An LLVM Implementation of SSAPRE

Tanya Brethour Joel Stanley Bill Wendling
University of Illinois at Urbana-Champaign
{tbrethou,jstanley,jwendlin}@uiuc.edu

December 8, 2002

1 Introduction

One of the primary goals of a compiler is to eliminate redundant computations present in the input program. Such redundancy elimination is especially beneficial in loops, since eliminating computations from a frequently executed region of code can lead to massive performance gains in the program overall. Two independent compiler optimizations are customarily used to eliminate redundancies: Global Common Subexpression Elimination (GCSE) and Loop Invariant Code Motion (LICM). GCSE essentially replaces computation sites with a saved version of a computation, provided that the value of the computation has not been altered since the last time it was made. Loop-invariant code motion is responsible for hoisting loop-invariant computations from the body of a loop or loop nest, provided that it can safely do so.

However, neither LICM nor GCSE can handle *partial* redundancies: redundant computations that occur on some execution paths reaching a given point, but not on others. Addressing this deficiency is the goal of a powerful dataflow-based optimization known as Partial Redundancy Elimination (PRE). PRE effectively subsumes both LICM and GCSE, in addition to safely transforming partial redundancies to full redundancies, which can then be removed.

In this report we present implementation details, empirical performance data, and notable modifications to an algorithm for PRE based on [1]. In [1], a particular realization of PRE, known as SSAPRE, is described, which is more efficient than traditional PRE implementations because it relies on useful properties of Static Single-Assignment (SSA) form to perform dataflow analysis in a much more sparse manner than the traditional bit-vector-based approach. Our implementation is specific to a SSA-based compiler infrastructure known as LLVM (Low-Level Virtual Machine).

This paper describes the current state of our implementation using the LLVM infrastructure, and delineates important modifications to the algorithm described in [1].

2 Existing Work

PRE was first developed by Morel and Renvoise [1979]. Their implementation used dataflow analysis to determine partial redundancies and eliminate them. This method was enhanced by the introduction of a code placement strategy called lazy code motion (LCM) [3], which finds the optimal placement for code within a control flow graph (CFG). However, the previous versions of PRE are based on a bit-vector formulation of the problem and on the iterative solution of data flow equations[1]. The primary drawback to the application of bit-vector-based dataflow optimizations to an SSA intermediate representation is the high cost of representational conversion. In order to propagate the dataflow predicates properly, the IR is essentially taken out of SSA form prior to the analysis and put back into SSA form after, a process which incurs high compile-time cost.

The SSAPRE paper provides an SSA-based version of PRE which combines the optimal placement properties of the previous algorithms for PRE with SSA's sparse use-definition information. In particular, it leverages features of SSA such as the single-assignment property and dominance invariants so that PRE analysis costs are greatly reduced in comparison to the traditional approaches.[1]

3 Overview

3.1 Definitions

We first present a few definitions from [1] which we use throughout this paper:

Definition (Redundant): If E_1 and E_2 are occurrences of some computation E and there is a control flow path from E_1 to E_2 containing nothing that may alter the value of E, we say that E_2 is *redundant with respect to* E_1.[1]

Definition (Partially Available): We say a computation is *partially available* at some point p in the program if there is a control flow path leading to p from some real occurrence of the computation and not crossing anything that may alter the value of the computation.[1]

Definition (Partially Redundant): We say an occurrence ω is *partially redundant* if it is an occurrence of a computation that is partially available just before ω.[1]

Definition (Φ): In the same way that the literature uses a ϕ operator in SSA form to factor the use-def relation for variables, we introduce a Φ operator that factors the redundancy relation for computation occurrences.[1]

Definition ($\perp$): There can be operands of Φ that are not partially redundant; these have no counterpart in SSA form, and we denote them by the symbol $\perp$.[1]

Definition (Representative Occurrence): We define the *representative occurrence* for an expression to be the nearest expression that is either a Φ Occurrence or a non-partially redundant real occurrence that dominates the expression. [1]

3.2 SSAPRE Algorithm

The paper presents two versions of the SSAPRE algorithm. The first version provides everything necessary to create a working version of SSAPRE for a compiler. There are six steps in the algorithm: Φ *Insertion*, *Rename*, *DownSafety*, *WillBeAvail*, *Finalize*, and *CodeMotion*. However, this version isn't sparse (there are potentially extraneous Φ nodes placed into the graph, and the naive rename algorithm considers many versions of variables that may not appear in any PRE candidate expression) and deals with all of the expressions in the program simultaneously, which can induce a large memory footprint.

The second version of the algorithm is a practical implementation of SSAPRE. It is a worklist driven version of the algorithm and requires a prepass over the code to collect all lexically identified occurrences of expressions into lexically equivalent sets. Once this is done, however, we no longer need to look at all of the code again but only at the collected occurrences. Each collected occurrence set is placed into the worklist then removed one at a time so that the algorithm can be applied to it. The practical implementation algorithm replaces the first two parts of the initial algorithm – Φ *Insertion* and *Rename* – with a demand-driven version of Φ *Insertion* and a delayed version of *Rename*. See Figure 1 for a graphical representation of the implementation of the worklist driven algorithm.

We chose to implement the worklist driven version of the algorithm.

4 Implementation

While we chose to implement the worklist driven version of the SSAPRE algorithm, our implementation doesn't actually use the worklist in the way a traditional worklist is used. In the paper, the worklist is needed for "compound" expressions (those of the form $a + b - c$, where $a + b$ is a subexpression of the whole expression). LLVM is a three-address representation and doesn't allow for compound expressions.

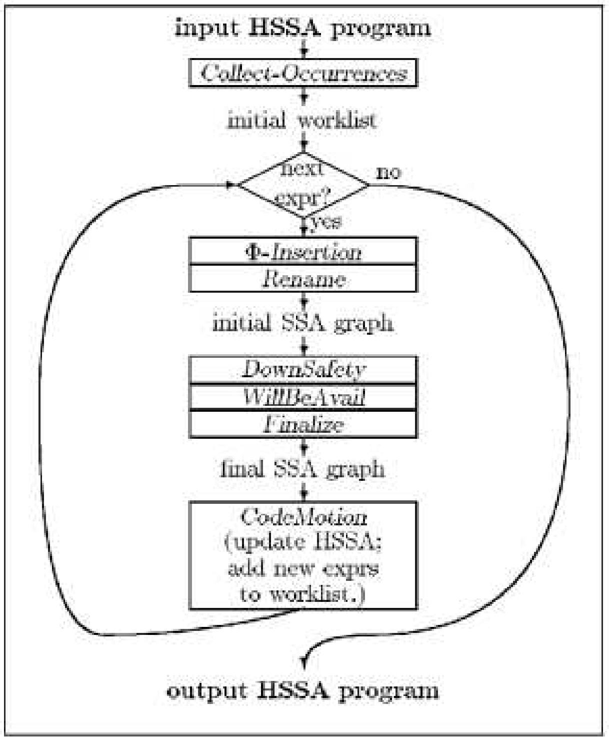

Figure 1: PRE Worklist Driven Approach from [1]

4.1 Assumptions

The assumptions that we make are as follows (the first two assumptions are directly stated in the paper, and the latter two can easily be inferred):

- "Each ϕ assignment has the property that its left-hand side and all of its operands are versions of the same original program variable"[1]
- "The live ranges of different versions of the same original program variable do not overlap"[1]
- All critical edges are broken; and
- We have access to the dominator tree and dominance frontiers of basic blocks

Because of the first two assumptions, we need to disable running a few optimization passes before our pass is run. In particular, `instcombine, mem2reg,` and `reassociate` shouldn't be run as they could potentially violate the first two assumptions. Of course, since our pass is supposed to subsume GCSE and LICM, both `gcse` and `licm` shouldn't be run. In order to ensure that critical edges have been broken, the break-crit-edges is required before our pass.

