HAL
open science

Improvements to the Psi-SSA representation

Francois de Ferriere

To cite this version:

Francois de Ferriere. Improvements to the Psi-SSA representation. 2007. hal-00146363

HAL Id: hal-00146363

https://hal.science/hal-00146363v1

Submitted on 15 May 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Improvements to the Psi-SSA Representation

François de Ferrière
ST Microelectronics
12 rue Jules Horowitz, 38000 Grenoble, France
francois.de-ferriere@st.com

hal-00146363, version 1 - 15 May 2007

Abstract

Modern compiler implementations use the Static Single Assignment representation [5] as a way to efficiently implement optimizing algorithms. However this representation is not well adapted to architectures with a predicated instruction set. The ψ-SSA representation was first introduced in [11] as an extension to the Static Single Assignment representation. The ψ-SSA representation extends the SSA representation such that standard SSA algorithms can be easily adapted to an architecture with a fully predicated instruction set. A new pseudo operation, the ψ operation, is introduced to merge several conditional definitions into a unique definition.

This paper presents an adaptation of the ψ-SSA representation to support architectures with a partially predicated instruction set. The definition of the ψ operation is extended to support the generation and the optimization of partially predicated code. In particular, a predicate promotion transformation is introduced to reduce the number of predicated operations, as well as the number of operations used to compute guard registers. An out of ψ-SSA algorithm is also described, which fixes and improves the algorithm described in [11]. This algorithm is derived from the out of SSA algorithm from Sreedhar et al. [10], where the definitions of liveness and interferences have been extended for the ψ operations. This algorithm inserts predicated copy operations to restore the correct semantics in the program in a non-SSA form.

The ψ-SSA representation is used in our production compilers, based on the Open64 technology, for the ST200 family processors. In this compiler, predicated code is generated by an if-conversion algorithm performed under the ψ-SSA representation [12, 1].

1. Introduction

The Static Single Assignment representation was introduced in [5] and is now widely used in modern compilers. The SSA representation has proven to be a very efficient internal compiler representation for performing various optimizations on scalar variables. In this representation, each definition of a scalar variable is renamed into a unique name, and variable uses are renamed to refer to these new definition names or to special ϕ instructions that are introduced to merge values coming from different control-flow paths. Most of the standard optimization algorithms have been adapted to this representation, such as constant propagation [13], dead-code elimination [9], induction variables optimization [14], and partial redundancy elimination [2]. These algorithms usually perform equally well or even better than their original versions on a non-SSA representation. However, these algorithms are more difficult to adapt in presence of aliased variables, partial definitions or conditional definitions. To overcome these difficulties, some extensions to the SSA representation have already been proposed, such as the HSSA representation [3] for aliases with pointers, the Array SSA form [8] for array variables and the ψ-SSA representation [11] to handle conditional definitions.

In this document we present an extension of the ψ-SSA representation for partially predicated architectures. The first section will present theoretical and practical aspects of the ψ-SSA representation. The second section will then describe the adaptation of the ψ-SSA representation to the context of partial predication. The third section will present an out of SSA algorithm for the ψ-SSA representation, for both full and partial predication. This algorithm improves the algorithm described in the original ψ-SSA paper and also fixes some errors. In the fourth section we will present some results we have on our production compiler for one of the ST200 family processors.

2. The Psi-SSA representation

The ψ-SSA representation was developed to extend the SSA representation with support for predicated operations. In the SSA representation, each definition of a variable is given a unique name, and new pseudo definitions are intro-

```
if(p)
   a = op1;          p?    a = op1;
else
   b = op2;          p̄?    b = op2;
x = Phi(a, b)             x = Psi(a, b)
```

Figure 1. ψ-SSA representation

```
if(p)
   a = 1;            p?    a = 1;
else
   b = −1;           p̄?    b = −1;
x = Phi(a, b)             x = Psi(a, b)
if(q)
   c = 0;            q?    c = 0;
y = Phi(x, c)             y = Psi(a, b, c)
```

Figure 2. ψ-SSA with non-disjoint predicates

duced on ϕ instructions to merge values coming from different control-flow paths. In this representation, each definition is an unconditional definition, and the value of a variable is the value of the expression on the unique assignment to this variable. This essential property of the SSA representation does not any longer hold when definitions may be conditionally executed. When the definition for a variable is a predicated operation, this operation is executed depending on the value of a guard register. As a result, the value of the variable after the predicated operation is either the value of the expression on the assignment if the predicate is true, or the value the variable had before this operation if the predicate is false. We need a way to express these conditional definitions whilst keeping the static single assignment property.

Predicated operations can be used to replace code that contains control-flow edges by straight line code containing predicated operations. Such a transformation is performed by an if-conversion optimization [6, 1]. A simple example of if-conversion is given in figure 1. In the rest of this paper, we use the notation `p? <exp>` to say that `<exp>` is executed only if the predicate `p` is TRUE.

In the ψ-SSA representation, ψ operations are added to the SSA representation. ψ operations are for predicated definitions what ϕ operations are for definitions on different control-flow edges. A ψ operation merges values that are defined under different predicates, and defines a single variable to represent these different values.

