Using the SSA-Form in a Code Generator

Benoît Dupont de Dinechin

Kalray SA

Abstract. In high-end compilers such as Open64, GCC or LLVM, the Static Single Assignment (SSA) form is a structural part of the target-independent program representation that supports most of the code optimizations. However, aggressive compilation also requires that optimizations that are more effective with the SSA form be applied to the target-specific program representations operated by the code generator, that is, the set of compiler phases after and including instruction selection.

While using the SSA form in the code generator has definite advantages, the SSA form does not apply to all the code generator program representations, and is not suited for all optimizations. We discuss some of the issues of inserting the SSA form in a code generator, specifically: what are the challenges of maintaining the SSA form on a program representation based on machine instructions; how the SSA form may be used in the if-conversion optimizations; why the SSA form does not seem to benefit instruction scheduling; and what is the state-of-the-art in SSA form destruction on machine code.

Keywords: SSA Form, Code Generation, If-Conversion, Instruction Scheduling.

1 Introduction

In a compiler for imperative languages such as C, C++, or FORTRAN, the code generator covers the set of code transformations and optimizations that operate on a program representation close to the target machine ISA, and produce an assembly source or relocatable file with debugging information as result.

The main duties of code generation are: lowering the program intermediate representation to the target machine instructions and calling conventions; laying out data objects in sections and composing the stack frames; allocating variable live ranges to architectural registers; scheduling instructions to exploit micro-architecture; and producing assembly source or object code.

Historically, the 1986 edition of the "Compilers Principles, Techniques, and Tools" Dragon Book by Aho et al. lists the tasks of code generation as:

- Instruction selection and lowering of calling conventions.
- Control-flow (dominators, loops) and data-flow (variable liveness) analyses.
- Register allocation and stack frame building.
- Peephole optimizations.

A. Cohen (Ed.): CC 2014, LNCS 8409, pp. 1–17, 2014.

Ten years later, the 1997 textbook "Advanced Compiler Design & Implementation" by Muchnich extends code generation with the following tasks:

- Loop unrolling and basic block replication.
- Instruction scheduling and software pipelining.
- Branch optimizations and basic block alignment.

In current releases of high-end compilers such as Open64 or GCC, code generation techniques have significantly evolved, as they are mainly responsible for exploiting the performance-oriented features of architectures and micro-architectures. In these compilers, code generator optimizations include:

- If-conversion using SELECT, conditional move, or predicated, instructions.
- Use of specialized addressing modes such as auto-modified and modulo.
- Exploitation of hardware looping or static branch prediction hints.
- Matching fixed-point arithmetic and SIMD idioms to special instructions.
- Memory hierarchy optimizations, including pre-fetching and pre-loading.
- VLIW instruction bundling, that may interfere with instruction scheduling.

This sophistication of modern compiler code generation motivates the introduction of the SSA form on the program representation in order to simplify some of the analyses and optimizations. In particular, liveness analysis, unrolling-based loop optimizations, and exploitation of special instructions or addressing modes benefit significantly from the SSA form. On the other hand, the SSA form does not apply after register allocation, and there is still debate as to whether it should be used in the register allocator [3].

In this paper, we review some of the issues of inserting the SSA form in a code generator, based on experience with a family of code generators and linear assembly optimizers for the ST120 DSP core [21] [20,49,44], the Lx/ST200 VLIW family [23] [17,18,8,7,5], and the Kalray VLIW core [19]. Section 2 presents the challenges of maintaining the SSA form on a program representation based on machine instructions. Section 3 discusses two code generator optimizations that seem at odds with the SSA form, yet must occur before register allocation. One is if-conversion, whose modern formulations require an extension of the SSA form. The other is pre-pass instruction scheduling, which currently does not seem to benefit from the SSA form. Going in and out of SSA form in a code generator is required in such case, so Section 4 characterizes various SSA form destruction algorithms with regards to satisfying the constraints of machine code.

2 SSA Form Engineering Issues

2.1 Instructions, Operands, Operations, and Operators

An *instruction* is a member of the machine instruction set architecture (ISA). Instructions access values and modify the machine state through *operand*s. We distinguish *explicit operands*, which are associated with a specific bit-field in the instruction encoding, from *implicit operands*, without any encoding bits.

Explicit operands correspond to allocatable architectural registers, immediate values, or instruction modifiers. Implicit operands correspond to single instance architectural registers and to registers implicitly used by some instructions, such as the status register, the procedure link register, or even the stack pointer.

An *operation* is an instance of an instruction that composes a program. It is seen by the compiler as an *operator* applied to a list of operands (explicit & implicit), along with operand naming constraints, and has a set of clobbered registers. The compiler view of operations also involves *indirect operands*, which are not apparent in the instruction behavior, but are required to connect the flow of values between operations. Implicit operands correspond to the registers used for passing arguments and returning results at function call sites, and may also be used for the registers encoded in register mask immediates.

2.2 Representation of Instruction Semantics

Unlike IR operators, there is no straightforward mapping between machine instructions and their operational semantics. For instance, a subtract with operands (a, b, c) may either compute $c \leftarrow a - b$ or $c \leftarrow b - a$ or any such expression with permuted operands. Yet basic SSA form code cleanups such as constant propagation and sign extension removal need to know what is actually computed by machine instructions. Machine instructions may also have multiple target operands, such as memory accesses with auto-modified addressing, or combined division-modulus instructions. There are two ways to address this issue.