4.2 Data Structures

Our implementation of the algorithm is based on infrastructure that consists of a hierarchy of Occurrence classes. There are 5 types of occurrences: Real, Φ, Φ Operand, Exit, and Inserted; their instances track any information about them that the paper specifies as necessary. The main Occurrence class maintains information that is shared by all types of occurrences. This information includes: the basic block it exists in, the instruction it represents, the cached temporary instruction that saves the result (if any), and the Redundancy Class Number (RCN). The Real Occurrence class has additional flags associated with it such as: Reload, Save, and a pointer to its representative Occurrence. Φ Occurrences have flags to indicate whether they are downsafe, extraneous, “canbeavail”, or are “later”. Φ Operand occurrences maintain what their representative occurrence is, what Φ they belong to, what instruction would be inserted there if needed, and a flag indicating if they have a real use. Inserted Occurrences and Exit Occurrences do not store any additional information.

4.3 Pass Details

The SSAPRE algorithm is done in six separate phases, which are discussed in detail below.

4.3.1 Collect Occurrences

The main idea behind the Collect Occurrences phase is to identify lexically identified[1] expressions to partition them in to equivalence classes (also referred to as “occurrence sets”). Unfortunately, the authors do not describe any information how this is best accomplished.

Since LLVM does not explicitly represent the “SSA version” of a particular “original” program variable, we discover versions of the same SSA variable by examining where values are merged by ϕ-nodes in the SSA representation. Whenever we witness a ϕ-node in the linear scan of the program instructions[2], we consider the def of the ϕ and its operands “equivalent” for purposes of hashing expressions to the proper lexically-identified equivalence classes. The hashing step can be made more effective through the previous application of reassociation, but we have not fully explored the efficacy of such reassociation, as the LLVM reassociation pass has the potential to violate the ϕ-operand deadness invariant required by the SSAPRE algorithm. For efficiency, we use a Union-Find mechanism with both Union-By-Rank and Path Compression, so that the runtime cost of determining different versions of the same variable is a *small* “constant”[3] for any conceivable program size.[4]

4.3.2 Φ *Insertion*

Our implementation uses the demand-driven Φ *Insertion* algorithm[1]. “The resulting algorithm is sparse in the sense that all the Φs inserted are justified either by appearing in the iterated dominance frontier of some real occurrence of the expression or by appearing at a point where the expression is partially anticipated.”[1]

With LLVM, sparse Φ insertion is easy. Using the dominance frontier information supplied by LLVM, we determine the iterated dominance frontier (IDF) for the basic block of an expression’s real occurrence. That is, we get the initial dominance frontier for the original basic block and then recurse on each individual basic block in that dominance frontier adding to the IDF if it isn’t in there already. Also, LLVM provides quick access to the definitions of operands in expressions, so determining if they were SSA ϕ nodes or not is simple. This involves a recursive step on the ϕ node to see if its operands were defined by ϕ nodes or not.

4.3.3 *Rename*

The primary purpose of the *Rename* pass is to assign redundancy class numbers (RCNs) to each occurrence which places them into equivalence classes. Two occurrences with equivalent RCNs have the same value throughout the program. That is, they are a *refinement* of the occurrence sets, which are populated with lexically identified expressions. Furthermore, it is straightforward to conclude that any two occurrences along a control path with two different RCNs will have a redefinition of one of their variables at some point between the occurrences on that path. The secondary purpose of *Rename* is to construct the Factored Redundancy Graph (FRG). The FRG is defined as a collection of real occurrences, and Φś in the same redundancy class, which represent the nodes. Upward edges in the FRG are from each partially redundant Φ Operand or Real Occurrence to their representative occurrence.

The non-worklist driven approach for *Rename* is modeled after the SSA Renaming algorithm in [5], modified slightly to maintain a stack for each expression in addition to stacks for each variable. The sole

[1]Recall that two expressions are lexically identified if their respective operands are versions of the same program variable. Thus, $a_5 + b_4$ is lexically identified with $a_0 + b_{19}$

[2]This is the only time the entire program is visited by SSAPRE

[3]That is, the Inverse Ackerman’s function

purpose of having the version stacks for the variables is to determine whether or not a new RCN needs to be assigned. Because performing rename this way requires the examination of many versions of variables that may not appear in any PRE candidate expression, the algorithm is not sparse. Thus, Kennedy et al. presents an alternative algorithm called Delayed Renaming[1].

For pure redundancy class assignment, *Delayed Renaming* uses a redundancy class stack for the expression being analyzed. *Delayed Renaming* maintains the invariant that, at any point during analysis, the top of the stack represents the current RCN and the representative occurrence node for the expression. Each RCN has a representative occurrence, which means that we can safely replace other occurrences with the same RCN and still maintain the original program semantics. This is due to the property expressed above that two occurrences of the same RCN have the same value. A representative occurrence is always a real occurrence or a Φ Occurrence, and Φ Occurrences always get a new RCN (since they represent a merge of expression computations), so there are only four situations that might arise when attempting to assign a RCN to an Occurrence:

1. The top of the stack is a Real Occurrence and
    (a) Our current occurrence is a Real
    (b) Our current occurrence is a Φ Operand
2. The top of the stack is a Φ Occurrence and
    (a) Our current occurrence is a Real
    (b) Our current occurrence is a Φ Operand

Delayed Renaming is performed in two steps. The first, *Rename1*, processes each Occurrence separately, pushing items onto the stack when they are assigned a new RCN, and popping items if they do not dominate the current occurrence. If the top of the stack is a Real Occurrence, we have the current version of the variables available and assigning a new RCN is as simple as comparing those versions to the current occurrence. In LLVM there is no notion of versions, so this is equivalent to performing comparisons of each operand's Value pointer. If the top of the stack is a Φ Occurrence, the versions of variables are not provided. To resolve this issue, *Rename1* uses dominance information to determine which RCN is appropriate. This dominance relation is that if all variable definitions of the current occurrence dominate the Φ Occurrence at the top of the stack, then the versions are identical[1].

However, there is one small detail overlooked in *Rename1*. When the current Occurrence is a Φ Operand, there exists no Real Occurrence which can provide us with the current versions of the variables. In these cases, Rename1 makes an optimistic assumption and assumes that the top of the redundancy class stack provides its variables versions and therefore is given the same RCN. This assumption is either correct or the Φ operand will have no representative occurrence, $\perp$. Having no representative occurrence means that the Φ Operand is not partially redundant. *Rename1* keeps track of each Real Occurrence that is defined by a Φ and places them into a set to be processed. This set is processed by *Rename2* which corrects the optimistic assumption regarding Φ operands if necessary.

Rename2 processes each item in the set constructed by *Rename1*. Each of these Real Occurrences are defined by a Φ and provides the versions of the variables at that Φ that defines it. *Rename2* first obtains the Φ for the Real Occurrence and notes what basic block it resides in. If there exists a ϕ for any of the variables of that Real Occurrence in the basic block of its defining Φ, we must double check our optimistic assumption made to the Φ operands.

