Decomposition of Parallel Copies with Duplication

G. N. Purohit[1], Venuka Sandhir[1]

[1]Centre for Mathematical Sciences, Banasthali University, Banasthali, Rajasthan, India

Abstract

SSA form is becoming more popular in the context of JIT compilation since it allows the compiler to perform important optimizations like common sub-expression elimination or constant propagation without the drawbacks of keeping huge data structures in memory or requiring a lot of computing power. The recent approach of SSA-based register allocation performs SSA elimination after register allocation. F. Bouchez *et al.* proposed parallel copy motion to prevent the splitting of edges when going out of colored SSA by moving the code that should be assigned to the edges to a more convenient place. Duplications in parallel copies pose some problems when moving them. In this paper an approach has been developed to decompose parallel copies so that duplications can be handled separately and parallel copies can be easily moved away without duplication. A simple and elegant application is moving duplicated copies out of critical edges. This is often beneficial compared to the alternative splitting the edge.

Keywords: *Register Allocation, SSA form, Critical edge, Parallel copy.*

1. Introduction

Register allocation is among the most important compiler optimizations affecting the performance of compiled code. It determines which of the program values (variables and temporaries) should be in machine registers (or memory) during the execution of a program.

In a real machine, registers are usually few and fast to access, so the problem addressed here is how to minimize the traffic between registers and memory hierarchy. Therefore, the challenge is to relegate the least program values to memory. Data dependencies in most programming languages are implicit. Some compilers use an Intermediate Representation in Static Single Assignment (SSA) in which each variable is only defined once to simplify analysis of data dependencies. The properties of the underlying dominance tree [1] and the implied use-def chains make possible the use of efficient, simple and fast algorithms for various code optimizations in SSA.

SSA-form contains ϕ-functions to merge values based on control flow. Once optimizations on SSA-form are performed, it is not trivial to translate SSA-form back to normal form because the properties of ϕ-functions cannot be translated directly to processor instructions and must be disposed off. Recently, solutions have been proposed to perform register assignment—assign variables to registers—while still under SSA and then try to go out of colored SSA. This is the case of Hack *et al.* [12], Hack and Goos [11] or Bouchez *et al.* [5] for instance. There are some advantages of this practice:

- copies are implicit: there is no need to add new copies and variables corresponding to a naïve out-of-SSA conversion.
- the dominance property can be exploited to perform a greedy coloring algorithm (using a “tree-scan” on the dominance tree for instance).
- SSA possesses nice properties avoiding use of an interference graph for coloring (liveness information is cheap, see Boissinot *et al.* [3]).
- one might use the SSA to perform other optimizations after coloring, like code motion or scheduling.

Performing the coloring of the variables under SSA i.e. doing register allocation before translating out of SSA, fit the scheme of register allocation in two separate steps, first spilling then splitting and finally coloring with coalescing [4]. The classical techniques to go out-of-SSA that insert copies at the beginning or end of basic blocks as do Sreedhar *et al.* [15] are too constrained in this case by the fact that variables cannot be created on demand. The alternative solution [5] is to place parallel copies corresponding to ϕ-functions on the incoming edges. The solution is then to convert parallel copies into permutations that are easier to move.

In light of previous work, the goal of this paper is to propose a general framework for moving around parallel copies with duplications in a register-allocated code. The remainder of the paper is organized as follows: Section 2 summarizes parallel copy motion inside a basic block and out of a control-flow edge. Section 3 illustrates the concept of compensation code with the notion of critical edges i.e. edges going from a block with multiple successors to a block with multiple predecessors. Section 4 describes our approach for moving a parallel copy away from critical edge with duplications and Section 5 concludes the results and future work.

Copyright (c) 2012 International Journal of Computer Science Issues. All Rights Reserved.

2. Parallel Copies

As explained earlier, special care should be taken when going out of colored SSA. A correct way to do it is to replace the ϕ-functions by parallel copies on the incoming edges. Parallel copies are virtual instructions that perform as many move instructions as required, all at the same time. Parallel copy is a fundamental instruction when dealing with program splitting. To split all variables at one program point, one needs to duplicate all variables alive at this point and insert a parallel copy between all the variables and their duplicates. Trying to split by inserting normal i.e. sequentialized copies would create interferences between some variables and the duplicates of others. Parallel copies can be seen as a way to "reorganize" values in variables and are sometimes referred to as "shuffle code." However, there is no such hardware instruction. At best, one can find instructions to swap values in registers or perform up to a fixed number of copies in parallel.