- Add properties to the instruction operator and to its operands, a technique used by the Open64 compiler. Operator properties include *isAdd*, *isLoad*, etc. Typical operand properties include *isLeft*, *isRight*, *isBase*, *isOffset*, *isPredicated*, etc. Extended properties that involve the operator and some of its operands include *isAssociative*, *isCommutative*, etc.
- Associate a *semantic combinator*, that is, a tree of IR-like operators, to each target operand of a machine instruction. This more ambitious alternative was implemented in the SML/NJ [35] compiler and the LAO compiler [20].

An issue related to the representation of instruction semantics is how to factor it. Most information can be statically tabulated by the instruction operator, yet properties such as safety for control speculation, or being equivalent to a simple IR instruction, can be refined by the context where the instruction appears. For instance, range propagation may ensure that an addition cannot overflow, that a division by zero is impossible, or that a memory access is safe for control speculation. Alternate semantic combinators, or modifiers of the instruction operator semantic combinator, need to be associated with each machine instruction of the code generator internal representation.

Finally, code generation for some instruction set architectures require that pseudo-instructions with known semantics be available, besides variants of ϕ-functions and parallel COPY operations.

- Machine instructions that operate on register pairs, such as the long multiplies on the ARM, or more generally on register tuples, are common. In such cases there is a need for pseudo-instructions to compose wide operands in register tuples, and to extract independently register allocatable operands from wide operands.
- Embedded architectures such as the Tensilica Xtensa provide hardware loops, where an implicit conditional branch back to the loop header is taken whenever the program counter matches some address. The implied loop-back branch is also conveniently materialized by a pseudo-instruction.
- Register allocation for predicated architectures requires that the live-ranges of pseudo-registers or SSA variables with predicated definitions be contained by kill pseudo-instructions [26].

2.3 Operand Naming Constraints

Implicit operands and indirect operands are constrained to specific architectural registers either by the instruction set architecture (ISA constraints), or by the application binary interface (ABI constraints). An effective way to deal with such *dedicated register* naming constraints in the SSA form is by inserting parallel COPY operations that write to the constrained source operands, or read from the constrained target operands of instructions. The new SSA variables thus created are pre-colored with the required architectural register. With modern SSA form destruction [48,7], COPY operations are aggressively coalesced, and the remaining ones are sequentialized into machine operations.

Explicit instruction operands may be constrained to use the same resource (an unspecified architectural register) between a source and a target operand, as illustrated by most x86 instructions and by DSP-style auto-modified addressing modes. A related naming constraint is to require different resources between two source operands, as with the MUL instructions on the ARM. The *same resource* naming constraints are represented under the SSA form by inserting a COPY operation between the constrained source operand and a new variable, then using this new variable as the constrained source operand. In case of multiple constrained source operands, a parallel COPY operation is used. Again, these COPY operations are processed by the SSA form destruction.

A wider case of operand naming constraint is when a variable must be bound to a specific architectural register at all points in the program. This is the case with the stack pointer, as interrupt handling may reuse the run-time stack at any program point. One possibility is to inhibit the promotion of the stack pointer to a SSA variable. Stack pointer definitions including memory allocations through `alloca()`, activation frame creation/destruction, are then encapsulated as instances of a specific pseudo-instruction. Instructions that use the stack pointer must be treated as special cases for the SSA form analyses and optimizations.

2.4 Non-kill Target Operands

The SSA form requires that variable definitions be kills. This is not the case for target operands such as a status register that contains several independent

bit-fields. Moreover, some instruction effects on bit-field may be *sticky*, that is, with an implied OR with the previous value. Typical sticky bits include exception flags of the IEEE 754 arithmetic, or the integer overflow flag on DSPs with fixed-point arithmetic. When mapping a status register to a SSA variable, any operation that partially reads or modifies the register bit-fields should appear as reading and writing the corresponding variable.

Predicated execution and conditional execution are other sources of definitions that do not kill their target register. The execution of predicated instructions is guarded by the evaluation of a single bit operand. The execution of conditional instructions is guarded by the evaluation of a condition on a multi-bit operand. We extend the ISA classification of [39] to distinguish four classes:

Partial Predicated Execution Support. SELECT instructions, first introduced by the Multiflow TRACE architecture [14], are provided. The Multiflow TRACE 500 architecture was to include predicated store and floating-point instructions [37].

Full Predicated Execution Support. Most instructions accept a Boolean predicate operand which nullifies the instruction effects if the predicate evaluates to false. EPIC-style architectures also provide predicate define instructions (PDIs) to efficiently evaluate predicates corresponding to nested conditions: Unconditional, Conditional, parallel-OR, parallel-AND [26].

Partial Conditional Execution Support. Conditional move (CMOV) instructions, first introduced by the Alpha AXP architecture [4], are provided. CMOV instructions are available in the ia32 ISA since the Pentium Pro.

Full Conditional Execution Support. Most instructions are conditionally executed depending on the evaluation of a condition of a source operand. On the ARM architecture, the implicit source operand is a bit-field in the status register and the condition is encoded on 4 bits. On the VelociTI™ TMS230C6x architecture [47], the source operand is a general register encoded on 3 bits and the condition is encoded on 1 bit.