For each Φ Operand, a Real Occurrence is manufactured with the correct versions of the variables at that point. The ϕ for the variable provides us with the version to use when manufacturing this real occurrence. The manufactured Real Occurrence is compared to the representative occurrence for the Φ Operands. If the representative occurrence is a Real Occurrence then pointers are compared. If it is a Φ Occurrence, we check if all the definitions of the variables in the manufactured occurrence dominate that Φ. If not in either case, the optimistic assumption was indeed wrong and the Φ Operand is set to $\perp$. If the RCN is determined to be correct and the representative occurrence is a Φ, the manufactured occurrence needs to be added to the set for further processing in order to ensure that the operands of that Φ are also correct.

It is important to note that the paper did not explain how to create this manufactured real occurrence, nor how its def edge ought to be set. Initially it seemed as simple as cloning the Real Occurrence, but later proved to be more complicated because a critical detail was simply left out in the algorithm. The edge in the FRG from this manufactured occurrence must not be an exact copy, but should be to the representative occurrence for the Φ Operand being examined. It is critical to recursively check Φ Occurrences and their operands as mentioned above.

Upon completion, *Delayed Renaming* will have assigned RCNs, and created FRGs for each redundancy class of the variable. This first pass is crucial to the success of the algorithm as a whole and during our implementation and testing, several bugs have been linked back to this pass due to its complexity.

4.3.4 *DownSafety*

In order for PRE to insert a computation it must be down safe or fully anticipated at the point of insertion[1]. Down safety is used to ensure that new exceptions or redundancy are not introduced by inserting an expression. Since insertions are only done at Φ Operands, it is sufficient to determine down safety only at Φ Occurrences. Note that it is only safe to do so because we require critical edges to be broken. DownSafety is done in two steps: Initialization and Propagation. In addition to determining DownSafety, this pass also sets the hasRealUse flags for Φ Operand Occurrences.

In order for a Φ Occurrence to not be down safe, there must exist a control flow path from the Φ such that the expression is either not computed prior to an exit or is not computed prior to a redefinition of one of its operands[1]. Because Rename is already processing the Occurrences in DT preorder, it is an appropriate place to perform the initialization. While the paper gives excellent details on what modifications to make to the non-worklist driven rename algorithm, it does not give any information on how to modify the delayed rename pass. Therefore, it took a significant amount of time to come up with the correct approach.

All downsafety flags are initialized to true, which is an optimistic assumption. Down safety can only be set to false if we see an Exit Occurrence before a Real Occurrence, or before a Φ Operand that is defined by a down safe Φ. The paper suggests that whenever Rename assigns a Real Occurrence a new RCN, sets a Φ Operand to $\perp$ or encounters a program exit, it checks the top of the stack to see if it is a Φ Occurrence. If so, it will reset that Φ's downsafety flag. This approach seems reasonable, except in delayed Renaming Occurrences are only pushed onto the stack if they are assigned a new redundancy class number. However, there are cases where a Real Occurrence is given the same RCN as a Φ or another Real Occurrence and consequently it is not pushed onto the stack. If a Real Occurrence is not pushed onto the stack and the next Occurrence is an Exit Occurrence, it presents a problem during down safety initialization. It will mark the Φ not down safe because it never witnessed the Real Occurrence at the top of the stack. This is also a problem for setting hasRealUse flags, where Φ Operands will not see a Real Occurrence on the top of the stack and their hasRealUse flags will be incorrectly set to false.

It is not sufficient to have only the stack described in Rename1 and still initialize the downsafety and hasRealUse flags appropriately. Therefore, we keep another stack that keeps track of all of the Occurrences that dominate the current expression and that have been processed. When determining if down safety should be reset or hasRealUse should be set, it looks at the top of this second stack. We avoid the problem of missing a Real Occurrence that was not assigned a new RCN.

The second part of downsafety is to propagate a non-down safe value to any Φ's that have operands that use the non-down safe Φ as their representative occurrence. This is a simple walk of the FRG.

4.3.5 *WillBeAvail*

The *WillBeAvail* step tells us if a value will be available at a Φ occurrence following insertions. If the Φ has its *will_be_avail* predicate set but a value isn't available there, later steps will insert an occurrence of the expression at this point. This, combined with the *DownSafety* step, gives us the optimal placement for new expressions in the final CFG.

The algorithm starts where *DownSafety* ends. It calculates if a value can be available at a Φ and whether or not it can be placed "later" in the CFG. The algorithm recursively visits each Φ node either clearing or setting the *can_be_avail* and *later* flags. The *will_be_avail* predicate is determined by the following equation:

$$will\_be\_avail = can\_be\_avail \ \wedge \ \neg later$$

4.3.6 *Finalize*

Finalize is responsible for transforming the FRG into a new form that reflects insertions and no Φ Operand is $\perp$. In addition this new form is optimized by removing any extraneous Φ Occurrences. The pass is separated into two parts: *Finalize1* and *Finalize2*.

Finalize1 is responsible for determining which Real Occurrences should be reloaded from a temporary or computed. It uses a STL map to associate redundancy class numbers to their available defining occurrence. Each Occurrence is processed in a preorder dominator tree traversal. Upon encountering a Real Occurrence the map for an available definition is accessed for its RCN. If no definition exists, or the definition does not dominate the Real Occurrence, it will become its RCN's defining occurrence and reset its reload flag. Otherwise, the Real Occurrence will set its reload flag to true and update the FRG by changing its upward edge to point to the available definition. When *Finalize1* processes a Φ Occurrence, it will only make this Φ the available definition for its RCN if it satisfies *will_ be_ avail*.

Lastly, when processing Φ Operands *Finalize1* must decide whether it is possible to insert an expression and change its representative occurrence to the Inserted Occurrence. In order to insert an expression, two conditions must hold[1]:

1. The Φ that it belongs to must satisfy *will_ be_ avail*
2. The Φ Operand must be $\perp$; or hasRealUse flag is false and its representative occurrence is a Φ that does not satisfy *will_ be_ avail*

If *insert* is satisfied, the current expression at the place the Φ Operand occurs in the CFG should be inserted. While this step seems very straight-forward, no details are provided in the paper about obtaining the correct expression to insert. Due to the fact that we implemented the worklist driven approach, it is inefficient to pass over the program to find the correct versions of the variables to formulate this inserted expression. Rather, the proper place to perform this analysis is in the Rename pass. This is a modification to the Rename algorithm not mentioned in the paper. When processing the Φ Operand Occurrences, Rename2 is aware of the current versions of variables at that point in the program. It is trivial to create the inserted instruction at that point, in the event that it is needed by *Finalize* in the future. It is a significantly more efficient to have this inserted instruction cached, versus obtaining it during the *Finalize* pass. If insert is not satisfied, the Φ Operand will update its representative occurrence to point to the available definition.

Finalize2 marks each Real Occurrence that is not reloaded as saved, and removes extraneous Φ 's to minimize the FRG. While not crucial to the success of PRE, leaving extraneous Φ 's requires more space in program representation and may impact the efficiency of future optimizations[1]. However, removing these extraneous Φ 's requires that the occurrences in its RCN refer to a different class which defines the value of the Φ Occurrence.

Finalize2 begins by setting each Φ in the FRG that satisfies *will_ be_ avail* to be extraneous. Recall that the *save* flags for Real Occurrences were initialized to false. *Finalize2* then looks at each Real Occurrence that has its *reload* flag set. If it is to be reloaded, it must update its representative occurrence by calling Set_save(). Set_save() looks at the representative occurrence, and if it is a a Real Occurrence the save flag for that Real is set to true. Otherwise, if it is a Φ Occurrence it will recursively call Set_save() in each of its Φ Operand Occurrences. Lastly, if the representative occurrence is a Real or Inserted Occurrence, it will declare each Φ in its iterated dominance frontier to be extraneous. *Finalize2* then needs to remove the extraneous Φ 's and update the FRG accordingly.