In the SSA representation, ϕ operations are placed at control-flow merge points where each argument flows from a different incoming edge. In the ψ-SSA representation, on a ψ operation, all the incoming edges of a ϕ operation are merged into a single execution path, and each argument is now defined on a different predicate.

In figure 1, variables `a` and `b` were initially the same variable. On the left-hand side, the SSA construction renamed the two definitions of this unique variable into two different names, and introduced a new variable `x` defined by a ϕ operation to merge the two values coming from the different control-flow paths. On the right-hand side, an if-conversion algorithm transformed this code to remove the control-flow edges. It introduced predicated operations for the definitions of the variables `a` and `b` and turned the ϕ operation into a ψ operation. Each argument of the ψ operation is defined by a predicated operation. The intersection of the domain of the two predicates is empty and the value of the ψ operation is given by one or the other of its arguments, depending on the value of the predicate.

The ψ operations can also represent cases where variables are defined on predicates that are computed from independent conditions. This is illustrated in figure 2, where the predicates `p` and `q` are independent. During the SSA construction a unique variable was renamed into the variables `a`, `b` and `c` and the variables `x` and `y` were introduced to merge values coming from different control-flow paths. In the non-predicated code, there is a control-dependency between `x` and `c`, which means the definition of `c` must be executed after the value for `x` has been computed. In the predicated form of this example, there are no longer any control dependencies between the definitions of `a`, `b` and `c`. A compiler transformation can now freely move these definitions independently of each other, which may allow more optimizations to be performed on this code. However, the semantics of the original code requires that the definition of `c` occurs after the definitions of `a` and `b`. We use the order of the arguments in a ψ operation to keep the information on the original order of the definitions. We take the convention that the order of the arguments in a ψ operation is, from left to right, equal to the order of their definitions, from top to bottom, in the control-flow dominance tree of the program in a non-SSA representation. This information is needed to maintain the correct semantics of the code during transformations of the ψ-SSA representation and when reverting the code back to a non ψ-SSA representation.

With this definition of the ψ-SSA representation, conditional definitions on predicated code are now replaced by unconditional definitions on ψ operations. Usual algorithms that perform optimizations or transformations on the SSA representation can now be easily adapted to the ψ-SSA representation, without compromising the efficiency of the transformations performed. Actually, within the ψ-SSA representation, predicated definitions behave exactly the same as non predicated ones for optimizations on the SSA representation. Only the ψ operations have to be treated in a specific way. As an example, the constant propagation algorithm described in [13] can be easily adapted to the ψ-SSA representation. In this algorithm, the only modification is that ψ operations have to be handled with

the same rules as the ϕ operations. We have also ported dead code elimination [9] and global value numbering [4] algorithms to this representation, and we expect that partial redundancy elimination [2], and induction variable analysis [14] should be easy to adapt.

In addition to standard algorithms that can now be easily adapted to ψ operations and predicated code, a number of additional transformations can be performed on the ψ operations. These transformation are ψ-inlining, ψ-reduction and ψ-projection, they are described in detail in [11]. ψ-inlining will recursively replace in a ψ operation an argument that is defined by another ψ operation by the arguments of this second ψ operation. ψ-reduction will remove from a ψ operation an argument whose value will always be overridden by arguments on its right in the argument list, because the domain of the predicate associated with this argument is included in the union of the domains of the predicates associated with the arguments on its right. ψ-projection will create from a ψ operation new ψ operations for uses in operations guarded by different predicates. Each new ψ operation is created as the projection on a given predicate of the original ψ operation . In this new ψ operation, arguments whose associated predicate has a domain that is disjoint with the domain of the predicate on which the projection is performed actually contribute no value to the ψ operation and are then removed.

3. Psi-SSA and partial predication

In the original paper on ψ-SSA we only considered the use of ψ-SSA for a fully predicated processor. We describe here how this representation has been modified to be used for a processor with a partially predicated instruction set.

In a partially predicated instruction set, only a subset of the instruction set of the targeted processor supports a predicate operand. For example, the instruction set may support only a conditional `move` instruction. It can also include more specific instructions such as a `select` instruction. A `select` instruction takes two arguments and a guard register, and assigns the value of one or the other of its arguments into a variable, depending on the value of the guard register.

The only impact of partial predication on the ψ-SSA representation is that when a ψ operation is created as a replacement for a ϕ operation, during if-conversion for example, some of its arguments may be defined by operations that cannot be predicated. A preliminary condition is that the ψ operation can be created only if these non-predicated arguments can be safely speculated, which means executed under some conditions where they would not have been executed otherwise. Although these definitions are speculated, their values were only meaningful under a given predicate in the original code. The information on this predicate must be kept in some way.

Figure 3 shows an example where some code with control-flow edges was transformed into a linear sequence of instructions. In this example, the `ADD` operation cannot be predicated.

In figure 3 b), we have introduced predicated move operations, so that ψ operations still have the definitions of their arguments being predicated, while allowing an if-conversion transformation to be performed even on operations that cannot be predicated. In the case where conditional move operations are not available on the target processor, when leaving the ψ-SSA representation, these operations, along with the ψ operation, will be replaced by other operations available on the target processor, such as a `select` instruction for example. The main disadvantage of this solution is that the semantics of the initial ϕ operation is now expressed by three operations. These operations will have to be treated all together during transformations on the ψ operations, and in particular when reverting the code back to non ψ-SSA representation.