2.5 Program Representation Invariants

Engineering a code generator requires decisions about what information is transient, or belongs to the invariants of the program representation. By invariant we mean a property which is ensured before and after each phase. Transient information is recomputed as needed by some phases from the program representation invariants. The applicability of the SSA form only spans the early phases of the code generation process: from instruction selection, down to register allocation. After register allocation, program variables are mapped to architectural registers or to memory locations, so the SSA form analyses and optimizations no longer apply. In addition, a program may be only partially converted to the SSA form. This motivates the engineering of the SSA form as extensions to a baseline code generator program representation.

Some extensions to the program representation required by the SSA form are better engineered as invariants, in particular for operands, operations, basic

blocks, and control-flow graph. Operands which are SSA variables need to record the unique operation that defines them as a target operand, and possibly to maintain the list of where they appear as source operands. Operations such as ϕ-functions, σ-functions of the SSI form [6], and parallel copies may appear as regular operations constrained to specific places in the basic blocks. The incoming arcs of basic blocks need also be kept in the same order as the source operands of each of its ϕ-functions.

A program representation invariant that impacts SSA form engineering is the structure of loops. The modern way of identifying loops in a CFG is the construction of a loop nesting forest as defined by Ramalingam [43]. Non-reducible control-flow allows for different loop nesting forests for a given CFG, yet high-level information such as loop-carried memory dependences, or user-level loop annotations, are provided to the code generator. This information is attached to a loop structure, which thus becomes an invariant. The impact on the SSA form is that some loop nesting forests, such as the Havlak [29] loop structure, are better than others for key analyses such as SSA variable liveness [5].

Up-to-date live-in and live-out sets at basic block boundaries are also candidates for being program representation invariants. However, when using and updating liveness information under the SSA form, it appears convenient to distinguish the ϕ-function contributions from the results of dataflow fix-point computation. In particular, Sreedhar et al. [48] introduced the ϕ-function semantics that became later known as *multiplexing mode*,where a ϕ-function $B_0 : a_0 = \phi(B_1 : a_1, \ldots, B_n : a_n)$ makes a_0 live-in of basic block B_0, and $a_1, \ldots a_n$ live-out of basic blocks $B_1, \ldots B_n$. The classic basic block invariants LiveIn(B) and LiveOut(B) are then complemented with PhiDefs(B) and PhiUses(B) [5].

Finally, some compilers adopt the invariant that the SSA form be *conventional* across the code generation phases. This approach is motivated by the fact that classic optimizations such as SSA-PRE [32] require that 'the live ranges of different versions of the same original program variable do not overlap', implying the SSA form is conventional. Other compilers that use SSA numbers and omit the ϕ-functions from the program representation [34] are similarly constrained. Work by Sreedhar et al. [48] and by Boissinot et al. [7] clarified how to convert the transformed SSA form conventional wherever required, so there is no reason nowadays for this property to be an invariant.

3 Code Generation Phases and the SSA Form

3.1 Classic If-conversion

If-conversion refers to optimizations that convert a program region to straight-line code. It is primarily motivated by instruction scheduling on instruction-level parallel cores [39], as removing conditional branches enables to:

- eliminate branch resolution stalls in the instruction pipeline,
- reduce uses of the branch unit, which is often single-issue,
- increase the size of the instruction scheduling regions.

In case of inner loop bodies, if-conversion further enables vectorization [1] and software pipelining (modulo scheduling) [41]. Consequently, control-flow regions selected for if-conversion are acyclic, even though seminal techniques [1,41] consider more general control-flow.

The scope and effectiveness of if-conversion depends on the ISA support. In principle, any if-conversion technique targeted to full predicated or conditional execution support may be adapted to partial predicated or conditional execution support. For instance, non-predicated instructions with side-effects such as memory accesses can be used in combination with SELECT to provide a harmless effective address in case the operation must be nullified [39].

Besides predicated or conditional execution, architectural support for if-conversion is improved by supporting speculative execution. Speculative execution (control speculation) refers to executing an operation before knowing that its execution is required, such as when moving code above a branch [37] or promoting operation predicates [39]. Speculative execution assumes instructions have reversible side effects, so speculating potentially excepting instructions requires architectural support. On the Multiflow TRACE 300 architecture and later on the Lx VLIW architecture [23], non-trapping memory loads known as *dismissible* are provided. The IMPACT EPIC architecture speculative execution [2] is generalized from the *sentinel* model [38].

The classic contributions to if-conversion did not consider the SSA form.

Allen et al. [1] convert control dependences to data dependences, motivated by inner loop vectorization. They distinguish forward branches, exit branches, and backward branches, and compute Boolean guards accordingly. As this work predates the Program Dependence Graph [24], complexity of the resulting Boolean expressions is an issue. When comparing to later if-conversion techniques, only the conversion of forward branches is relevant.

Park & Schlansker [41] propose the RK algorithm based the control dependences. They assume a fully predicated architecture with only Conditional PDIs. The R function assigns a minimal set of Boolean predicates to basic blocks, and the K function express the way these predicates are computed. The algorithm is general enough to process cyclic and irreducible rooted flow graphs, but it practice it is applied to single entry acyclic regions.