The algorithm for *Finalize2* did not work according to the paper. When removing extraneous Φ 's infinite loops were occurring. This was due to the paper leaving out the detail that once a Φ has been removed and its FRG updated, it should not be processed again.

4.3.7 *CodeMotion*

An algorithm for *CodeMotion* wasn't given explicitly in the paper. We came up with the following algorithm based on the description given in [1].

```
for f ∈ F in preorder traversal of the Dominator Tree do
   if f is a real occurrence
      if save(f)
         generate_save(f)
      else if reload(f)
         generate_reload(f)
   else if f is a Φ
      generate_ssa_phi(f)
   else if f is a Φ operand
      generate_reload(f)
   else if f is an inserted occurrence
      generate_save(f)
end
```

After the *Finalize* phase is finished, we have a set with Real Occurrences, Φ nodes, Φ operand, and Inserted Occurrences.

For Real Occurrences, if they are to be "saved," we generate a save of that expression to a temporary. In LLVM, this involves creating a `cast` of the Real Occurrence's instruction and placing it in the CFG after that instruction. This will act as this instruction's "current temporary version." If the Real Occurrence should be "reloaded," then we generate a reload of the instruction. This is done by simply taking the occurrence's defining instruction's current temporary version and replacing the instruction with that current temporary.

For Φ nodes, we notice that these are the places where two or more expressions are merged in the CFG. The expressions coming in are in registers (`Value*`s in LLVM). We create an SSA ϕ node to perform this merging.

For Φ operands, we want to reload the temporary value of its defining instruction. In our implementation, this doesn't require any modifications since we will use the Φ operand's defining instruction's current temporary instead of doing an actual insertion of code at this point.

For Inserted Occurrences, we need to generate a save of the instruction into a temporary variable. We treat this in same way we treat a Real Occurrence that is to be saved.

4.4 Limitations & Weaknesses

After much discussion, it was determined that the SSAPRE algorithm should only need to be run once on the code to gain the full benefits of PRE. However, it requires that the occurrence sets that are collected for each expression type be topographically sorted and run in order. That is, if an expression in set A uses the result of an expression in set B, then set B should be run through the algorithm before set A. Our implementation doesn't keep this topographical ordering.

Our algorithm currently does not use the value numbering interface to find expressions that produce the same value, but are not lexically equivalent. As a side effect of this, we are unable to take advantage of the `load-vn` value numbering pass, which would allow our algorithm to transparently handle partially redundant loads disambiguated by a user-selectable alias analysis implementation. We consider this to be a straight-forward extension of our current implementation, which will be easy to implement once the other deficiencies of the underlying algorithm are fixed.

Running our pass on code twice results in code which is no longer correct; unfortunately, this fact seems to be from a latent bug which would require more time to find than we had.

4.5 Status

The implementation of SSAPRE is almost complete. At the time of this report, our implementation is successfully removing partial redundancies properly from a good deal of input codes, although there are still some bugs present which we did not have time to fix. In particular, we ran into some falsifiability issues with our input codes, in the sense that it was difficult to ascertain whether or not the ϕ-operand deadness invariant had been maintained by preoptimization passes. As we see it, there are four primary actions which must occur before our implementation is robust enough to be fully integrated into LLVM as a drop-in replacement for GCSE and LICM:

- More testing to expose latent bugs and fix the existing ones.
- A solution to the ϕ-operand deadness invariant that is compile-time efficient and correct. This is primarily to relax the stringent requirements imposed by the provided SSAPRE algorithm [1]. The authors of [1] do discuss the possibility of relaxing this criteria, but do not go into detail.
- Determine how value numbering information (particularly for load instructions) can be used to increase the efficacy of PRE.
- Implement a topologically-ordered expression visitation mechanism so that our implementation SSAPRE can be more aggressive in discovering redundancies in the input code.

5 Issues with Paper

While the algorithm presented in this paper takes advantage of the sparseness of SSA and can perform comparable to LICM and GCSE, it unfortunately has a few drawbacks. The biggest drawback is the requirement that live ranges of SSA versions of the same variable can not overlap. While this is holds true immediately after SSA construction, it is not guaranteed to hold true after several optimizations have been performed on a given program. To assume that PRE is to operate in a vacuum isn't valid. Most likely it will occur near the end of a long list of optimizations. Therefore, it is our belief that further research is needed on this algorithm to avoid this requirement.

The majority of the phases of the SSAPRE algorithm were presented in a fairly straightforward manner. However, there were significant, crucial gaps left for the reader to infer and some implementation details missing from the algorithms presented in the figures but stated in a few lines in the text. In particular it seems as though the Worklist driven section was not detailed. The status of particular phases with respect to how useful the paper was is as follows:

- Collect Occurrences — No algorithm or details on how to do this.
- Rename — No details on what it really means to copy a real occurrence (in particular, how to set the def edge properly for manufactured real occurrences).
- Down Safety — No details on how to do initialization in the delayed renaming algorithm
- Finalize — We witness an infinite loop in set_replacement for the implementation given in the paper. We have fixed the problem and believe that we are correct, but there is a bit of uncertainty present. Furthermore, no information is provided regarding the contents of the inserted occurrences. In particular, the algorithm does not explicitly state how to construct the occurrence to insert at a Φoperand when insert is satisfied.

6 Experimental Results

Table 1 shows how SSAPRE performs on some of the benchmarks that work under the LLVM infrastructure. Due to some problems with the Sparc back-end, we used the `lli` command line utility to interpret LLVM bytecode and count the number of dynamic instructions. In almost all cases, we see a marked reduction in the number of dynamic instructions generated, which corresponds directly to the elimination of redundant computations.

The "Raw" config denotes application of PRE to raw, unoptimized bytecode, and "Opt" refers to application of PRE after many preoptimization passes[4]. We consider the Opt version of Olden_perimeter to be

[4]In particular, funcresolve, globaldce, deadtypeelim, constantmerge, verifier, deadinstelim, raiseallocs, indvarsimplify, raisepointerrefs, mem2reg, simplifycfg, sccp, instcombine, aggressivedce, simplifycfg

Benchmark	Config	No PRE	PRE	% Improvement
matTranspose	Raw	761157	591739	28.63
sumarray	Raw	3848	3424	12.38
DuffsDevice	Raw	3750	3554	5.51
pi	Raw	95446	78008	22.35
sumarray2d	Raw	512250	452464	13.21
sumarraymalloc	Raw	4697	4215	11.44
test_indvars	Raw	724987	583004	24.35
Olden_tsp (512)	Raw	9206950	8889874	3.57
Olden_treeadd (10)	Raw	4508523	4295345	4.96
Olden_treeadd (10)	Opt	1986671	1986671	0.0
Olden_health	Raw	215848	201590	7.07
Olden_perimeter (5)	Raw	2728844	2618785	4.20
Olden_perimeter (5)	Opt	1248856	1310623	-4.71

Table 1: Dynamic instruction reduction resulting from application of SSAPRE

an outlier [5], and presume that no redundancies existed in the Opt version Olden_treeadd.

Table 2 compares the dynamic instruction reduction induced by PRE vs. application of LLVM's GCSE and LICM implementation.