In figure 3 c), we chose to express these `conditional move` operations directly in the ψ operation, by means of a predicate associated with each argument of the ψ operation. With this representation, the information represented in the ϕ operation by the control-flow edges is now present in the ψ operation by means of predicates.

In the general case, the definition of a variable can be predicated. Using the representation in figure 3 c), there can be one predicate associated with the definition of a variable, and there will be one predicate associated with the use of the variable in a ψ operation. The two domains for these two predicates do not need to be equal, only the domain of the predicate on the definition has to contain the domain of the predicate on the ψ argument. This extension to the representation of the ψ operations allows one to perform a copy folding algorithm to remove all `mov` operations in the representation, whether they are predicated or not.

3.1. Psi-predicate promotion

The extension of the ψ-SSA representation to the context of partial predication brings another useful transformation to the ψ operations, the ψ-predicate promotion.

The predicate associated with an argument in a ψ operation can be promoted, without changing the semantics of the ψ operation. By predicate promotion, we mean that a predicate can be replaced by a predicate with a larger predicate domain. This promotion must obey the two following conditions so that the semantics of the ψ operation after the transformation is valid and unchanged.

- **Condition 1** For an argument in a ψ operation, the domain of the predicate used on the definition of this argument must contain the domain of the new predicate associated with this argument.

```
if(p)
   a = ADD i, 1;
else
   b = ADD i, 2;

x = Phi(a, b)
```

a) before if − conversion

```
      a = ADD i, 1;
p?    c = a
      b = ADD i, 2;
p̄?    d = b
      x = Psi(c, d)
```

b) conditional moves

```
a = ADD i, 1;

b = ADD i, 2;

x = Psi(p?a, p̄?b)
```

c) extended Psi operation

Figure 3. Psi-SSA for partial predication

for the instructions

$\mathtt{p?}\quad \mathtt{x} = ...$

$\quad\quad \mathtt{y} = \mathtt{Psi}(..., \mathtt{q?x}, ...)$

then

$\quad\quad \mathtt{q} \subseteq \mathtt{p}$

- **Condition 2** For an argument in a ψ operation, the domain of the new predicate associated with it can be extended up to include the domains of the predicates associated with arguments in the ψ operation that were defined after the definition for this argument in the original program.

  for an instruction

  $\quad\quad \mathtt{y} = \mathtt{Psi}(\mathtt{p_1?x_1}, \mathtt{p_2?x_2}, ..., \mathtt{p_i?x_i}, ..., \mathtt{p_n?x_n})$

  transformed to

  $\quad\quad \mathtt{y} = \mathtt{Psi}(\mathtt{p_1?x_1}, \mathtt{p_2?x_2}, ..., \mathtt{p'_i?x_i}, ..., \mathtt{p_n?x_n})$

  then

  $\quad\quad \mathtt{p'_i} \subseteq \bigcup_{\mathtt{k=i}}^{\mathtt{n}} \mathtt{p_k}$

The first condition ensures that the ψ operation is still valid. This condition means that, in addition to predicate promotion, speculation may have to be performed first on the definition of the argument of the ψ operation. The second condition ensures that the value of the ψ operation is not changed. We already said that the order of the arguments in a ψ operation is, from left to right, the order, from top to bottom, of the definitions in the control-flow dominance tree of the original program. Thus, the domain of the predicate associated with an argument in a ψ operation can be extended up to include the domains of each of the predicates associated with arguments at its right in the ψ argument list. With this condition, we ensure that the conditions under which arguments at the left of the promoted argument can have there value overridden by arguments at their right in the ψ operation remain unchanged. This condition also means that the first argument of a ψ operation can be promoted independently of the other arguments in the ψ operation, provided that the first condition is still satisfied.

This ψ-predicate promotion transformation allows us to reduce the number of predicates that need to be computed, and to reduce the dependencies between predicate computations and conditional operations. In fact, the first argument of a ψ operation can usually be promoted under the TRUE predicate, provided that speculation can be applied. Also, when disjoint conditions are computed, one of them can be promoted to include the other conditions, usually reducing the dependency height of the predicated expressions. The ψ-predicate promotion transformation can be applied during an if-conversion algorithm for example. A side effect of this transformation is that it may increase the number of copy instructions to be generated during the out of ψ-SSA phase, because of more live-range interference between arguments in a ψ operation, as will be explained in the next section.

4. An out of Psi-SSA algorithm

We have now described the semantics of the ψ operation along with the transformations that can be applied on it. Then, after optimizations have been applied on a ψ-SSA representation, the code must eventually be reverted back to a standard, non SSA, form. On the SSA representation this is called the out of SSA phase. This pass must be adapted to the ψ-SSA representation.

In the original paper on the ψ-SSA representation [11], an out of ψ-SSA algorithm was described. In this section, we present a complete algorithm that extends the original algorithm to our new representation, and also fixes one error in the original description.

4.1. Conventional SSA

The algorithm described in the original paper on ψ-SSA and the algorithm we present here are both derived from the out of SSA algorithm from Sreedhar et al. [10].