In this paper, we deal with colored variables and use the notation $x^{< R_i >}$ to state that variable x is assigned to register R_i. In our case, the ϕ-functions represent a flow of values between registers instead of a flow of values between variables. Moreover, there is no way of creating "temporary variables" as only registers are available. Still, some registers might be free if the register pressure is lower than R. They can be temporarily used to keep some values.

When dealing with colored variables, it is handy to implement parallel copies as arrays. Let $//c$ be a parallel copy, then for each register R_i ($1 \leq i \leq R$), the i-th element of the array (indexed from 1 to R) indicates the register from which new value of R_i will be copied i.e. if $//c(R_i) = R_j$, the value of register R_j is copied into R_i during the parallel copy. A register that simply holds its value is represented as $//c(R_i) = R_i$ and one that does not receive any live value is represented as $//c(R_i) = \bot$. Note that it is important to differentiate in the parallel copy representation, registers that are not modified ($//c(R_i) = R_i$) from register that do not receive any value ($//c(R_i) = \bot$), since the latter are "free" registers and the former hold the value of live variables. In this paper a graphical representation of parallel copies is used in which registers are nodes and directed edges represent the flow of the values. These are called "register transfer graphs", Hack [10].

In general, R_i holds the value of a live variable before the parallel copy iff there exists $1 \leq i \leq R$ such that $//c(R_i) = R_j$ i.e. an edge leaving the node j in the graph representation. Register R_i holds the value of a live variable after the parallel copy iff $//c(R_i) \neq \bot$ i.e. there exists an edge entering the node i in the graph representation. If $j = i$ the value stays in the same register, which is represented by a self edge. Moreover, we have considered that two registers containing the values of live variables at one point interfere at this point, even if the values are the same. Hence it is forbidden that a parallel copy defines a register more than once: two different registers cannot put their values into same i.e. there should not be two entering edges in a node of the graph representation of the parallel copy.

When performing SSA-based register allocation: ϕ-functions are removed after the register assignment phase, which leads, due to the semantics of these functions to the introduction of register-to-register parallel copies on the edges leading to the ϕ-functions. Fig. 1 illustrates an example where R_1 is copied to R_2 and R_3 on the left edge from B_s to B_d, because the left arguments of the ϕ-functions are in different registers than the variables defined. On the contrary, variables b and c is assigned to R_2 and R_3 on the right edge from $B_{s'}$ to B_d, so the values of R_2 and R_3 should remain there.

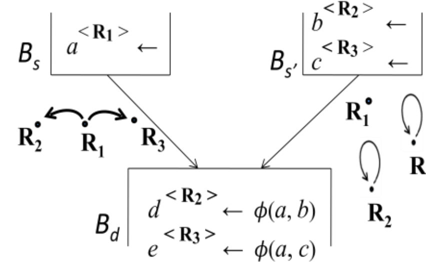


Fig. 1 Example of parallel copies where a must be duplicated because d and e interfere.

If a parallel copy is injective i.e. there is no two $i \neq j$ such that $//c(R_i) = //c(R_j)$, it is reversible in the mathematical sense. A *reversible parallel copy* $//c$ is a one-to-one mapping from its *live-in* set $\{s_i\}$ to its *live-out* set $\{d_i\}$. We use the notation $//c : (d_1,\ldots, d_n) \leftarrow (s_1,\ldots, s_n)$ where $//c(s_i) = d_i$ and $//c^{-1}(d_i) = s_i$. However, the $\bot$ value makes it difficult for parallel copies to be reversible. Indeed, it means that $//c$ is not injective whenever more than one register does not receive the value of another register.

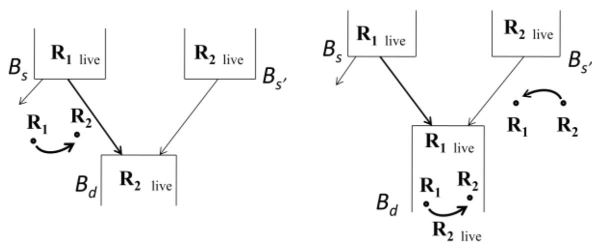
The *live-in* and *live-out* sets are subsets of the register set. Note that these two sets are not necessarily disjoint. Hence, care must be taken to implement the mapping with sequential instructions (possibly with swaps). For $R_i \notin$ *live-in*, we abusively write $//c(R_i) = \bot$ and, for $R_i \notin$ *live-out*, $//c^{-1}(R_i) = \bot$. Since parallel copies contain liveness information, it is not possible to replace $//c^{-1} \circ //c$ by $//c \circ //c^{-1}$ in the general case. Indeed, *live-out* of $//c$ is the same as *live-in* of $//c^{-1}$ but is in general different

Copyright (c) 2012 International Journal of Computer Science Issues. All Rights Reserved.

