Chapter 1

Hashed SSA form: HSSA —(*M. Mantione*)

1 Introduction

Hashed SSA (or in short HSSA), as described in [Chow...], is an SSA extension that can effectively represent how aliasing relations affect a program in SSA form. It works equally well for aliasing among local variables and, more generally, for indirect load and store operations on arbitrary memory locations. This allows the application of all common SSA based optimizations uniformly both on local variables and on external memory areas.

It should be noted, however, that HSSA is a technique useful for representing aliasing effects, but not for detecting aliasing. For this purpose, a separate alias analysis pass must be performed, and the effectiveness of HSSA will be influenced by the accuracy of this analysis.

The following sections explain how HSSA works. Initially, given aliasing information, we will see how to represent them in SSA form for scalar variables. Then we will introduce a technique that reduces the overhead of the above representation, avoiding an explosion in the number of SSA versions for aliased variables. Subsequently we will represent indirect operations on external memory areas as operations on ”virtual variables” in SSA form, which will be handled uniformly with scalar (local) variables. Finally we will apply global value numbering (GVN) to all of the above, obtaining the Hashed SSA form.

2 SSA and aliasing: 2 and functions

Aliasing is the situation when, inside a compilation unit, one single storage location (that contains a value) can be potentially accessed through different program ”variables”. This can happen in one of the four following ways:

- First, when two or more storage locations partially overlap. This, for instance, happens with the C ”union” construct, where different parts of a program can access the same storage location under different names.
- Second, when a local variable is referred by a pointer used in an indirect operation. In this case the two ways in which the variable can be accessed are *directly*, through the variable name, and *indirectly*, through its address stored in another variable.
- Third, when the address of a local variable is passed to a function, which in turn can then access the variable indirectly.
- Finally, variables that are not local can obviously be accessed by different functions, and in this case every function call can potentially access every global variable, unless the compiler uses global optimizations techniques where every function is analyzed before the actual compilation takes place.

HSSA, extension to _represent_ aliasing

a scalar?

to perform

excellent!

different steps to obtain HSSA

Aliasing. What it is. Where is comes from.

awkward

you always talk about variables. Is it scalar variables, arrays. Storage are registers, memory locations...? You should be more precise.

The real problem with aliasing is that these different accesses to the same program variable are difficult to predict. Only in the first case (explicitly overlapping locations) the compiler has full knowledge of when each access takes place. In all the other cases (indirect accesses through the address of the variable) the situation becomes more complex, because the access depends on the address that is effectively stored in the variable used in the indirect operation. This is a problem because every optimization pass is concerned with the actual value that is stored in every variable, and when those values are used. If variables can be accessed in unpredictable program points, the only safe option for the compiler is to handle them as ”volatile” and avoid performing optimizations on them, which is not desirable.

Lack of aliasing information => no optimizations

would be nice to have some optimizations that are limited due to aliasing pbs:
- scheduling/code motion issues
- various analysis (such as value-range)
- ...?

awkward

Intuitively, in presence of aliasing the compiler could try to track the values of variable addresses inside other variables (and this is exactly what HSSA does), but the formalization of this process is not trivial. The first thing that is needed is a way to model the *effects* of aliasing on a program in SSA form. To do this, assuming that we have already performed alias analysis, we must formally define the effects of indirect definitions and uses of variables. Particularly, each definition can be a *”MustDef”* operation in the direct case, or a *”MayDef”* operation in the indirect case. We will represent MayDef operations with the χ operator. Similarly, uses can be *”MustUse”* or *”MayUse”* operations (respectively in the direct and indirect case), and we will represent MayUse operations with the μ operator. Obviously the argument of the μ operator is the potentially used variable. Less obviously, the argument to the χ operator operator is the assigned variable itself. This makes so that liveness information is naturally correct: the χ operator only *potentially* modifies the variable, so the original value could ”flow through” it. By making the χ operator use the original value, a store can be considered dead only if it and all its associated χs are not marked live.

mu as mayUse (put just before)
Chi as may def (put just after)
Put it only if variable live @ that point
Then see it as normal instructions

I would start from here with the example below. Because it is not clear when you say "each definition" that you are talking about program points where a given variable might be "redifined".

Do you mean that you introduce mu and Chi only at live points?

The use of μ and χ operations does not alter the complexity of transforming a program in SSA form. All that is necessary is a pre-pass that inserts them before or after the instructions that involve aliasing. Particularly, μ operations must be inserted immediately *before* the involved statement or expression, and χ operations immediately *after* it. This distinction allows us to model call effects correctly: the called function appears to potentially use the values of variables before the call, and the potentially modified values appear after the call.