Blickstein et al. [4] pioneer the use of CMOV instructions to replace conditional branches in the GEM compilers for the Alpha AXP architecture.

Lowney et al. [37] match the innermost if-then constructs in the Multiflow Trace Scheduling compiler in order to generate the SELECT and the predicated memory store operations.

Fang [22] assumes a fully predicated architecture with Conditional PDIs. The proposed algorithm is tailored to acyclic regions with single entry and multiple exits, and as such is able to compute R and K functions without relying

on explicit control dependences. The main improvement of this algorithm over [41] is that it also speculates instructions up the dominance tree through predicate promotion, except for stores and PDIs. This work further proposes a pre-optimization pass to hoist or sink common sub-expressions before predication and speculation.

Leupers [36] focuses on if-conversion of nested if-then-else (ITE) statements on architectures with full conditional execution support. A dynamic programming technique appropriately selects either a conditional jump or a conditional instruction based implementation scheme for each ITE statement, and the objective is the reduction of worst-case execution time (WCET).

A few contributions to if-conversion did use the SSA form but only internally.

Jacome et al. [31] propose the Static Single Assignment - Predicated Switching (SSA-PS) transformation aimed at clustered VLIW architectures, with predicated move instructions that operate inside clusters (internal moves) or between clusters (external moves). The first idea of the SSA-PS transformation is to realize the conditional assignments corresponding to ϕ-functions via predicated switching operations, in particular predicated move operations. The second idea is that the predicated external moves leverage the penalties associated with inter-cluster data transfers. The SSA-PS transformation predicates non-move operations and is apparently restricted to innermost if-then-else statements.

Chuang et al. [13] introduce a predicated execution support aimed at removing non-kill register writes from the micro-architecture. They propose SELECT instructions called *phi-ops*, predicated memory accesses, Unconditional PDIs, and ORP instructions for OR-ing multiple predicates. A restriction of the RK algorithm to single-entry single-exit regions is proposed, adapted to the Unconditional PDIs and the ORP instructions. Their other contribution is the generation of phi-ops, whose insertion points are computed like the SSA form placement of the ϕ-functions. The ϕ-functions source operands are replaced by ϕ-lists, where each operand is associated with the predicate of its source basic block. The ϕ-lists are processed by topological order of the predicates to generate the phi-ops.

3.2 If-conversion under SSA Form

The ability to perform if-conversion on the SSA form of a program representation requires the handling of operations that do not kill the target operand because of predicated or conditional execution.

Stoutchinin & Ferrière [49] introduce ψ-functions in order to represent fully predicated code under the SSA form, which is then called the ψ-SSA form. The ψ-functions arguments are paired with predicates and are ordered in dominance order in the ψ-function argument list, a correctness condition re-discovered by Chuang et al. [13] for their phi-ops.

Stoutchinin & Gao [50] propose an if-conversion technique based on the predication of Fang [22] and the replacement of ϕ-functions by ψ-functions. They prove the conversion is correct provided the SSA form is conventional. The technique is implemented in Open64 for the ia64 architecture.

Bruel [10] targets VLIW architectures with SELECT and dismissible load instructions. The proposed framework reduces acyclic control-flow constructs from innermost to outermost, and the monitoring of the if-conversion benefits provides the stopping criterion. The core technique control speculates operations, reduces height of predicate computations, and performs tail duplication. It can also generate ψ-functions instead of SELECT operations.

Ferrière [25] extends the ψ-SSA form algorithms of [49] to architectures with partial predicated execution support, by formulating simple correctness conditions for the predicate promotion of operations that do not have side-effects. This work also details how to transform the ψ-SSA form to conventional ψ-SSA form by generating CMOV operations.

Thanks to these contributions, virtually all if-conversion techniques formulated without the SSA form can be adapted to the ψ-SSA form, with the added benefit that already predicated code may be part of the input. In practice, these contributions follow the generic steps of if-conversion proposed by Fang [22]:

- if-conversion region selection;
- code hoisting and sinking of common sub-expressions;
- assignment of predicates to the basic blocks;
- insertion of operations to compute the basic block predicates;
- predication or speculation of operations;
- and conditional branch removal.

The result of an if-converted region is a hyper-block, that is, a sequence of basic blocks with predicated or conditional operations, where control may only enter from the top, but may exit from one or more locations [40].

Although if-conversion based on the ψ-SSA form appears effective for the different classes of architectural support, the downstream phases of the code generator require at least some adaptations of the plain SSA form algorithms to handle the ψ-functions. The largest impact of handling ψ-function is apparent in the ψ-SSA form destruction [25], whose original description [49] was incomplete.

In order to avoid such complexities, the Kalray VLIW code generator adopts simpler solution than ψ-functions to represent the non-kill effects of conditional operations on target operands. This solution is based on the observation that under the SSA form, a CMOV operation is equivalent to a SELECT operation with a same resource naming constraint between one source and the target operand. Unlike other predicated or conditional instructions, a SELECT instruction kills its target register. Generalizing this observation provides a simple way to handle predicated or conditional operations in plain SSA form:

- For each target operand of the predicated or conditional instruction, add a corresponding source operand in the instruction signature.
- For each added source operand, add a same resource naming constraint with the corresponding target operand.