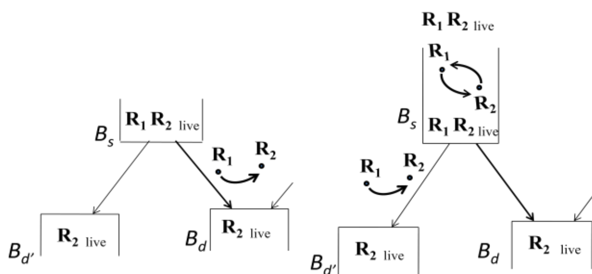
than the *live-in* of //*c* (which in turn is the same as *live-out* of $//c^{-1}$). In terms of graph representation, a parallel copy is a set of disjoint sub-graphs, where each sub-graph is a chain or a simple cycle [14].

3. Compensation

When trying to move a parallel copy away from an edge E from source basic block B_s to destination basic block B_d, there are two possibilities: either move it up i.e. to the bottom of B_s, or move it down, i.e., to the top of B_d. Indeed, the code of the parallel copy needs to be performed whenever the execution path goes through the edge E. This works fine whenever E is the only edge leaving B_s or the only one entering B_d. However if E is a critical edge, it is in general false to move the parallel copy either on B_s or on B_d. Indeed, the code of the parallel copy should not be executed if an edge other than E is chosen. A well-known example of the problem of critical edges is the "lost copy problem". In fact, in the case of a colored SSA code, simpler examples exist: it is sufficient that a parallel copy overwrite a register containing another live variable on the predecessor block. However, it is possible to compensate its effects on the other edges leaving B_s (resp. entering B_d) by placing some code on them.



(a) Moving the parallel copy //*c* down



(b) Moving the parallel copy //*c* up

Fig. 2 The differences from moving down or up come from the asymmetry of liveness between the source and the destination of edges.

When moving a parallel copy //*c* down, one needs to pre-compensate the copy on other incoming edges. The idea is to "prepare" the registers so that, when arriving on B_d, //*c* will move the values into their right registers. This is correct because the sets of variables alive are the same at the beginning of B_d and on every edge arriving in B_d. The fact that *live-out*(//c) = *live-in*($//c^{-1}$) and *live-in*(//*c*) = *live out*($//c^{-1}$) shows that, in the end, the effect is that: if the flow comes from another edge than E, $//c^{-1}$ is followed by //*c*, i.e., the identity is done on the registers alive.

On the contrary, moving //*c* up is more difficult. On (b), //*c* needs to be modified to take the liveness at the end of B_s into account. Indeed, //*c* was created with the liveness of E in mind, which is a subset of *live-out*(B_s). That is why, in the examples of Fig. 2, the parallel copy was modified when moved up and not when moved down. On (b), both R_1 and R_2 contain a live value on *Bs*. R_2 must be saved so registers are swapped. Then, R_2 must be restored with its original value on the left edge.

As explained, this forces to be careful in order not to erase any value of a live variable and not to add useless copies during compensation, while it is possible to augment or project parallel copies when required, a more elegant solution [5] in which parallel copies are converted to permutations: //*c* is made a bijection by replacing the every ⊥ by a well-chosen register in the array representation.

In fact, permutation motion can be viewed in [5], more generally, as region recoloring, a technique that allows permutation to be moved not only from control-flow edges but also inside basic blocks. In the presence of non-splittable critical edges, the permutation motion can sometimes fail: if the number of duplications exceeds the register pressure and in the presence of multiplexing regions, in this case classical graph coloring techniques are used to recolor the multiplexing regions, possibly requiring additional spills. Nevertheless, in practice, the compiler hardly generates such regions, thus it does not appear to be an issue for performance.

4. Duplications in parallel copies

Parallel copies can contain duplications, i.e., the value in one register is copied into two registers (or more), as R_1 indicates duplication in the Fig.1. More formally, there is a duplication if for a register R_k every time there is two registers R_i and R_j ($i \neq j$) such that $//c(R_i) = R_k$ and $//c(R_j) = R_k$. The value contained in register R_k is duplicated. This happens for instance if, at the beginning of a basic block, the same variable is used twice as argument, as in [$d \leftarrow (a,\ldots.)$; $e \leftarrow \phi(a,\ldots.)$] or if two

Copyright (c) 2012 International Journal of Computer Science Issues. All Rights Reserved.

arguments have been coalesced and renamed into one variable.