This algorithm uses ϕ congruence classes to create a conventional SSA representation. Two variables x and y are in a ϕ-congruence relation if they are referenced in the same ϕ function, or if there exists a variable z such that x is in a ϕ-congruence relation with z and y is in a ϕ-congruence relation with z. Then we define a ϕ congruence class as the transitive closure of the ϕ-congruence relation. The conventional SSA representation has the property that the renaming of all the resources from a ϕ congruence class into

a representative name, and the elimination of the ϕ operations, will not violate the semantics of the program. The Sreedhar algorithm gives three methods, the third one being the most efficient, to convert an SSA representation into a conventional SSA form.

4.2. Conventional Psi-SSA

We define the conventional ψ-SSA (ψ-*CSSA*) form in a similar way to the Sreedhar definition of the conventional SSA (*CSSA*) form. The congruence relation is extended to the ψ operations. Two variables x and y are in a ψ-congruence relation if they are referenced in the same ϕ or ψ function, or if there exists a variable z such that x is in a ψ-congruence relation with z and y is in a ψ-congruence relation with z. Then we define a ψ congruence class as the transitive closure of the ψ-congruence relation. The property of the ψ-CSSA form is that the renaming into a single variable of all variables that belong to the same congruence class, and the removal of the ψ and ϕ operations, results in a program with the same semantics as the original program.

Now, look at figure 4 to examine the transformations that must be performed to convert a program from a ψ-SSA form into a program in ψ-CSSA form.

Looking at the first example, the dominance order of the definitions for the variables a and b differs from their order from left to right in the ψ operation. Such code may appear after a code motion algorithm has moved the definitions for a and b relatively to each other. We have said that the semantics of a ψ operation is dependent on the order of its arguments, and that the order of the arguments in a ψ operation is the order of their definitions in the dominance tree in the original program. In this example the renaming of the variables a, b and x into a single variable will not preserve the semantics of the original program. The order in which the definitions of the variables a, b and x occur must be corrected. This is done through the introduction of the variable c that is defined as a copy of the variable b, and is inserted after the definition of a. Now, the renaming of the variables a, c and x into a single variable will result in the correct semantics.

In the second example, the renaming of the variables a, b, c, x and y into a single variable will not give the correct semantics. In fact, the value of a used in the second ψ operation would be overridden by the definition of b before the definition of the variable c. Such code will occur after copy folding has been applied on a ψ-SSA representation. We see that the value of a has to be preserved before the definition of b, resulting in the code given for the ψ-CSSA representation. Now, the variables a, b and x can be renamed into a single variable, and the variables d, c and y will be renamed in another variable, resulting in a program in a non-SSA form with the correct semantics.

We will now present an algorithm that will transform a program from a ψ-SSA form into its ψ-CSSA form. This algorithm is made of three parts.

- **ψ-normalize** This part will put all ψ operations in what we call a *normalized* form.
- **ψ-congruence** This part will grow ψ-congruence classes from ψ operations, and will introduce repair code where needed.
- **ϕ-congruence** This part will extend the ψ-congruence classes with ϕ operations. This part is very similar to the Sreedhar algorithm.

We detail now the implementation of each of these three parts.

4.3. Psi-normalize

We define the notion of *normalized-ψ*. When ψ operations are created during the construction of the ψ-SSA representation, as described in [11], they are naturally built in their normalized form. The normalized form of a ψ operation has two characteristics:

- The predicate associated with each argument in a normalized-ψ operation is equal to the predicate used on the unique definition of this argument.
- The order of the arguments in a normalized-ψ operation is, from left to right, equal to the order of their definitions, from top to bottom, in the control-flow dominance tree.

When transformations are applied to the ψ-SSA representation, predicated definitions may be moved relatively to each others. Operation speculation and copy folding may enlarge the domain of the predicate used on the definition of a variable. These transformations may cause some ψ operations to be in a non-normalized form.

In the original algorithm described in [11], *Condition 1* for the definition of the ψ-SSA Consistency was identical to the second characteristic of the normalized form we describe here. However, the original algorithm did not include a specific normalization phase for the out of ψ-SSA algorithm. There are two reasons why this step is now needed. The first reason is that in the original ψ representation, there was no predicate associated with an argument in a ψ operation. Implicitly, this predicate was equal to the predicate used on the definition of the argument, but these predicates can now be different in our representation. The second reason is to fix a problem in the original algorithm. In figure 5, we show an example where copy folding on predicated code has transformed the second ψ operation into a

```
p?   b = ...
     a = ...

     x = Psi(1?a, p?b)
```

Psi – SSA form

```
p?   b = ...
     a = ...
p?   c = b
     x = Psi(1?a, p?c)
```

Psi – CSSA form

```
p?   b = ...
     x = ...
p?   x = b
```

non – SSA form

```
     a = ...

p?   b = ...
q?   c = ...
     x = Psi(1?a, p?b)
     y = Psi(1?a, q?c)
```

Psi – SSA form

```
     a = ...
     d = a
p?   b = ...
q?   c = ...
     x = Psi(1?a, p?b)
     y = Psi(1?d, q?c)
```

Psi – CSSA form

```
     x = ...
     y = x
p?   x = ...
q?   y = ...
```

non – SSA form

Figure 4. ψ-SSA and ψ-CSSA forms

```
p?   a = op1
q?   b = op2
p?   c = a
     x = Psi(p?a, q?b)
     y = Psi(q?b, p?c)
```

a) Normalized form

```
p?   a = op1
q?   b = op2

     x = Psi(p?a, q?b)
     y = Psi(q?b, p?a)
```

b) non normalized form

Figure 5. Copy folding on ψ-SSA representation

non-normalized form. The original algorithm assumed that for variables used in the argument list of ψ operations, it was always possible to define a strict order relation between variables in a congruence class, noted $\succ_c$. This order was determined using the relative order of the variables in the different ψ argument lists where these variables were used. Clearly, in this example, there is no such relation between variables a and b in the non-normalized form of the ψ operations.