				% Improvement
Benchmark	Config	GCSE/LICM	PRE	(PRE vs LICM)
Olden_tsp (512)	Raw	7867153	8889874	-11.5
Olden_treeadd (10)	Raw	4170526	4295345	-2.91
Olden_treeadd (10)	Opt	1973272	1986671	-0.67
Olden_health	Raw	172438	201590	-14.46
Olden_perimeter	Raw	2378062	2618785	-9.19
Olden_perimeter	Opt	1189636	1310623	-9.23

Table 2: Dynamic instruction reduction in GCSE/LICM vs. SSAPRE

Unfortunately, our implementation of SSAPRE doesn't beat the LLVM implementation of GCSE and LICM. We believe this to be because we are not using value numbering information to discover more redundancies than those available to the analysis by considering only lexically identified expressions. Furthermore, we weren't able to eliminate redundant loads because value numbering information that simply yields equivalence of load instructions is insufficient to prove a load redundant and correctly eliminate the redundancy in some cases. For example, if we were to employ (load) value-numbering analysis, two subsequent loads preceded by a related store in the body of a loop nest may be VN-equivalent and both be proven redundant by our implementation, since it wouldn't explicitly look for preceding related stores that ought to prevent hoisting.

It is our belief that the a proper worklist-driven implementation, wherein the expression equivalence classes are visited in the proper order would set SSAPRE closer to the results obtained via application of GCSE and LICM. Additionally, the application of the `instcombine` pass after SSAPRE would be useful, since SSAPRE introduces a lot of casts which are able to be folded together (i.e. copy propagation). We speculate that this could be why we witness an increase in the dynamic instruction count of Olden_perimeter.

[5] We realize that PRE should never increase number of dynamic instructions; unfortunately, we did not have time to investigate this issue

7 Conclusion

This wraps up our presentation of the SSAPRE algorithm. We've completed an initial implementation of the algorithm presented in the paper. Eventhough it has some deficiencies, we've learned a lot from the process of implementation and have identified several problems with the algorithm as presented in the paper.

We've shown that PRE is very good at reducing the number of dynamic instructions executed and believe PRE will be an importat part of an SSA based optimizer when the algorithm matures.

References

[1] Kennedy, R., Chan, S., Liu, S., Lo, R., Tu, P., and Chow, F. 1999. Partial Redundancy Elimination in SSA Form. In *ACM Transactions on Programming Languages and Systems, Vol. 21, No. 3.* 627-674.

[2] Morel, E. and Renviose, C. 1979. Global optimization by suppression of partial redundancies. In *Communications of the ACM.* 96-103.

[3] Knoop, J., Rüthing, O., Steffen, B. 1992. Lazy Code Motion. In *ACM SIGPLAN '92.* 224-234.

[4] Cormen, T., Leiserson, C., Rivest, R, Stein, C. 2001. Introduction to Algorithms, 2nd edition. MIT Press / McGraw-Hill 2001.

[5] Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, F. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In *ACM Transactions on Programming Languages and Systems.* Vol 3, 4. 451-490.

APPENDIX

A Lazy Code Motion Example

We took the CFG from the Lazy Code Motion paper ([3] Figure 1) and created a program in LLVM which has the same CFG with computations in the same basic blocks. Using this, we can determine if our pass is placing computations in the optimal places predicted by [3]. Though our pass works on all expressions in the program, for the sake of brevity we will focus only on the "core" expressions — i.e., those of the form `%foo = mul int %a, %b`.

A.1 Before SSAPRE Pass

```
%.LCA = internal global [ 13 x sbyte ] c"B%d: A = %d\0A\00"    ; <[13 x sbyte*]>
%.LCB = internal global [ 13 x sbyte ] c"B%d: B = %d\0A\00"    ; <[13 x sbyte*]>
%.LCX = internal global [ 13 x sbyte ] c"B%d: X = %d\0A\00"    ; <[13 x sbyte*]>
%.LCY = internal global [ 13 x sbyte ] c"B%d: Y = %d\0A\00"    ; <[13 x sbyte*]>
%.LCZ = internal global [ 13 x sbyte ] c"B%d: Z = %d\0A\00"    ; <[13 x sbyte*]>
%.LCSUM = internal global [ 17 x sbyte ] c"B%d: Y Sum = %d\0A\00"    ; <[17 x sbyte*]>

implementation   ; Functions:

declare int %printf(sbyte*, ...)

int %main(int %argc, sbyte** %argv) {
B1: ; No predecessors!
  %B1cond = setge int %argc, 2
  %LCA = getelementptr [13 x sbyte]* %.LCA, long 0, long 0
  %LCB = getelementptr [13 x sbyte]* %.LCB, long 0, long 0
  %LCX = getelementptr [13 x sbyte]* %.LCX, long 0, long 0
  %LCY = getelementptr [13 x sbyte]* %.LCY, long 0, long 0
  %LCZ = getelementptr [13 x sbyte]* %.LCZ, long 0, long 0
  %LCSUM = getelementptr [17 x sbyte]* %.LCSUM, long 0, long 0
  br bool %B1cond, label %B2, label %B4

B2:
  %a0 = cast int 1 to int
  %b0 = cast int 3 to int
  br label %B3

B3:
  %x0 = mul int %a0, %b0
  call int (sbyte*, ...)* %printf( sbyte* %LCX, int 3, int %x0 )
  br label %B5

B4:
  %a1 = cast int 1 to int
  %b1 = cast int 27 to int
  %x1 = cast int 0 to int
  br label %B5

B5:
  ;; Expect Factor node here for expr in B3
  %a2 = phi int [ %a0, %B3 ], [ %a1, %B4 ]
  %b2 = phi int [ %b0, %B3 ], [ %b1, %B4 ]
  %x2 = phi int [ %x0, %B3 ], [ %x1, %B4 ]
  call int (sbyte*, ...)* %printf( sbyte* %LCA, int 5, int %a2 )
  call int (sbyte*, ...)* %printf( sbyte* %LCB, int 5, int %b2 )
  call int (sbyte*, ...)* %printf( sbyte* %LCX, int 5, int %x2 )
  %B5cond = seteq int %b2, 3
  br bool %B5cond, label %B6, label %B7

B6:
  %B6cond = seteq int %argc, 3
  br bool %B6cond, label %B8, label %B9

B7:
  call int (sbyte*, ...)* %printf( sbyte* %LCA, int 7, int %a2 )
  call int (sbyte*, ...)* %printf( sbyte* %LCB, int 7, int %b2 )
  call int (sbyte*, ...)* %printf( sbyte* %LCX, int 7, int %x2 )
  br label %B18

B8:
  %y0 = cast int 0 to int
  %y_sum0 = cast int 0 to int
  %count0 = cast int %x2 to int
  br label %B11

B10:
  %y1 = mul int %a2, %b2    ;; This expression is inside of a loop and
                            ;; is invariant to that loop. It should be
                            ;; moved to basic block B8.
  %y_sum1 = add int %y_sum2, %y1
  %count1 = sub int %count2, 1
  br label %B11

B11:
  ;; Expect Factor node here for the expr in B10
  %y2 = phi int [ %y0, %B8 ], [ %y1, %B10 ]
  %y_sum2 = phi int [ %y_sum0, %B8 ], [ %y_sum1, %B10 ]
  %count2 = phi int [ %count0, %B8 ], [ %count1, %B10 ]
  %B11cond = setge int %count2, 0
  br bool %B11cond, label %B10, label %B14

B14:
  call int (sbyte*, ...)* %printf( sbyte* %LCSUM, int 14, int %y_sum2 )
  br label %B16

B9:
  br label %B12

B12:
  %B12cond = seteq int %argc, 4
  br bool %B12cond, label %B15, label %B17

B15:
  %y3 = mul int %a2, %b2    ;; This expression won't be moved out of B15
```

```
                            ;; because this is the earliest position for
                            ;; it with respect to those exprs and their
                            ;; uses in B15 and B16.
  call int (sbyte*, ...)* %printf( sbyte* %LCY, int 15, int %y3 )
  br label %B16

B16:
  ;; Expect Factor node here for exprs in B10 and B15
  %y4 = phi int [ %y3, %B15 ], [ %y2, %B14 ]
  %z0 = mul int %a2, %b2    ;; This expression will be converted into an
                            ;; assignment because there are evaluations
                            ;; of this expression coming in from B8 and
                            ;; B15 after the pass is run.
  call int (sbyte*, ...)* %printf( sbyte* %LCZ, int 16, int %z0 )
  br label %B18

B17:
  %x3 = mul int %a2, %b2    ;; This expression will remain here because
                            ;; there's no earlier placement for this
                            ;; computation that is optimal.
  call int (sbyte*, ...)* %printf( sbyte* %LCX, int 17, int %x3 )
  br label %B18

B18:
  ;; Expect Factor node here for exprs in B10, B15, B16, B17
  ret int 0
}
```