One particularity of duplications is that the register pressure (minimal number of registers necessary to allocate all the variables live at that program point) is higher after a parallel copy that contains duplications than it was before. This is one of the reasons why special care must be taken when dealing with them. In a parallel copy with duplication, the flow edges of values between register cannot be "reversed" to obtain a parallel copy that has an effect opposite to that of the first one (else, there would be two edges pointing to the same register).They cannot be ignored as they are mandatory in many actual cases. In practice, the duplications can be extracted from the parallel copies and placed in the predecessor basic block but this task may lead to additional spilling. For these reasons, we will try to get rid of duplications, which is being done as follows:

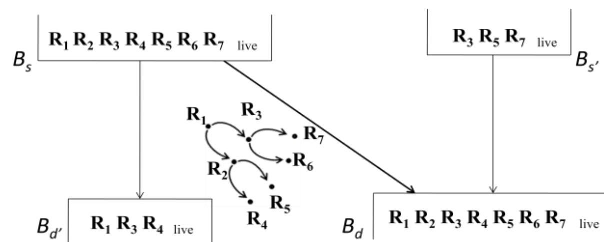
KEY POINTS OF MOVING A PARALLEL COPY AWAY FROM CRITICAL EDGE WITH DUPLICATIONS

An overview of the general process is given before detailing each individual step. Conceptually, our approach comprises four successive phases:

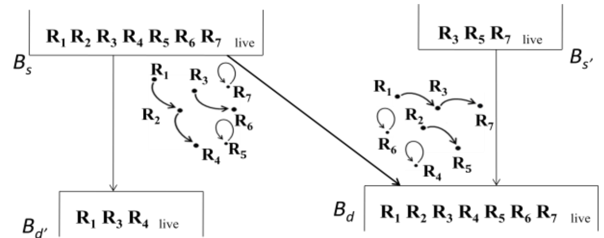
- Identify the duplications in Register Flow Graph (RFG).
- Decomposition of a parallel copy containing duplications.
- Moving parallel copies away from an edge.
- Insert compensation code.

A parallel copy *//c* is considered on an edge *E* that cannot be split, or we do not want to split. First, it has been calculated that how many times a register is used to define another one in *//c*. If it is used more than twice, register contains duplication. For simplicity, it has been considered that any register in the graph representation of the parallel copy has at most two leaving edges. Decomposition of a parallel copy containing duplications would be easier if the RFG contains "free" registers: they do not contain the value of any live variable at the end of B_s (so they also do not on the edge of *//c*) and do not receive any value in *//c* or that receive a value but whose value is not used. If duplication consists of a self edge, it does not need to be copied (but then, every other duplication involving that register does). Taking all live registers at the end of B_s, it is possible to decompose the RFG by considering one leaving edge of each duplicated register in edge-disjoint sub-graphs: RFG equals the union of the flows of registers of sub-graphs. Moving parallel copies *//c* while inserting the compensation code on adjacent edges are described in Section 3.

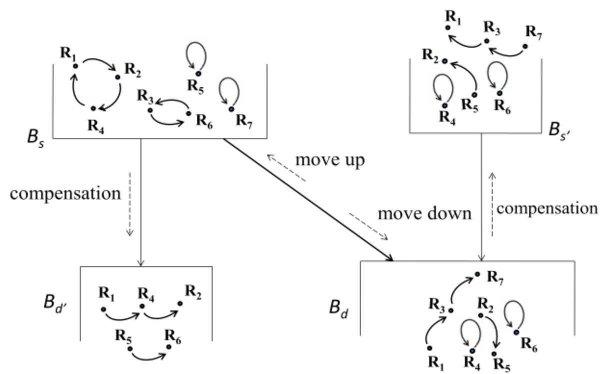
As illustrated by this example (Fig. 3), when trying to move a reversible parallel copy away from a critical edge E: $B_s \rightarrow B_d$ with duplications R_1, R_2 and R_3. First decompose the RFG into sub-graphs, directed the flows of registers: $R_4 \leftarrow R_2 \leftarrow R_1$, $R_6 \leftarrow R_3$, $R_5 \leftarrow R_5$, $R_7 \leftarrow R_7$ and $R_7 \leftarrow R_3 \leftarrow R_1$, $R_5 \leftarrow R_2$, $R_4 \leftarrow R_4$, $R_6 \leftarrow R_6$. Moving the latter parallel copies to the top of B_d and pre-compensated the copy on other incoming edges and another to the bottom of B_s and post-compensated the copy on other out going edges.