PSI-normalize implementation. A dominator tree must be available for the control-flow graph to lookup the dominance relation between basic blocks. The dominance relation between two operations in a same basic block will be given by their relative positions in the basic block.

Each ψ operation is processed independently. An analysis of the ψ operations in a top down traversal of the dominator tree reduces the amount of repair code that is inserted during this pass. We only detail the algorithm for such a traversal.

For a ψ operation, the argument list is processed from left to right. For each argument arg_i, the predicate associated with this argument in the ψ operation and the predicate used on the definition of this argument are compared. If they are not equal, a new variable is introduced and is initialized just below the definition for arg_i with a copy of arg_i. This definition is predicated with the predicate associated with arg_i in the ψ operation. Then, arg_i is replaced by this new variable in the ψ operation.

Then, we consider the dominance order of the definition for arg_i, with the definition of the next argument in the ψ argument list, arg_{i+1}. When arg_{i+1} is defined on a ψ operation, we recursively look for the definition of the first argument of this ψ operation, until a non-ψ operation is found. Now, if the definition we found for arg_{i+1} dominates the definition for arg_i, repair code is needed. A new variable is created for this repair. This variable is initialized with a copy of arg_{i+1}, guarded by the predicate associated with this argument in the ψ operation. This copy operation is inserted at the lowest point, either after the definition of arg_i or arg_{i+1}[1]. Then, arg_{i+1} is replaced in the ψ operation by this new variable.

The algorithm continues with the argument arg_{i+1}, until all arguments of the ψ operation are processed. When all arguments are processed, the ψ is in its normalized form. When all ψ operations are processed, the function will contain only normalized-ψ operations.

The top-down traversal of the dominator tree will ensure that when a variable in a ψ operation is defined by another ψ operation, this ψ operation has already been analyzed and put in its normalized form. Thus the definition of its first variable already dominates the definitions for the other arguments of the ψ operation.

In figure 6 we show how this algorithm works. The first ψ operation is analyzed. The analysis starts with argument a. The predicate associated with this argument is equal to the predicate used on the definition for a, and the definition of a dominates the definition of b, thus no repair code is needed.

[1]When arg_{i+1} is defined by a ψ operation, its definition may appear after the definition for arg_i, although the non-ψ definition for arg_{i+1} appears before the definition for arg_i.

```
        d = ...                          d = ...
p?      a = ...                  p?      a = ...
r?      c = ...                  r?      c = ...
                                 s?      f = d
        b = ...                          b = ...
                                 q?      e = b
p|q?    x = Psi(p?a, q?b)        p|q?    x = Psi(p?a, q?e)
r|s?    y = Psi(r?c, s?d)        r|s?    y = Psi(r?c, s?f)
                                 r|s?    g = y
        z = Psi(p|q?x, r|s?y)            z = Psi(p|q?x, r|s?g)
```

Figure 6. Converting ψ operations into their normalized form

The analysis continues with argument b. The predicate associated with the argument b in the ψ operation is not equal to the predicate used on the definition of b. A new variable e is introduced, and is defined as a predicated copy of b using the predicate associated with b in the ψ operation. Then b is replaced by e in this ψ operation. On the next ψ operation, the definition for c does not dominate the definition for d. A new variable f is introduced and initialized with d under predicate s. This copy operation is inserted just after the definition for c. On the last ψ operation, since y is defined on a ψ operation we use the definition of c as the definition point for y. The definition of x does not dominate the definition for c, so a repair is needed. The copy g = y is inserted after the definition of y, and is predicated with the predicate associated with y in the ψ operation.

This algorithm ensures that the program contains only normalized ψ operations. This property is used by the next two passes of the algorithm.

4.4. Psi-congruence

In this pass, we repair the ψ operations when variables cannot be put into the same congruence class, because their live ranges interfere. In the same way as Sreedhar gave a definition of the liveness on the ϕ operation, we first give a definition for the liveness on ψ operations. With this definition of liveness, an interference graph is built.

Liveness and interferences in Psi-SSA. We have already seen that in some cases, repair code is needed so that the arguments and definition of a ψ operation can be renamed into a single name. Here, we give a definition of the liveness on ψ operations such that these cases can be easily detected by observing that live-ranges for variables in a ψ operation overlap. Our definition of liveness differs from the definition used in the original paper, and allows for more precise detection and repair of the interferences between variables in ψ operations.