This simple transformation enables the SSA form analyses and optimizations to remain oblivious to predicated or conditional code. The drawback of this solution is that non-kill definitions of a given variable (before SSA variable renaming) remain in dominance order across program transformations, as opposed to ψ-SSA where predicate value analysis may enable this order to be relaxed.

3.3 Pre-pass Instruction Scheduling

Further down the code generator, the last major phase before register allocation is pre-pass instruction scheduling. Innermost loops with a single basic block, super-block or hyper-block body are candidates for software pipelining techniques such as modulo scheduling [45]. For innermost loops that are not software pipelined, and for other program regions, acyclic instruction scheduling techniques apply: basic block scheduling [27]; super-block scheduling [30]; hyper-block scheduling [40]; tree region scheduling [28]; or trace scheduling [37].

By definition, pre-pass instruction scheduling operates before register allocation. At this stage, instruction operands are mostly virtual registers, except for instructions with ISA or ABI constraints that bind them to specific architectural registers. Moreover, preparation to pre-pass instruction scheduling includes virtual register renaming, also known as register web construction, in order to reduce the number of anti dependences and output dependences in the instruction scheduling problem. Other reasons why it seems there is little to gain from scheduling instructions on a SSA form of the program representation include:

- Except in case of trace scheduling which pre-dates the use of SSA form in production compilers, the classic scheduling regions are single-entry and do not have control-flow merge. So there are no ϕ-functions in case of acyclic scheduling, and only ϕ-functions in the loop header in case of software pipelining. Keeping those ϕ-functions in the scheduling problem has no benefits and raises engineering issues, due to their parallel execution semantics and the constraint to keep them first in basic blocks.
- Instruction scheduling must account for all the instruction issue slots required to execute a code region. If the only ordering constraints between instructions, besides control dependences and memory dependences, are limited to true data dependences on operands, code motion will create interferences that must later be resolved by inserting COPY operations in the scheduled code region. (Except for interferences created by the overlapping of live ranges that results from modulo scheduling, as these are resolved by modulo renaming [33].) So scheduling instructions with SSA variables as operands is not effective unless extra dependences are added to the scheduling problem to prevent such code motion.

- Some machine instructions have partial effects on special resources such as the status register. Representing special resources as SSA variables even though they are accessed at the bit-field level requires coarsening the instruction effects to the whole resource, as discussed in Section 2.4. In turn this implies def-use variable ordering that prevents aggressive instruction scheduling. For instance, all sticky bit-field definitions can be reordered with regards to the next use, and an instruction scheduler is expected to do so. Scheduling OR-type predicate define operations [46] raises the same issues. An instruction scheduler is also expected to precisely track accesses to unrelated or partially overlapping bit-fields in a status register.
- Aggressive instruction scheduling relaxes some flow data dependences that are normally implied by SSA variable def-use ordering. A first example is *move renaming* [51], the dynamic switching of the definition of a source operand defined by a COPY operation when the consumer operations ends up being scheduled at the same cycle or earlier. Another example is *inductive relaxation* [16], where the dependence between additive induction variables and their use as base in base+offset addressing modes is relaxed to the extent permitted by the induction step and the range of the offset. These techniques apply to acyclic scheduling and to modulo scheduling.

To summarize, trying to keep the SSA form inside the pre-pass instruction scheduling appears more complex than operating on the program representation with classic compiler temporary variables. This representation is obtained after SSA form destruction and aggressive coalescing. If required by the register allocation, the SSA form should be re-constructed.

4 SSA Form Destruction Algorithms

The destruction of the SSA form in a code generator is required before the pre-pass instruction scheduling and software pipelining, as discussed earlier, and also before non-SSA register allocation. A weaker form is the conversion of transformed SSA form to conventional SSA form, which is required by classic SSA form optimizations such as SSA-PRE [32] and SSA form register allocators [42]. For all such cases, the main objective besides removing the SSA form extensions from the program representation is to ensure that the operand naming constraints are satisfied. Another objective is to avoid critical edge splitting, as this interferes with branch alignment [12], and is not possible on some control-flow edges of machine code such as hardware loop back edges.

The contributions to SSA form destruction techniques can be characterized as an evolution towards correctness, the ability to manage operand naming constraints, and the reduction of algorithmic time and memory requirements.

Cytron et al. [15] describe the process of *translating out of SSA* as 'naive replacement preceded by dead code elimination and followed by coloring'. They replace each ϕ-function $B_0 : a_0 = \phi(B_1 : a_1, \ldots, B_n : a_n)$ by n copies $a_0 = a_i$, one per basic block B_i, before applying Chaitin-style coalescing.

Briggs et al. [9] identify correctness issues in Cytron et al. [15] out of (transformed) SSA form translation and illustrate them by the *lost-copy problem* and the *swap problem*. These problems appear in relation with the critical edges, and because a sequence of ϕ-functions at the start of a basic block has parallel assignment semantics [7]. Two SSA form destruction algorithms are proposed, depending on the presence of critical edges in the control-flow graph. However the need for parallel COPY operations is not recognized.