A.2 Expected Results

The Lazy Code Motion paper [3] concludes that the above program should have a computation of `mul int %a, %b` in basic blocks B3, B8, B15, and B17 with uses of those computations in basic blocks B3, B10, B15, B16, and B17 ([3] Figure 7). As shown in the next section, the result of running the SSAPRE pass on the above code does just this.

A.3 After SSAPRE Pass

```
%.LCA = internal global [13 x sbyte] c"B%d: A = %d\0A\00" ; <[13 x sbyte]*> [#uses=1]
%.LCB = internal global [13 x sbyte] c"B%d: B = %d\0A\00" ; <[13 x sbyte]*> [#uses=1]
%.LCX = internal global [13 x sbyte] c"B%d: X = %d\0A\00" ; <[13 x sbyte]*> [#uses=1]
%.LCY = internal global [13 x sbyte] c"B%d: Y = %d\0A\00" ; <[13 x sbyte]*> [#uses=1]
%.LCZ = internal global [13 x sbyte] c"B%d: Z = %d\0A\00" ; <[13 x sbyte]*> [#uses=1]
%.LCSUM = internal global [17 x sbyte] c"B%d: Y Sum = %d\0A\00" ; <[17 x sbyte]*> [#uses=1]

implementation   ; Functions:

declare int %printf(sbyte*, ...)

int %main(int %argc, sbyte** %argv) {
B1:    ; No predecessors!
  %B1cond = setge int %argc, 2    ; <bool> [#uses=1]
  %LCA = getelementptr [13 x sbyte]* %.LCA, long 0, long 0    ; <sbyte*> [#uses=2]
  %LCB = getelementptr [13 x sbyte]* %.LCB, long 0, long 0    ; <sbyte*> [#uses=2]
  %LCX = getelementptr [13 x sbyte]* %.LCX, long 0, long 0    ; <sbyte*> [#uses=4]
  %LCY = getelementptr [13 x sbyte]* %.LCY, long 0, long 0    ; <sbyte*> [#uses=1]
  %LCZ = getelementptr [13 x sbyte]* %.LCZ, long 0, long 0    ; <sbyte*> [#uses=1]
  %LCSUM = getelementptr [17 x sbyte]* %.LCSUM, long 0, long 0    ; <sbyte*> [#uses=1]
  br bool %B1cond, label %B2, label %B4

B2:    ; preds = %B1
  %a0 = cast int 1 to int    ; <int> [#uses=2]
  %b0 = cast int 3 to int    ; <int> [#uses=2]
  br label %B3

B3:    ; preds = %B2
  %x0 = mul int %a0, %b0    ; <int> [#uses=2]
  call int (sbyte*, ...)* %printf( sbyte* %LCX, int 3, int %x0 )    ; <int>:0 [#uses=0]
  br label %B5

B4:    ; preds = %B1
  %a1 = cast int 1 to int    ; <int> [#uses=1]
  %b1 = cast int 27 to int   ; <int> [#uses=1]
  %x1 = cast int 0 to int    ; <int> [#uses=1]
  br label %B5

B5:    ; preds = %B4, %B3
  %a2 = phi int [ %a0, %B3 ], [ %a1, %B4 ]    ; <int> [#uses=5]
  %b2 = phi int [ %b0, %B3 ], [ %b1, %B4 ]    ; <int> [#uses=6]
  %x2 = phi int [ %x0, %B3 ], [ %x1, %B4 ]    ; <int> [#uses=3]
  call int (sbyte*, ...)* %printf( sbyte* %LCA, int 5, int %a2 )    ; <int>:1 [#uses=0]
  call int (sbyte*, ...)* %printf( sbyte* %LCB, int 5, int %b2 )    ; <int>:2 [#uses=0]
  call int (sbyte*, ...)* %printf( sbyte* %LCX, int 5, int %x2 )    ; <int>:3 [#uses=0]
  %B5cond = seteq int %b2, 3    ; <bool> [#uses=1]
  br bool %B5cond, label %B6, label %B7

B6:    ; preds = %B5
  %B6cond = seteq int %argc, 3    ; <bool> [#uses=1]
  br bool %B6cond, label %B8, label %B9

B7:    ; preds = %B5
  call int (sbyte*, ...)* %printf( sbyte* %LCA, int 7, int %a2 )    ; <int>:4 [#uses=0]
  call int (sbyte*, ...)* %printf( sbyte* %LCB, int 7, int %b2 )    ; <int>:5 [#uses=0]
  call int (sbyte*, ...)* %printf( sbyte* %LCX, int 7, int %x2 )    ; <int>:6 [#uses=0]
  br label %B18

B8:    ; preds = %B6
  %y0 = cast int 0 to int           ; <int> [#uses=2]
  %T_3 = cast int %y0 to int        ; <int> [#uses=1]
  %count0 = cast int %x2 to int     ; <int> [#uses=1]
  %y1_clone = mul int %a2, %b2      ; <int> [#uses=1]
  %T_0 = cast int %y1_clone to int  ; <int> [#uses=3]
  br label %B11

B10:    ; preds = %B11
  %y_sum1 = add int %y_sum2, %T_0   ; <int> [#uses=1]
  %count1 = sub int %count2, 1      ; <int> [#uses=1]
  br label %B11
```

```
B11:    ; preds = %B10, %B8
  %y2 = phi int [ %y0, %B8 ], [ %T_0, %B10 ]            ; <int> [#uses=1]
  %y_sum2 = phi int [ %T_3, %B8 ], [ %y_sum1, %B10 ]    ; <int> [#uses=2]
  %count2 = phi int [ %count0, %B8 ], [ %count1, %B10 ] ; <int> [#uses=2]
  %B11cond = setge int %count2, 0    ; <bool> [#uses=1]
  br bool %B11cond, label %B10, label %B14

B14:    ; preds = %B11
  call int (sbyte*, ...)* %printf( sbyte* %LCSUM, int 14, int %y_sum2 )    ; <int>:7 [#uses=0]
  br label %B16

B9:    ; preds = %B6
  br label %B12

B12:    ; preds = %B9
  %B12cond = seteq int %argc, 4    ; <bool> [#uses=1]
  br bool %B12cond, label %B15, label %B17

B15:    ; preds = %B12
  %y3 = mul int %a2, %b2        ; <int> [#uses=3]
  %T_2 = cast int %y3 to int    ; <int> [#uses=1]
  call int (sbyte*, ...)* %printf( sbyte* %LCY, int 15, int %y3 )    ; <int>:8 [#uses=0]
  br label %B16

B16:    ; preds = %B15, %B14
  %y4 = phi int [ %y3, %B15 ], [ %y2, %B14 ]    ; <int> [#uses=0]
  %T_1 = phi int [ %T_2, %B15 ], [ %T_0, %B14 ] ; <int> [#uses=1]
  call int (sbyte*, ...)* %printf( sbyte* %LCZ, int 16, int %T_1 )    ; <int>:9 [#uses=0]
  br label %B18

B17:    ; preds = %B12
  %x3 = mul int %a2, %b2    ; <int> [#uses=1]
  call int (sbyte*, ...)* %printf( sbyte* %LCX, int 17, int %x3 )    ; <int>:10 [#uses=0]
  br label %B18

B18:    ; preds = %B17, %B16, %B7
  ret int 0
}
```