(a) Parallel copies with duplications



(b) Decomposition of a parallel copy containing duplications



(c) Moving parallel copies with compensation

Fig. 3 Example of moving a parallel copy away from a critical edge with duplication. First, *//c* is decomposed and then the decomposed copies can be moved with compensation code.

Copyright (c) 2012 International Journal of Computer Science Issues. All Rights Reserved.

5. Conclusion and Future Work

The goal of this paper is to present an approach to prevent the splitting of edges when going out of colored SSA by moving the duplicated code that should be on edges to a more convenient place. Our solution is based on an idea that, to our knowledge, is known in the literature: parallel copies can be moved away from edges provided that compensation code is inserted on other edges. Duplications just make copies of registers. So, as long as there are enough free registers, it is possible to move duplications. It is indeed not a problem to change the value of a register that will not be used on the other successor blocks. The only restriction is that duplications should not erase any live value, so an approach has been developed to decompose parallel copies so that duplications can be handled separately.

Better approach could be devised for the decomposition, using for instance information on the place where *//c* will be sequentialized. This is not our purpose here. Our purpose is just to show that it is not a problem to place parallel copies with duplications on edges as these can be moved away from edges that cannot be split or that one does not want to split, not to provide the best way to do it, if there is one.

Our plan for future research includes design and implementation of an algorithm that uses the approach presented in this paper. Implementation of different heuristics to improve the precision of this algorithm without sacrificing the compilation time will also be done.

References

[1] A. W. Appel and J. Palsberg, “Modern Compiler Implementation in Java”, 2nd edition, Cambridge University Press, 2002.

[2] B. Boissinot, A. Darte, F. Rastello, B. D. de Dinechin and C. Guillon, “Revisiting out-of-ssa translation for correctness, code quality and efficiency”, In Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO ’09, Washington, DC, USA, 2009, pp. 114–125.

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

[4] F. Bouchez, “A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases” Ph.D. Thesis. December 22, 2008.

[5] F. Bouchez, Q. Colombet, A. Darte, F. Rastello and C. Guillon, “Parallel copy motion” In SCOPES, ACM, 2010, pp. 1-10.

[6] P. Briggs, “Register allocation via graph coloring” Ph.D. thesis, Rice University, Houston, TX, USA, 1992.

[7] 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, Jul. 1998, pp. 859–881.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, “Efficiently com-puting static single assignment form and the control dependence graph”, ACM Transactions on Programming Languages and Systems, Vol. 13, No. 4, 1991, pp. 451 – 490.

[9] L. George and A. W. Appel, “Iterated register coalescing”, ACM Transactions on Programming Languages and Systems, Vol. 18, No. 3, May 1996.

[10] S. Hack, “Register Allocation for Programs in SSA Form”, PhD thesis, Universität Karlsruhe, Oct. 2007.

[11] S. Hack and G. Goos, “Copy coalescing by graph recoloring”, In ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI’08), 2008, pp. 227–237.

[12] S. Hack, D. Grund and G. Goos, “ Register allocation for programs in SSA form”, In International Conference on Compiler Construction (CC’06), Volume 3923 of LNCS. Springer, 2006, pp. 247-262.

[13] F. M. Q. Pereira and J. Palsberg, “Register allocation via coloring of chordal graphs”, In Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS’05), Tsukuba, Japan, Nov. 2005, pp. 315–329.

[14] F. M. Q. Pereira and J. Palsberg, “SSA elimination after register allocation”, In CC, 2009, pp. 158 -173.

[15] 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), 1999, pp. 194 – 204.

Prof. G. N. Purohit is a Dean of Apaji Institute of Mathematics and Applied Computer Technology at Banasthali University, Rajasthan. Before joining Banasthali University, he was Head of the Department of Mathematics, University of Rajasthan, Jaipur. He had been Chief-editor of a research journal and regular reviewer of many journals. His present interest is in Operational Research, Discrete Mathematics and Communication and Sensor Networks. He has published around 40 research papers in various journals.

Venuka Sandhir earned her B.Sc. and M.Sc. degree in Applied Mathematics from University of Delhi. Currently she is doing her M.Phil. in the Mathematical Sciences from Banasthali University at the Centre for Mathematical Sciences. Her research interests include Numerical Simulation, Graph Theory, Computer and Communication.

Copyright (c) 2012 International Journal of Computer Science Issues. All Rights Reserved.