Sreedhar et al. [48] define the ϕ-congruence classes as the sets of SSA variables that are transitively connected by a ϕ-function. When none of the ϕ-congruence classes have members that interfere, the SSA form is called *conventional* and its destruction is trivial: replace all the SSA variables of a ϕ-congruence class by a temporary variable, and remove the ϕ-functions. In general, the SSA form is *transformed* after program optimizations, that is, some ϕ-congruence classes contain interferences. In Method I, the SSA form is made conventional by inserting COPY operations that target the arguments of each ϕ-function in its predecessor basic blocks, *and also* by inserting COPY operations that source the target of each ϕ-function in its basic block. The latter is the key for not depending on critical edge splitting [7]. The code is then improved by running a new SSA variable coalescer that grows the ϕ-congruence classes with COPY-related variables, while keeping the SSA form conventional. In Method II and Method III, the ϕ-congruence classes are initialized as singletons, then merged while processing the ϕ-functions in some order. In Method II, two variables of the current ϕ-function that interfere directly or through their ϕ-congruence classes are isolated by inserting COPY operations for both. This ensures that the ϕ-congruence class which is grown from the classes of the variables related by the current ϕ-function is interference-free. In Method III, if possible only one COPY operation is inserted to remove the interference, and more involved choices about which variables to isolate from the ϕ-function congruence class are resolved by a maximum independent set heuristic. Both methods are correct except for a detail about the live-out sets to consider when testing for interferences [7].

Leung & George [35] are the first to address the problem of satisfying the same resource and the dedicated register operand naming constraints of the SSA form on machine code. They identify that Chaitin-style coalescing after SSA form destruction is not sufficient, and that adapting the SSA optimizations to enforce operand naming constraints is not practical. They operate in three steps: collect the renaming constraints; mark the renaming conflicts; and reconstruct code, which adapts the SSA destruction of Briggs et al. [9]. This work is also the first to make explicit use of parallel COPY operations.

Budimlić et al. [11] propose a lightweight SSA form destruction motivated by JIT compilation. It uses the (strict) SSA form property of dominance of variable definitions over uses to avoid the maintenance of an explicit interference graph. Unlike previous approaches to SSA form destruction that coalesce increasingly larger sets of non-interfering ϕ-related (and COPY-related) variables, they first

construct SSA-webs with early pruning of obviously interfering variables, then de-coalesce the SSA webs into non-interfering classes. They propose the *dominance forest* explicit data-structure to speed-up these interference tests. This SSA form destruction technique does not handle the operand naming constraints, and also requires critical edge splitting.

Rastello et al. [44] revisit the problem of satisfying the *same resource* and *dedicated register* operand constraints of the SSA form on machine code, motivated by erroneous code produced by the technique of Leung & George [35]. Inspired by work of Sreedhar et al. [48], they include the ϕ-related variables as candidates in the coalescing that optimizes the operand naming constraints. This work avoids the patent of Sreedhar et al. (US patent 6182284).

Boissinot et al. [7] analyze the previous contributions to SSA form destruction to their root principles, and propose a generic approach to SSA form destruction that is proved correct, handles operand naming constraints, and can be optimized for speed. The foundation of the approach is to transform the program to conventional SSA form by isolating the ϕ-functions like in Method I of Sreedhar et al. [48]. However, the COPY operations inserted are parallel, so a parallel COPY sequentialization algorithm is provided. The task of improving the conventional SSA form is then seen as a classic aggressive variable coalescing problem, but thanks to the SSA form the interference relation between SSA variables is made precise and frugal to compute. Interference is obtained by combining the intersection of SSA live ranges, and the equality of values which is easily tracked under the SSA form across COPY operations. Moreover, the use of the dominance forest data-structure of Budimlić et al. [11] to speed-up interference tests between congruence classes is obviated by a linear traversal of these classes in pre-order of the dominance tree. Finally, the same resource operand constraints are managed by pre-coalescing, and the dedicated register operand constraints are represented by pre-coloring the congruence classes. Congruence classes with a different pre-coloring always interfere.

5 Summary and Conclusions

The target independent program representations of high-end compilers are nowadays based on the SSA form, as illustrated by the Open64 WHIRL, the GCC GIMPLE, or the LLVM IR. However support of the SSA form in the code generator program representations is more challenging. The main issues to address are the mapping of SSA variables to special architectural resources, the management of instruction set architecture (ISA) or application binary interface (ABI) operand naming constraints, and the representation of non-kill effects on the target operands of machine instructions. Moreover, adding the SSA form attributes and invariants to the program representations appears detrimental to the pre-pass instruction scheduling (including software pipelining).

The SSA form benefits most the phases of code generation that run before pre-pass instruction scheduling. In particular, we review the different approaches to

if-conversion, a key enabling phase for the exploitation of instruction-level parallelism by instruction scheduling. Recent contributions to if-conversion leverage the SSA form but introduce ψ-functions in order to connect the partial definitions of predicated or conditional machine operations. This approach effectively extends the SSA form to the ψ-SSA form, which is more complicated to handle especially in the SSA form destruction phase.

We propose a simpler alternative for the representation of non-kill target operands without the ψ-functions, allowing the early phases of code generation to operate on the standard SSA form only. This proposal requires that the SSA form destruction phase be able to manage operand naming constraints. This motivated us to extend the technique of Sreedhar et al. (SAS'99), the only one at the time that was correct, and which did not require critical edge splitting. Eventually, this work evolved into the technique of Boissinot et al. (CGO'09).