B Multiply Nested Loops

In one pass of the algorithm, SSAPRE can "hoist" code which is loop invariant out of the innermost loop of a loop nest to its proper place. Our example is a program that has a triply nested loop which has two loop invariant instructions in it.

B.1 Before SSAPRE Pass

```
%.LCASUM = internal global [ 17 x sbyte ] c"B%d: A Sum = %d\0A\00"    ; <[17 x sbyte*]>
%.LCBSUM = internal global [ 17 x sbyte ] c"B%d: B Sum = %d\0A\00"    ; <[17 x sbyte*]>
%.LCCSUM = internal global [ 17 x sbyte ] c"B%d: C Sum = %d\0A\00"    ; <[17 x sbyte*]>

implementation   ; Functions:

declare int %printf(sbyte*, ...)

int %main(int %argc, sbyte** %argv) {
B1: ; No predecessors!
  %x = cast int 27 to float
  %y = cast int 3 to float
  %r = cast int 927 to float
  %i0 = cast int 10 to int
  %asum0 = cast int 0 to int
  %bsum0 = cast int 0 to int
  %csum0 = cast int 0 to int
  %LCASUM = getelementptr [17 x sbyte]* %.LCASUM, long 0, long 0
  %LCBSUM = getelementptr [17 x sbyte]* %.LCBSUM, long 0, long 0
  %LCCSUM = getelementptr [17 x sbyte]* %.LCCSUM, long 0, long 0
  br label %B2

B2:
  %i2 = phi int [ %i0, %B1 ], [ %i1, %B2_end ]
  %asum2 = phi int [ %asum0, %B1 ], [ %asum1, %B2_end ]
  %bsum2 = phi int [ %bsum0, %B1 ], [ %bsum1, %B2_end ]
  %csum2 = phi int [ %csum0, %B1 ], [ %csum1, %B2_end ]
  %j0 = cast int 10 to int
  %a = mul int %i2, 10
  %asum1 = add int %asum2, %a
  br label %B3

B3:
  %j2 = phi int [ %j0, %B2 ], [ %j1, %B3_end ]
  %bsum3 = phi int [ %bsum2, %B2 ], [ %bsum1, %B3_end ]
  %csum3 = phi int [ %csum2, %B2 ], [ %csum1, %B3_end ]
  %k0 = cast int 10 to int
  %b = mul int %j2, 10
  %bsum1 = add int %bsum3, %b
  br label %B4

B4:
  %k2 = phi int [ %k0, %B3 ], [ %k1, %B4 ]
  %csum4 = phi int [ %csum3, %B3 ], [ %csum1, %B4 ]
  %z = div float %x, %y     ;; Loop invariant computation: This should be
                            ;; moved to before the outer-most loop.
```

```
                              ;;
  %q = mul float %r, %z       ;; Loop invariant computation: This should
                              ;; also be moved to before the outer-most
                              ;; loop, but this doesn't occur because we
                              ;; do not keep a topological sort of the
                              ;; occurrence sets.
  %c = mul int %k2, 10
  %csum1 = add int %csum4, %c
  %k1 = sub int %k2, 1
  %b4cond = setgt int %k1, 0
  br bool %b4cond, label %B4, label %B3_end

B3_end:
  %j1 = sub int %j2, 1
  %b3cond = setgt int %j1, 0
  br bool %b3cond, label %B3, label %B2_end

B2_end:
  %i1 = sub int %i2, 1
  %b2cond = setgt int %i1, 0
  br bool %b2cond, label %B2, label %B_exit

B_exit:
  %q2 = mul float %q, %q
  call int (sbyte*, ...)* %printf( sbyte* %LCASUM, int 8, int %asum1 )
  call int (sbyte*, ...)* %printf( sbyte* %LCBSUM, int 8, int %bsum1 )
  call int (sbyte*, ...)* %printf( sbyte* %LCCSUM, int 8, int %csum1 )
  ret int 0
}
```

B.2 Expected Results

We expect both loop invariant instructions `%z = div float %x, %y` and `%q = mul float %r, %z` to be placed in basic block B1. However, as mentioned in the paper, this requires an ordering on the occurrence sets which our implementation doesn't enforce. So, as shown in the next section, only one instruction is moved to basic block B1.