Ideally, should be placed in parallel to the involved statement. Most IR do not allow that, so a possible trick might be to insert mu just before and Chi just after.

SSA update will work also. This is just about adding more def points...

This can be clarified with an example (FIXME: turn the example into a picture). Consider the following code fragment in C syntax:

on an example

```
i = 2;
if (j) {f();}
else {a = *p; *p = 3;}
return i;
```

and assume that p might alias i, and that function f might indirectly use i but not alter it. The code, after the μ and χ insertion pass, becomes the following:

```
i = 2; if (j)
{$\mu$(i); f();}
else {*p = 3; i = $\chi$(i);}
return i;
```

mu(i) and f() should be represented "in parallel" somehow" because this is f who may access i.

where is a=*p.

since we know that if modified it get the value 3, it would have been better to have something such as chi(i,3)

not sure this is useful as it is trivial. We just need to say that mu and Chi have to be treated as normal instructions.

The same code in SSA form becomes:

```
i1 = 2;
if (j1) {$\mu$(i1); f();}
else {*p = 3; i2 = $\chi$(i1);}
i3 = $\phi$(i1,i2);
return i3;
```

where it is clear that the SSA renaming and ϕ insertion must be applied in the usual way, handling μ and χ operators as every other expression and statement.

then under SSA form

3 Introducing "zero versions" to limit the explosion of the number of variables

zero versioning of variables for which we do not have any information because of aliasing access.
=> avoid explosion in the number of variables

While it is true that μ and χ insertion does not alter the complexity of SSA construction, applying it to a production compiler as described in the previous section would make working with code in SSA form terribly inefficient. This is because χ operators cause an explosion in the number of variable values, inducing the insertion of new ϕ functions which in turn create new variable versions. In practice the resulting IR would be needlessly large, not in terms of instructions, but in number of distinct variable versions. The biggest issue is that the SSA versions introduced by χ operators are mostly useless: since χ definitions are by nature uncertain, the actual value of a variable after a χ definition is unknown: it could be its original value, or it could be the one indirectly assigned by the χ, but any optimization that deals with variable values will not be able to operate on that.

do we have some numbers (just an example to illustrate)?

Intuitively, the solution to this problem is to factor all these useless variable versions together, so that no space is wasted to distinguish among them. We assign number 0 to this special variable version, and call it "*zero version*".

The formal definition of zero version relies on the concept of "*real occurrence*", which is an occurrence of a variable in the original program. Therefore, in SSA form variable occurrences in μ, χ and ϕ operators are not "real occurrences". The idea is that variable versions that have no real occurrence do not influence the program output (because once the program is converted back from SSA form these variables are removed from the code). Since they do not directly appear in the code, and their value is unknown, distinguishing among them is almost pointless. A direct definition of zero versions is that they are versions of variable that have no real occurrence, and whose value comes from at least one χ operator (optionally through ϕ functions). An equivalent, recursive definition is the following:

actual def & _use_?

- The result of a χ has zero version if it has no real occurrence.
- If the operand of a ϕ has zero version, the ϕ result has zero version if it has no real occurrence.

you have to say that you do not have use-def chains for zero versioned variables anymore.

To detect zero versions, we associate a "HasRealOcc" flag to each variable version, setting it to true whenever a real occurrence is met in the code (this can be done while constructing the SSA form). Moreover, we associate to each original (unversioned) program variable a list "NonZeroPhiList", initially empty. The detection algorithm is the following:

Algo for zero versioning.
- Start marking variables that we do _not_ want to be zero versioned has HasRealOcc.

would prefer an exemple (see .dot file attached) to illustrate and no pseudo-code which is straightforward. You can add a reference to the paper. You can summaries the main princips saying that because we do not have def-use chains we use working lists of non already zero versioned, the complexity becoming quadratic in the size of the longest use-def chain. Not to costly in practice according to the authors

- zero version all other variables defined by a Chi
- propagate zero versioning along use-def chains on phi (would be simpler with def-use chains)
- propagate also HasRealOcc to fasten the process (not necessary)

– For each version of each variable:
  - if HasRealOcc is false and the version is defined by a χ, set version to zero
  - if HasRealOcc is false and the version is defined by a ϕ, examine the ϕ operands:
    * if all of them have HasRealOcc true, set the ϕ HasRealOcc to true
    * if one or more has version zero, set ϕ version to zero
    * otherwise, add ϕ to NonZeroPhiList of its variable
– For each program variable, iterate until NonZeroPhiList no longer changes, and for each ϕ in NonZeroPhiList, examine the ϕ operands:
  - if all of them have HasRealOcc true, set the ϕ HasRealOcc to true
  - if one or more has version zero, set ϕ version to zero
  - if any of the two above conditions is met, remove the ϕ from the list

Quadratic but better than having too many variables.