References

1. Allen, J.R., Kennedy, K., Porterfield, C., Warren, J.: Conversion of control dependence to data dependence. In: Proc. of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1983, pp. 177–189 (1983)
2. August, D.I., Connors, D.A., Mahlke, S.A., Sias, J.W., Crozier, K.M., Cheng, B.C., Eaton, P.R., Olaniran, Q.B., Hwu, W.M.W.: Integrated predicated and speculative execution in the impact epic architecture. In: Proc. of the 25th Annual International Symposium on Computer Architecture, ISCA 1998, pp. 227–237 (1998)
3. Barik, R., Zhao, J., Sarkar, V.: A decoupled non-ssa global register allocation using bipartite liveness graphs. ACM Trans. Archit. Code Optim. 10(4), 63:1–63:24 (2013)
4. Blickstein, D.S., Craig, P.W., Davidson, C.S., Faiman Jr., R.N., Glossop, K.D., Grove, R.B., Hobbs, S.O., Noyce, W.B.: The GEM optimizing compiler system. Digital Technical Journal 4(4), 121–136 (1992)
5. Boissinot, B., Brandner, F., Darte, A., de Dinechin, B.D., Rastello, F.: A non-iterative data-flow algorithm for computing liveness sets in strict ssa programs. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 137–154. Springer, Heidelberg (2011)
6. Boissinot, B., Brisk, P., Darte, A., Rastello, F.: SSI properties revisited. ACM Trans. on Embedded Computing Systems (2012); special Issue on Software and Compilers for Embedded Systems
7. Boissinot, B., Darte, A., Rastello, F., de Dinechin, B.D., Guillon, C.: Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency. In: CGO 2009: Proc. of the 2009 International Symposium on Code Generation and Optimization, pp. 114–125 (2009)
8. Boissinot, B., Hack, S., Grund, D., de Dinechin, B.D., Rastello, F.: Fast Liveness Checking for SSA-Form Programs. In: CGO 2008: Proc. of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 35–44 (2008)
9. Briggs, P., Cooper, K.D., Harvey, T.J., Simpson, L.T.: Practical Improvements to the Construction and Destruction of Static Single Assignment Form. Software – Practice and Experience 28, 859–881 (1998)

10. Bruel, C.: If-Conversion SSA Framework for partially predicated VLIW architectures. In: ODES 4, pp. 5–13 (March 2006)
11. Budimlic, Z., Cooper, K.D., Harvey, T.J., Kennedy, K., Oberg, T.S., Reeves, S.W.: Fast copy coalescing and live-range identification. In: Proc. of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, PLDI 2002, pp. 25–32. ACM, New York (2002)
12. Calder, B., Grunwald, D.: Reducing branch costs via branch alignment. In: Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS VI, pp. 242–251. ACM, New York (1994)
13. Chuang, W., Calder, B., Ferrante, J.: Phi-predication for light-weight if-conversion. In: Proc. of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO 2003, pp. 179–190 (2003)
14. Colwell, R.P., Nix, R.P., O'Donnell, J.J., Papworth, D.B., Rodman, P.K.: A vliw architecture for a trace scheduling compiler. In: Proc. of the Second International conference on Architectual Support for Programming Languages and Operating Systems, ASPLOS-II, pp. 180–192 (1987)
15. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Trans. on Programming Languages and Systems 13(4), 451–490 (1991)
16. de Dinechin, B.D.: A unified software pipeline construction scheme for modulo scheduled loops. In: Malyshkin, V.E. (ed.) PaCT 1997. LNCS, vol. 1277, pp. 189–200. Springer, Heidelberg (1997)
17. de Dinechin, B.D.: Time-Indexed Formulations and a Large Neighborhood Search for the Resource-Constrained Modulo Scheduling Problem. In: 3rd Multidisciplinary International Scheduling Conference: Theory and Applications, MISTA (2007)
18. Dupont de Dinechin, B.: Inter-Block Scoreboard Scheduling in a JIT Compiler for VLIW Processors. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 370–381. Springer, Heidelberg (2008)
19. de Dinechin, B.D., Ayrignac, R., Beaucamps, P.E., Couvert, P., Ganne, B., de Massas, P.G., Jacquet, F., Jones, S., Chaisemartin, N.M., Riss, F., Strudel, T.: A clustered manycore processor architecture for embedded and accelerated applications. In: IEEE High Performance Extreme Computing Conference, HPEC 2013, pp. 1–6 (2013)
20. de Dinechin, B.D., de Ferrière, F., Guillon, C., Stoutchinin, A.: Code Generator Optimizations for the ST120 DSP-MCU Core. In: CASES 2000: Proc. of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 93–102 (2000)
21. de Dinechin, B.D., Monat, C., Blouet, P., Bertin, C.: Dsp-mcu processor optimization for portable applications. Microelectron. Eng. 54(1-2), 123–132 (2000)
22. Fang, J.Z.: Compiler algorithms on if-conversion, speculative predicates assignment and predicated code optimizations. In: Sehr, D., Banerjee, U., Gelernter, D., Nicolau, A., Padua, D. (eds.) LCPC 1996. LNCS, vol. 1239, pp. 135–153. Springer, Heidelberg (1997)
23. Faraboschi, P., Brown, G., Fisher, J.A., Desoli, G., Homewood, F.: Lx: A Technology Platform for Customizable VLIW Embedded Processing. In: ISCA 2000: Proc. of the 27th Annual Int. Symposium on Computer Architecture, pp. 203–213 (2000)
24. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (1987)