B.3 After SSAPRE Pass

```
%.LCASUM = internal global [17 x sbyte] c"B%d: A Sum = %d\0A\00"    ; <[17 x sbyte]*> [#uses=1]
%.LCBSUM = internal global [17 x sbyte] c"B%d: B Sum = %d\0A\00"    ; <[17 x sbyte]*> [#uses=1]
%.LCCSUM = internal global [17 x sbyte] c"B%d: C Sum = %d\0A\00"    ; <[17 x sbyte]*> [#uses=1]

implementation   ; Functions:

declare int %printf(sbyte*, ...)

int %main(int %argc, sbyte** %argv) {
B1:    ; No predecessors!
  %x = cast int 27 to float         ; <float> [#uses=1]
  %y = cast int 3 to float          ; <float> [#uses=1]
  %r = cast int 927 to float        ; <float> [#uses=1]
  %i0 = cast int 10 to int          ; <int> [#uses=2]
  %T_1 = cast int %i0 to int        ; <int> [#uses=2]
  %asum0 = cast int 0 to int        ; <int> [#uses=2]
  %T_2 = cast int %asum0 to int     ; <int> [#uses=2]
  %LCASUM = getelementptr [17 x sbyte]* %.LCASUM, long 0, long 0    ; <sbyte*> [#uses=1]
  %LCBSUM = getelementptr [17 x sbyte]* %.LCBSUM, long 0, long 0    ; <sbyte*> [#uses=1]
  %LCCSUM = getelementptr [17 x sbyte]* %.LCCSUM, long 0, long 0    ; <sbyte*> [#uses=1]
  %z_clone_clone_clone = div float %x, %y           ; <float> [#uses=1]
  %T_0 = cast float %z_clone_clone_clone to float   ; <float> [#uses=1]
  br label %B2

B2:    ; preds = %B2_end.B2_crit_edge, %B1
  %i2 = phi int [ %i0, %B1 ], [ %i1, %B2_end.B2_crit_edge ]             ; <int> [#uses=2]
  %asum2 = phi int [ %asum0, %B1 ], [ %asum1, %B2_end.B2_crit_edge ]    ; <int> [#uses=1]
  %bsum2 = phi int [ %T_2, %B1 ], [ %bsum1, %B2_end.B2_crit_edge ]      ; <int> [#uses=1]
  %csum2 = phi int [ %T_2, %B1 ], [ %csum1, %B2_end.B2_crit_edge ]      ; <int> [#uses=1]
  %a = mul int %i2, 10    ; <int> [#uses=1]
  %asum1 = add int %asum2, %a    ; <int> [#uses=2]
  br label %B3

B3:    ; preds = %B3_end.B3_crit_edge, %B2
  %j2 = phi int [ %T_1, %B2 ], [ %j1, %B3_end.B3_crit_edge ]            ; <int> [#uses=2]
  %bsum3 = phi int [ %bsum2, %B2 ], [ %bsum1, %B3_end.B3_crit_edge ]    ; <int> [#uses=1]
  %csum3 = phi int [ %csum2, %B2 ], [ %csum1, %B3_end.B3_crit_edge ]    ; <int> [#uses=1]
  %b = mul int %j2, 10          ; <int> [#uses=1]
  %bsum1 = add int %bsum3, %b   ; <int> [#uses=3]
  br label %B4

B4:    ; preds = %B4.B4_crit_edge, %B3
  %k2 = phi int [ %T_1, %B3 ], [ %k1, %B4.B4_crit_edge ]            ; <int> [#uses=2]
  %csum4 = phi int [ %csum3, %B3 ], [ %csum1, %B4.B4_crit_edge ]    ; <int> [#uses=1]
  %q = mul float %r, %T_0       ; <float> [#uses=2]
  %c = mul int %k2, 10          ; <int> [#uses=1]
  %csum1 = add int %csum4, %c   ; <int> [#uses=4]
  %k1 = sub int %k2, 1          ; <int> [#uses=2]
  %b4cond = setgt int %k1, 0    ; <bool> [#uses=1]
  br bool %b4cond, label %B4.B4_crit_edge, label %B3_end

B4.B4_crit_edge:    ; preds = %B4
  br label %B4

B3_end:    ; preds = %B4
  %j1 = sub int %j2, 1          ; <int> [#uses=2]
  %b3cond = setgt int %j1, 0    ; <bool> [#uses=1]
  br bool %b3cond, label %B3_end.B3_crit_edge, label %B2_end

B3_end.B3_crit_edge:    ; preds = %B3_end
  br label %B3

B2_end:    ; preds = %B3_end
  %i1 = sub int %i2, 1          ; <int> [#uses=2]
  %b2cond = setgt int %i1, 0    ; <bool> [#uses=1]
  br bool %b2cond, label %B2_end.B2_crit_edge, label %B_exit

B2_end.B2_crit_edge:    ; preds = %B2_end
```

```
  br label %B2

B_exit:    ; preds = %B2_end
  %q2 = mul float %q, %q    ; <float> [#uses=0]
  call int (sbyte*, ...)* %printf( sbyte* %LCASUM, int 8, int %asum1 )    ; <int>:0 [#uses=0]
  call int (sbyte*, ...)* %printf( sbyte* %LCBSUM, int 8, int %bsum1 )    ; <int>:1 [#uses=0]
  call int (sbyte*, ...)* %printf( sbyte* %LCCSUM, int 8, int %csum1 )    ; <int>:2 [#uses=0]
  ret int 0
}
```

C The Role of *Later*

In the paper, they discuss the role of the *later* predicate on a Φ node. In essence, a Φ node can satisfy *down_safe* and *can_be_avail* but if it also satisfies *later*, then we won't use that Φ to insert expressions. Doing so would not eliminate any redundancies and would unnecessarily extend the live range of the temporary variable.[1] The example given here models the CFG given in the paper ([1] Fig. 9).

C.1 Before SSAPRE Pass

```
implementation   ; Functions:

int %main(int %argc, sbyte** %argv) {
BBegin:
  %a1 = cast int 37 to int
  %b1 = cast int 27 to int
  %cond = setle int %argc, 2
  br bool %cond, label %B1, label %B0

B0:
  %bb0cond = setle int %argc, 3
  br bool %bb0cond, label %B3, label %B2

B1:
  %x0 = add int %a1, %b1
  br label %B3

B2:
  ;; If later were false for the PHI node in B5, we would expect an
  ;; inserted computation of a + b here.
  %a2 = cast int 927 to int
  br label %B5

B3:
  ;; Expect PHI node here for the expr in B1
  %bb3cond = seteq int %argc, 2
  br bool %bb3cond, label %BExit, label %B4

B4:
  ;; If later were false for the PHI node in B5, we would expect an
  ;; inserted computation of a + b here.
  br label %B5

B5:
  ;; Expect PHI node here for the expr in B1
  %a3 = phi int [ %a1, %B4 ], [ %a2, %B2 ], [ %a3, %B5 ]
  %bb4cond = seteq int %argc, 0
  br bool %bb4cond, label %B6, label %B5

B6:
  %x1 = add int %a3, %b1
  br label %BExit

BExit:
  ret int 0
}
```

C.2 Expected Results

We expect none of the the `add int %a, %b` expressions to be moved since the Φ in `B3` isn't *down_safe* and the Φ in `B5` satisfies *later*. As shown in the next section, that is what occurs.

C.3 After SSAPRE Pass

```
implementation   ; Functions:

int %main(int %argc, sbyte** %argv) {
BBegin:    ; No predecessors!
  %a1 = cast int 37 to int      ; <int> [#uses=2]
  %b1 = cast int 27 to int      ; <int> [#uses=2]
  %cond = setle int %argc, 2    ; <bool> [#uses=1]
  br bool %cond, label %B1, label %B0

B0:    ; preds = %BBegin
  %bb0cond = setle int %argc, 3 ; <bool> [#uses=1]
  br bool %bb0cond, label %B0.B3_crit_edge, label %B2

B0.B3_crit_edge:    ; preds = %B0
  br label %B3

B1:    ; preds = %BBegin
  %x0 = add int %a1, %b1    ; <int> [#uses=0]
  br label %B3

B2:    ; preds = %B0
  %a2 = cast int 927 to int                 ; <int> [#uses=1]
  %bb4cond_clone1 = seteq int %argc, 0      ; <bool> [#uses=1]
  %T_2 = cast bool %bb4cond_clone1 to bool  ; <bool> [#uses=1]
  br label %B5

B3:    ; preds = %B1, %B0.B3_crit_edge
  %bb3cond = seteq int %argc, 2    ; <bool> [#uses=1]
  br bool %bb3cond, label %B3.BExit_crit_edge, label %B4

B3.BExit_crit_edge:    ; preds = %B3
  br label %BExit

B4:    ; preds = %B3
  %bb4cond_clone = seteq int %argc, 0       ; <bool> [#uses=1]
  %T_0 = cast bool %bb4cond_clone to bool   ; <bool> [#uses=1]
  br label %B5

B5:    ; preds = %B5.B5_crit_edge, %B4, %B2
  %a3 = phi int [ %a1, %B4 ], [ %a2, %B2 ], [ %a3, %B5.B5_crit_edge ]       ; <int> [#uses=2]
  %T_1 = phi bool [ %T_0, %B4 ], [ %T_2, %B2 ], [ %T_1, %B5.B5_crit_edge ]  ; <bool> [#uses=2]
  br bool %T_1, label %B6, label %B5.B5_crit_edge

B5.B5_crit_edge:    ; preds = %B5
  br label %B5
B6:    ; preds = %B5
  %x1 = add int %a3, %b1    ; <int> [#uses=0]
  br label %BExit

BExit:    ; preds = %B6, %B3.BExit_crit_edge
  ret int 0
}
```