Consider the code in figure 7. The ψ operation has been

```
        a = op1                          a = op1
p?      b = op2                          b = p ? op2 : a
q?      c = op3                          c = q ? op3 : b
        x = Psi(1?a, p?b, q?c)           x = c
```

a) Psi-SSA form **b) select form**

Figure 7. ψ and select operations equivalence

replaced by explicit select operations on each predicated definition. In this example, there is no relation between predicates p and q. Each of these select operations makes an explicit use of the variable immediately to its left in the argument list of the original ψ operation. We can see that a renaming of the variables a, b, c and x into a single representative name will still compute the same value for the variable x. Note that this transformation can only be performed on normalized ψ operations, since the definition of an argument must be dominated by the definition of the argument immediately to its left in the argument list of the ψ operation. Using this equivalent representation for the ψ operation, we now give a definition of the liveness for the ψ operations.

Definition *We say that the point of use of an argument in a normalized ψ operation occurs at the point of definition of the argument immediately to its right in the argument list of the ψ operation. For the last argument of the ψ operation, the point of use occurs at the ψ operation itself.*

Given this definition of liveness on ψ operations, and using the definition of liveness for ϕ operations given by Sreedhar, a traditional liveness analysis can be run. Then an interference graph can be built to collect the interferences between variables involved in ψ or ϕ operations.

Repairing interferences on ψ operations. We now present an algorithm that creates congruence classes with ψ operations such that there are no interference between two variables in the same congruence class.

First, the congruence classes are initialized such that each variable in the ψ-SSA representation belongs to its own congruence class. Then, ψ operations are processed one at a time, in no specific order. Two arguments of a ψ operation interfere if one or more variables from the congruence class of the first argument and one or more variables from the congruence class of the second argument interfere. When there is an interference, the two ψ arguments are marked as needing a repair. When all pairs of arguments of the ψ operation are analyzed, repair code is inserted. For each argument in the ψ operation that needs a repair, a new variable is introduced. This new variable is initialized with a predicated copy of the argument's variable. The copy operation is inserted just below the definition of the argument's

```
p?   a = ...                      p?   a = ...
q?   b = ...                      q?   b = ...
                                  q?   e = b
r?   c = ...                      r?   c = ...
                                  r?   f = c
     x = Psi(p?a, q?b, r?c)            g = Psi(p?a, q?e, r?f)
                                       x = g
s?   d = b + 1                    s?   d = b + 1
```

Figure 8. Elimination of ψ live-interference

variable, predicated with the predicate associated with the argument in the ψ operation.

Once a ψ operation has been processed, the interference graph must be updated, so that other ψ operations are correctly handled. Interferences for the newly introduced variables must be added to the interference graph. Conservatively, we can say that each new variable interferes with all the variables that the original variable interfered with, except those variables that are now in its congruence class. Also, conservatively, we can say that the original variable interferes with the new variable in order to avoid a merge of a later ψ or ϕ operation of the two congruence classes these two variables belong to. The conservative update of the interference graph may increase the number of copies generated during the conversion to the ψ-CSSA form.

Consider the code in figure 8 to see how this algorithm works. The definition of liveness on the ψ operation will create a live-range for variable a that extends down to the definition of b, but not further down. Thus, the variable a does not interfere with the variables b, c or x. The live-range for variable b extends down to its use in the definition of variable d. This live-range creates an interference with the variables c and x. Thus variables b, c and x cannot be put into the same congruence class. These variables are renamed respectively into variables e, f and g and initialized with predicated copies. These copies are inserted respectively after the definitions for b, c and x. Variables a, e, f and g can now be put into the same congruence class, and will be renamed later into a unique representative name.

4.5. Phi-congruence

When all ψ operations are processed, the congruence classes built from ψ operations are extended to include the variables in ϕ operations. In this part, the algorithm from Sreedhar is used, with a few modifications.

The first modification is that the congruence classes must not be initialized at the beginning of this process. They have already been initialized at the beginning of the ψ-congruence step, and were extended during the processing of ψ operations. These congruence classes will be extended now with ϕ operations during this step.

The other modification is that the live-analysis run for this part must also take into account the special liveness rule on the ψ operations. The reason for this is that for any two variables in the same congruence class, any interference, either on a ψ or on a ϕ operation, will not preserve the correct semantics if the variables are renamed into a representative name.

All other parts of the algorithm are unchanged, and in particular, any of the three algorithms described for the conversion into a CSSA form can be used.

We have described a complete algorithm to convert a ψ-SSA representation into a ψ-CSSA representation. The final step to convert the code into a non-SSA form is a simple renaming of all the variables in the same congruence class into a representative name. The ψ and ϕ operations are then removed.

Now that a complete algorithm has been described to convert a ψ-SSA representation to a ψ-CSSA representation, we will present some improvements that can be added so as to reduce the number of copies inserted by this algorithm.

4.6. Improvements to the out of Psi-SSA algorithm

Below we present a list of improvements that can be added to the algorithm.

Non-normalized ψ operations with disjoint predicates. When two arguments in a ψ operation do not have their definitions correctly ordered, the ψ operation is not normalized. We presented an algorithm to restore the normalized property by adding a new predicated definition of a new variable. However, if we know that the predicate domains of the two arguments are actually disjoint, the semantics of the ψ operation is independent on their relative order. So, instead of adding repair code, these two arguments can simply be reordered in the ψ operation itself, to restore the normalized property.

