar. 2007, pp. 102–114.

[22] L. George and A. W. Appel, "Iterated register coalescing," *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 3, May 1996.

[23] M. D. Smith, N. Ramsey, and G. Holloway, "A generalized algorithm for graph-coloring register allocation," in *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'04)*. ACM, 2004, pp. 277–288.

[24] G. J. Chaitin, "Register allocation & spilling via graph coloring," in *SIGPLAN Symposium on Compiler Construction (CC'82)*, 1982, pp. 98–101.

[25] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)*. ACM, 1988, pp. 1–11.

[26] F. Bouchez, A. Darte, C. Guillon, and F. Rastello, "Register allocation and spill complexity under SSA," LIP, ENS-Lyon, France, Tech. Rep. RR2005-33, Aug. 2005.

[27] B. Dupont de Dinechin, c. d. Fran C. Guillon, and A. Stoutchinin, "Code generator optimizations for the ST120 DSP-MCU core," in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'00)*, 2000, pp. 93 – 103.

[28] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 428–455, 1994.

[29] P. Briggs, K. D. Cooper, and L. T. Simpson, "Value numbering," *Software – Practice and Experience*, vol. 27, no. 6, pp. 701–724, 1997.

[30] C. May, "The parallel assignment problem redefined," *IEEE Transactions on Software Engineering*, vol. 15, no. 6, pp. 821–824, Jun. 1989.