The time spent in the first iteration grows linearly with the number of variable versions, which in turn is proportional to the number of definitions and therefore to the code size. On the other hand, the upper bound of each final iteration on a NonZeroPhiList is the longest chain of contiguous ϕ assignments in the program. All in all, zero version detection in the presence of μ and χ functions does not change the complexity of SSA construction in a significant way, while the corresponding reduction in the number of variable versions is definitely desirable.

I see this more has a motivation (what information we want to propagate along use-def chains that would not sufur not being able to go through zero versioned).

ok to break use-def chains for constant prog. Over-approximation for DCE (but need to mark Chi as actual use).

This loss of information has almost no consequences on the effectiveness of subsequent optimization passes. Since variables with zero versions have no known value, not being able to distinguish them does not affect optimizations that operate on values. On the other hand, when performing DEADCE zero versions must be assumed live. This is necessary because we cannot know their actual liveness status (all their uses appear as uses of generic zero versions, so they are no longer distinct). However also this has no practical effect, because zero versions have no real definitions (they have no real occurrence), so there is no real statement that would be eliminated by the DEADCE pass if we could detect that a zero version occurrence is dead. There is only one case in DEADCE where the information loss is evident. A zero version can be used in a μ and defined in a χ, which in turn can have a real occurrence as argument. If the μ is eliminated by some optimization, the real occurrence used in the χ becomes dead but we cannot detect it (because the χ will not be eliminated). According to [Chow...] this case is sufficiently rare in real world code that using zero versions is anyway convenient.

4 SSA and indirect memory operations: indirect variables

scalar

such as arrays

The techniques described in the previous sections (μ and χ operators introduction and using zero version for SSA variables with no real occurrences or meaningful values) only apply to "regular" variables in a compilation unit, and not to arbitrary memory locations accessed indirectly. So, using them we can apply SSA based optimizations to variables also when they are affected by aliasing, but memory access operations are still excluded from the SSA form.

This situation is far from ideal, because code written in current mainstream imperative languages (like C++, Java or C#) typically contains many operations on data stored in global memory areas. Every heap allocated object is referred through pointers, and

be more precise. Chose a language such as C and use the corresponding terms

typically to access individual object fields offsets are added to the pointer values. Code that works with those objects will usually access the fields directly inside expressions and statements, and the result is that a lot of operations are performed on values stored in indirectly accessed global memory locations. It is also worth noting that operations on array element suffer from the same problem.

we want to handle loads and stores (indirect variables)

The fact that you restrict the beginning of the chapter to scalar variables is confusing. Indeed for me aliasing is between memory locations. so you need to make this clear from the beginning.

The situation is not ideal because conventional SSA techniques only handle local variables without aliasing. By annotating the code with μ and χ operators we can deal with aliasing in local variables, but we are still unable to optimize memory accesses (indirect loads and stores) and all the operations that use values that come from them.

The purpose of HSSA is to handle direct and indirect operations uniformly, and be able to apply all SSA based optimizations on them. To do this, in the general case, we assume that the code intermediate representation supports a *dereference* operator, which performs an indirect load (memory read) from the given address. This operator can be placed in expressions on both the left and right side of the assignment operator, and we will represent it with the usual "*" C language operator. We will then, for simplicity, represent indirect stores with the usual combination of dereference and assignment operators, like in C. Some examples of this notation are:

indirect variable represented under SSA.

- *p: access memory at address p.
- *(p+1): access memory at address p+1 (like to access an object field at offset 1).
- **p: double indirection.
- *p = expression: indirect store.