Interference with disjoint predicates. When the live-ranges of two variables overlap, an interference is added for these two variables in the interference graph. If the definitions for these variables are predicated definitions, their live-ranges are only valid under a specific predicate domain. These domains are the domains of the predicates used on the definitions of the variables. Then, if these domains are disjoint, then although the live-range overlap, they are on disjoint conditions and thus they do not create an interference in the interference graph. Removing this interference from the interference graph will avoid the need to add repair code when live-ranges on disjoint predicates overlap.

Repair interference on the left argument only. When an interference is detected between two arguments in a ψ operation, only the argument on the left actually needs a repair. The reason is that, since the ψ operations are normalized, the definition of an argument is always dominated by the definition of an argument on its left. Thus adding a copy for the argument on the right will not remove the interference.

Interference with the result of a ψ operation. When the live-range for an argument of a ψ operation overlaps with the live-range of the variable defined by the ψ operation, this interference can be ignored. Actually, there are two cases to consider:

- If the argument is not the last one in the ψ operation, and its live-range overlaps with the live-range of the definition of the ψ operation, then this live-range also overlaps with the live-range of the last argument. Thus this interference will already be detected and repaired.

- If the argument is the last one of the ψ operation, then the value of the ψ operation is the value of this last argument, and this argument and the definition will be renamed into the same variable out of the SSA representation. Thus, there is no need to introduce a copy here.

5. Experimental results

The ψ-SSA representation has been implemented in our production compiler for the ST200 family processors [7]. This compiler is based on the Open64 compiler technology, and the ψ-SSA representation has been used to implement optimizations in the code generator part of the compiler. The experiment has been conducted on a variant of the ST231 processor. The ST231 is a 4-issue VLIW processor that targets multimedia and digital consumer embedded applications. It is composed of four 32-bit integer ALUs, two 32x32 multipliers, one load/store unit, a branch unit, 64 32-bit general purpose registers and 8 1-bit branch registers. The variant we used includes support for partial predication, through predicated load and store instructions and a `select` instruction.

The ψ-SSA representation is used in the backend of our compiler to implement several optimizations. These optimizations include a range-propagation analysis to remove redundant or useless computations, an address expressions analysis to optimize the use of available addressing modes, and an if-conversion algorithm [1]. However, these transformations will only very occasionally produce non-normalized ψ operations or add interferences between variables in ψ operations.

In order to analyze the situations where some repair code is introduced on ψ operations, we added a copy folding algorithm just before the out of ψ-SSA algorithm. We ran our algorithms on a set of small benchmarks from multimedia applications. The results are reported in figures 9 and 10. In these experiments we measured the number of copy operations that were inserted during each of the three steps of the out of ψ-SSA algorithm, and we measured the total number of copy operations in the program after the out of ψ-SSA phase.

In figure 9, we report the figures when no ψ-predicate promotion algorithm was applied. In the first column, we report the number of copies when the if-conversion and the copy folding optimizations are not run. As expected, the transformations that are performed on the SSA representations do not break the ψ-SSA conventional property on these benchmarks, which results in no copy operation being inserted during the out of ψ-SSA phase. The second column shows the results when the if-conversion transformation is performed. A number of copy operations are inserted during the ψ-normalize step, which shows that the if-conversion algorithm generates non-normalized ψ operations. Most of these non-normalized ψ operations are due to the predicate being different on the definition of the variable and on its use in the ψ operation. The ψ-congruence step creates no additional copy operations, which means that no interference was detected between variables on ψ operations during this step. In the third column, copy folding was performed in addition to the if-conversion transformation. This resulted in additional non-normalized ψ operations. These additional non-normalized operations are created when predicated copy operations are folded, resulting in more ψ operations with a different predicate on the definition for a variable and its use in the ψ operation. There is also one interference in the ψ-congruence step that was created by the copy folding. This copy operation cannot be optimized away. The large number of copy operations generated during the ϕ-congruence step is mostly due to the fact that we only implemented the second method of the Sreedhar algorithm, use of the third method would reduce this number. Finally, the total number of copy instructions after the out of ψ-SSA phase is greater after copy folding has been performed, mostly due to the number of copy operations generated during the ϕ-congruence step.

In figure 10, we report the figures when ψ-predicate promotion algorithm was performed. The ψ-predicate promotion propagates into the ψ operations the effect of the speculation that was performed during the if-conversion algorithm. The main reason to perform the ψ-predicate promotion is to reduce the number of predicates that must be computed in the code. This transformation also reduces the number of non-normalized ψ operations, so that fewer copy operations need to be inserted during the ψ-normalize step.

This is shown in the second and third columns for the ψ-normalize step. The number of copy instructions introduced in this step is reduced compared to the number of copy instructions that were introduced in the same step without the ψ-predicate promotion. On the ψ-congruence step, we see that performing the ψ-predicate promotion actually increased the number of interferences to be repaired. In fact, these interferences also existed without the ψ-predicate promotion, but, due to the smaller number of non-normalized ψ-operations, they were not repaired as a side effect of the ψ-normalize step.

