- 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 prepass 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)