An intuitive approach to deal with indirect operations in SSA form would be to consider, in each indirect operation, the referred location as an "indirect variable". In this approach we would first put in SSA form all direct variables. At this point each occurrence of the "*" operator would have as argument an address expression, with all its variables in SSA form (in the simplest case this expression would be a single variable). We can then consider each "(*expression)" construct as an indirect variable, because it represents the location referred by the "*" operator, and apply the SSA construction algorithm again on these "variables". For instance, the code "i = *p; p = p + 1; j = *p;", when put in SSA form, becomes "i1 = *p1; p2 = p1 + 1; j1 = *p;". Applying SSA to indirect variables we obtain the notation "i1 = (*p1)1; p2 = p1 + 1; j1 = (*p2)2;". In this notation it is evident that (*p1)1 and (*p2)2 have different values, while, for the properties of SSA code, identical expressions will have the same value. If the code contains multiple indirections (like "*(*p + 1)"), the renaming process must be run separately (and sequentially) for each subsequent level of indirection.

say that your indirect variables are (*p1)1 and *(p2)2 here.
Another interesting example would be:
(*p)=...
...=(*p)
(*p)=...
...=(*p)
Say that (*p) is an indirect variable and provide the corresponding SSA form.

need several "SSA reconstruction" pass for each level of indirection.

5 From *indirect variables to* virtual variables

guess you need to perform kind of value numbering to identify address expressions that are identical.

Indirect variables, as described in the previous section, are intuitive but impractical. The biggest problem they have, from a practical point of view, is that they require the application of the SSA renaming algorithm multiple times. But they also raise the issue of dealing with aliasing among indirect variables in a sound way, just like we do with direct variables using μ and χ operators: the algorithm we showed for placing them only works for direct variables.

I do not understand the difference between indirect variables (that should be instantiated somehow, so using eg a variable...) and virtual variables.

To deal with this, we introduce the concept of "*virtual variable*". The virtual variable of an indirect variable is an imaginary scalar variable that has identical aliasing characteristics as the indirect variable. It represent the location (or better the set of potential locations) referred by the indirect memory operation of the indirect variable. Contrast this with the indirect operation itself, which is the actual code that is emitted by the compiler (the dereferencing operation).

By reading the initial paper I understood that a virtual variable may regroup several indirect variables. This is not clear here.
What is not clear both in the paper and here is how you decide that two occurences of memory access correspond to the same indirect variable or not. For scalars this is straightforward: a temporary name corresponds to a unique storage location. While here a storage location is represented by an expression. That may be why value numbering comes into the game afterward. Then you need to make clear that

The key part of the definition of "virtual variable" is that it must have identical aliasing characteristics as its indirect variable. In this sense, while an indirect variable intuitively *is* the referred memory location, its corresponding virtual variable is a "formal placeholder" for the memory location, with the same aliasing properties. As a placeholder, we handle it like every other scalar variable, and particularly μ and χ operators are inserted in he code in the same pass and according to the same rules. Therefore, each occurrence of an indirect variable is annotated with the μ and χ of its virtual variable. As an added bonus, to improve the accuracy of aliasing information, alias analysis can take into account their address expressions to deduce that two indirect variables do not alias each other (because this means that also their virtual variables will not alias each other). For example, it is clear that *p and *(p + 1) do not alias each other.

SSA renaming on virtual variables works as usual, and in the same pass as with scalar variables. In this context, use-def relationships of virtual variables now represent use-def relationships of indirect variables. Note how this approach effectively solves the SSA renaming problem for indirect variables: their SSA version is the version numbers of their virtual variable (unless the address expression itself has changed version, in which case we need to generate a new number). It is the fact that virtual variables have the same aliasing properties as indirect variables that allow us to perform one single, uniform SSA renaming pass, with μ and χ operators automatically imposing new version numbers as a consequence of indirect operations. Also the zero versioning technique can be applied unaltered to virtual variables.

6 GVN and indirect memory operations: HSSA

In the previous sections we sketched the foundations of a framework for dealing with aliasing and indirect operations in SSA form: we identified the effects of aliasing on local variables, introduced μ and χ operators to handle them, applied zero versioning to keep the number of SSA versions acceptable, considered indirect variables as a way to handle arbitrary memory accesses and defined virtual variables as a concrete (and feasible) way to apply SSA also to them.

However, in practice in HSSA we do not renumber indirect variables. Instead, we apply *Global Value Numbering* (FIXME: Find a bibliographic reference for it) (GVN) to both scalar and virtual variables, handling all of them uniformly. In the resulting code representation value numbers have the same role of SSA versions, which can effectively be discarded.

In this sense virtual variables are just an aid to apply aliasing effects to indirect operations, be aware of liveness and perform SSA renaming (with any of the regular SSA renaming algorithms). They have no real counterpart in the program code, and after GVN has been applied they are not considered anymore.

A program in HSSA form is a sequence of statements (assignments and procedure calls), where the expressions cannot have side effects and are composed by five kinds of nodes:

- Three kinds of leaf nodes: constants, addresses and variables.
- Operand nodes.
- Indirect variables ("*ivar*"), which are half operand and half variable nodes.