On the last line of this figure, we see that after the out of ψ-SSA phase there is a small decrease in the number of copy instructions in the code when ψ-predicate promotion is performed. The cases where fewer copy operations are generated occur in loops where a ψ operation uses and defines variables that are used in the same ϕ operation. Such a situation is described in figure 11. The ψ operation in the code on the left is not normalized, because the predicate for the variable c is different on its definition in the ϕ operation and on its use in the ψ operation. In the code on the right, a variable e has been added to normalize this ψ operation. The ψ-congruence step creates a congruence class with the variables e, d and b, since there is no interference between these variables. The ϕ operation is then processed during the ϕ-congruence step. The interferences between the variable c and the variables in the congruence class for b are checked. In fact, the variables c and e interfere, which will require that a new variable is introduced and a new copy instruction is inserted. When the predicate promotion is performed first on the ψ operation for the variable c, the variable e is no longer introduced. The ψ-congruence step creates a congruence class with variables c, d and b. In the ϕ-congruence step, when processing the ϕ operation, no interference needs to be repaired since the variables b and c are already in the same congruence class, and thus no additional copy instruction is inserted.

Future work will include improving the out of SSA algorithm in order to reduce the number of copies generated during this phase. In particular, we will work on a better integration between the ψ-congruence and the ϕ-congruence steps to avoid the cases where repair code introduced in the ψ-congruence step creates interferences to be repaired in the ϕ-congruence step.

6. Conclusion

In this article we presented several aspects of the ψ-SSA representation. The ψ-SSA representation is an extension of the SSA representation to support predicated code, where some definitions are conditionally executed depending on the value of a guard register. We presented an improvement to the original ψ-SSA representation to better support architectures with only a partially predicated instruction set. We added a new transformation that can be performed on the ψ operations in the context of partial predication, namely the ψ-predicate promotion, which is useful for example in an if-conversion algorithm. Finally, we presented a detailed implementation of the out of ψ-SSA representation algorithm, which includes the support for partially predicated architectures and fixes an error in the original algorithm. The ψ-SSA representation is implemented in our production compiler for the ST200 family processors, and is used to perform several algorithms on the ψ-SSA representation, including an if-conversion optimization.

copies	no if-conv	if-conv	if-conv + folding
psi-normalize	+0	+129	+163
psi-congruence	+0	+0	+1
phi-congruence	+0	+82	+1543
total copies	7041	7268	7948

Figure 9. Out of ψ-SSA without ψ-predicate promotion

copies	no if-conv	if-conv	if-conv + folding
psi-normalize	+0	+34	+65
psi-congruence	+0	+5	+11
phi-congruence	+0	+11	+1493
total copies	7041	7107	7810

Figure 10. Out of ψ-SSA with ψ-predicate promotion

```
     loop :                          loop :
     c = Phi(a, b)                   c = Phi(a, b)
                                 p̄?  e = c
     q = c < 10                      q = c < 10
p?   d = op1                     p?  d = op1
     b = Psi(p̄?c, p?d)               b = Psi(p̄?e, p?d)
q?   goto loop                   q?  goto loop
```

Figure 11. ψ-normalize adds new interferences

7. Acknowledgements

I would like to thank Christian Bruel for his implementation of an if-conversion algorithm under ψ-SSA, Christophe Guillon and Fabrice Rastello for their very useful remarks and feedback, and Stephen Clarke for his review.

References

[1] C. Bruel. If-conversion ssa framework for partially predicated vliw architectures. SIGPLAN, ACM and IEEE, March 2006.

[2] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on ssa form. *ACM SIGPLAN Notices*, 32(5):273 – 286, 1997.

[3] F. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In T. Gyimothy, editor, *Compiler Construction, 6th International Conference*, volume 1060 of *Lecture Notes in Computer Science*, pages 253–267, Linköping, Sweden, 24–26 Apr. 1996. Springer.

[4] C. Click. Global code motion global value numbering. In *SIGPLAN International Conference on Programming Languages Design and Implementation*, pages 246 – 257, 1995.

[5] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451 – 490, 1991.

[6] J. Fang. Compiler algorithms on if-conversion, speculative predicates assignment and predicated code optimizations. In *9th International Workshop on Languages and Compilers for Parallel Computing (LCPC), LNCS #1239*, pages 135 – 153, 1996.

[7] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *The 27th Annual International Symposium on Computer architecture 2000*, pages 203–213, New York, NY, USA, 2000. ACM Press.

[8] K. Knobe and V. Sarkar. Array ssa form and its use in parallelization. In *ACM Symposium on the Principles of Programming Languages*, pages 107 – 120, 1998.

[9] R. Morgan. Building an optimizing compiler, January 1998.

[10] V. Sreedhar, R. Ju, D. Gillies, and V. Santhanam. Translating out of static single assignment form. In *Static Analysis Symposium, Italy*, pages 194 – 204, 1999.

[11] A. Stoutchinin and F. de Ferrière. Efficient static single assignment form for predication. In *34th annual ACM/IEEE international symposium on Microarchitecture*, pages 172–181. IEEE Computer Society, 2001.

[12] A. Stoutchinin and G. Gao. If-conversion in SSA form. In *Proceedings of Euro-Par 2004 Parallel Processing*, volume 3149 of *LNCS*, pages 336–345. SIGPLAN, Springer Verlag, Aug 2004.

[13] M. Wegman and K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181 – 210, 1991.

[14] M. Wolfe. Beyond induction variables. In *SIGPLAN International Conference on Programming Languages Design and Implementation*, pages 162 – 174, 1992.