25. de Ferrière, F.: Improvements to the Psi-SSA representation. In: Proc. of the 10th International Workshop on Software & Compilers for Embedded Systems, SCOPES 2007, pp. 111–121 (2007)
26. Gillies, D.M., Ju, D.C.R., Johnson, R., Schlansker, M.: Global predicate analysis and its application to register allocation. In: Proc. of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 29, pp. 114–125 (1996)
27. Goodman, J.R., Hsu, W.C.: Code scheduling and register allocation in large basic blocks. In: Proc. of the 2nd International Conference on Supercomputing, ICS 1988, pp. 442–452 (1988)
28. Havanki, W., Banerjia, S., Conte, T.: Treegion scheduling for wide issue processors. In: International Symposium on High-Performance Computer Architecture, 266 (1998)
29. Havlak, P.: Nesting of reducible and irreducible loops. ACM Trans. on Programming Languages and Systems 19(4) (1997)
30. Hwu, W.M.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Ouellette, R.G., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., Lavery, D.M.: The superblock: An effective technique for vliw and superscalar compilation. J. Supercomput. 7(1-2), 229–248 (1993)
31. Jacome, M.F., de Veciana, G., Pillai, S.: Clustered vliw architectures with predicated switching. In: Proc. of the 38th Design Automation Conference, DAC, pp. 696–701 (2001)
32. Kennedy, R., Chan, S., Liu, S.M., Lo, R., Tu, P., Chow, F.: Partial redundancy elimination in ssa form. ACM Trans. Program. Lang. Syst. 21(3), 627–676 (1999)
33. Lam, M.: Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In: PLDI 1988: Proc. of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, pp. 318–328 (1988)
34. Lapkowski, C., Hendren, L.J.: Extended ssa numbering: introducing ssa properties to languages with multi-level pointers. In: Proc. of the 1996 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON 1996, pp. 23–34. IBM Press (1996)
35. Leung, A., George, L.: Static single assignment form for machine code. In: Proc. of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, PLDI 1999, pp. 204–214 (1999)
36. Leupers, R.: Exploiting conditional instructions in code generation for embedded vliw processors. In: Proc. of the Conference on Design, Automation and Test in Europe, DATE 1999 (1999)
37. Lowney, P.G., Freudenberger, S.M., Karzes, T.J., Lichtenstein, W.D., Nix, R.P., O'Donnell, J.S., Ruttenberg, J.: The multiflow trace scheduling compiler. J. Supercomput. 7(1-2), 51–142 (1993)
38. Mahlke, S.A., Chen, W.Y., Hwu, W.M.W., Rau, B.R., Schlansker, M.S.: Sentinel scheduling for vliw and superscalar processors. In: Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-V, pp. 238–247 (1992)
39. Mahlke, S.A., Hank, R.E., McCormick, J.E., August, D.I., Hwu, W.M.W.: A comparison of full and partial predicated execution support for ilp processors. In: Proc. of the 22nd Annual International Symposium on Computer Architecture, ISCA 1995, pp. 138–150 (1995),
40. Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., Bringmann, R.A.: Effective compiler support for predicated execution using the hyperblock. SIGMICRO Newsl. 23(1-2), 45–54 (1992)

41. Park, J.C., Schlansker, M.S.: On predicated execution. Tech. Rep. HPL-91-58, Hewlett Packard Laboratories, Palo Alto, California (1991)
42. Pereira, F.M.Q., Palsberg, J.: Register allocation by puzzle solving. In: Proc. of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, PLDI 2008, pp. 216–226. ACM (2008)
43. Ramalingam, G.: On loops, dominators, and dominance frontiers. ACM Trans. on Programming Languages and Systems 24(5) (2002)
44. Rastello, F., de Ferrière, F., Guillon, C.: Optimizing Translation Out of SSA Using Renaming Constraints. In: CGO 2004: Proc. of the International Symposium on Code Generation and Optimization, pp. 265–278 (2004)
45. Rau, B.R.: Iterative modulo scheduling. International Journal of Parallel Programming 24(1), 3–65 (1996)
46. Schlansker, M., Mahlke, S., Johnson, R.: Control cpr: A branch height reduction optimization for epic architectures. In: Proc. of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, PLDI 1999, pp. 155–168 (1999)
47. Seshan, N.: High velociti processing. IEEE Signal Processing Magazine, 86–101 (1998)
48. Sreedhar, V.C., Ju, R.D.C., Gillies, D.M., Santhanam, V.: Translating Out of Static Single Assignment Form. In: SAS 1999: Proc. of the 6th International Symposium on Static Analysis, pp. 194–210 (1999)
49. Stoutchinin, A., de Ferrière, F.: Efficient Static Single Assignment Form for Predication. In: Proc. of the 34th Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 34, pp. 172–181 (2001)
50. Stoutchinin, A., Gao, G.: If-Conversion in SSA Form. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 336–345. Springer, Heidelberg (2004)
51. Young, C., Smith, M.D.: Better global scheduling using path profiles. In: Proc. of the 31st Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 31, pp. 115–123 (1998)