Ivar nodes correspond to indirect operations. They have a double role because as operations they behave as operands in the IR (they are not leaf nodes), but as "variables" they represent a memory location that holds a value (identified by its value number). We call them *indirect* variables to distinguish them from the original program scalar variables, which are leaf nodes. But note that ivar nodes, considered as code, are operations and not variables: the compiler back end, when processing them, will emit indirect memory accesses to the address passed as their operand (in practice they correspond to the "*" operator in the C programming language). Their value number is needed because through it they can participate to all value based optimizations (like constant folding and propagation, redundancy elimination, register promotion, dead code elimination...). It is in this way that HSSA extends all SSA based optimizations to indirect operations, even if they were originally designed to be applied to "plain" program variables.

not sure we need this section
An example would be more useful

7 Building HSSA

2 pages

The HSSA construction algorithm.

We now present the HSSA construction algorithm. It is straightforward, because it is a simple composition of μ and χ insertion, zero versioning and virtual variable introduction (described in previous sections), together with regular SSA renaming and GVN application.

SSA form construction:

- Perform alias analysis and assign virtual variables to indirect variables
- Insert μ and χ operands for scalar and virtual variables
- Insert ϕ operands (considering both regular and χ store operations)
- Perform SSA renaming on all scalar and virtual variables

At the end of this step we have code in plain SSA form. The use of μ and χ operands guarantees that SSA versions are correct also in case of aliasing. Moreover, indirect operations are "annotated" with virtual variables, and also virtual variables have SSA version numbers. However, note how virtual variables are sort of "artificial" in the code and will not contribute to the final code generation pass, because what really matters are the indirect operations themselves.

Detecting zero versions:

- In one single pass:
  - perform DEADCE (also on χ and ϕ stores)

    - perform steps 1 and 2 of the zero version detection (up to setting HasRealOcc for all variable versions)
- Perform the remaining passes of the zero version algorithm

At the end of this step the code has exactly the same structure as before, but the number of unique SSA versions had diminished because of the application of zero versions.

GVN application:

- Perform a pre-order traversal of the dominator tree, applying GVN to the whole code (generate a unique value number and var node for each scalar and virtual variable version that is still live)
    - expressions are processed bottom up, reusing existing expression nodes and using variable nodes of the appropriate SSA version (the current one in the DT traversal)
    - two ivar nodes have the same value number if these conditions are both true:
        * their address expressions have the same value number, and
        * their virtual variables have the same versions, or are separated by definitions that do not alias the ivar (possible to verify because of the DT traversal order)

The application of global value numbering is straightforward, as described in [FIXME: find a suitable reference, or maybe write a side bar briefly describing GVN]. As a result of this, each node in the code representation has a proper value number, and nodes with the same number are guaranteed to produce the same value (or hold it in the case of variables). The crucial issue is that the code must be traversed following the dominator tree in pre-order. This is important because when generating value numbers we must be sure that all the involved definitions already have a value number. Since in SSA form all definitions dominate their use, a dominator tree pre-order traversal satisfies this requirement.

Note that after this step virtual variables are not needed anymore, and can be discarded from the code representation: the information they convey about aliasing of indirect variables has already been used to generate correct value numbers for ivar nodes.

Linking definitions:

- The left side of each assignment (direct and indirect, real, ϕ and χ) is updated to point to its var or ivar node (which will point back to the defining statement)
- Also all ϕ, μ and χ operands are updated to point to the corresponding GVN table entry

At the end of this step the HSSA form is complete, and every value in the program code is represented by a reference to a node in the HSSA value table.

8 Using HSSA

1 page

As seen in the previous sections, HSSA is an internal code representation that applies SSA to indirect memory operations and builds a global value number table, valid also for values computed by indirect operations.

This representation, once built, is particularly memory efficient because expressions are shared among use points (they are represented as HSSA table nodes). In fact the original program code can be discarded, keeping only a list of statements pointing to the shared expressions nodes.

All optimizations conceived to be applied on scalar variables work ”out of the box” on indirect locations: of course the implementation must be adapted to use the HSSA table, but their algorithm (and computational complexity) is the same even when dealing with values accessed indirectly.

Indirect operation (ivar) nodes are both variables and expressions, and benefit from the optimizations applied to both kinds of nodes. Particularly, it is relatively easy to implement ”register promotion” (the paper authors call it ”indirect removal”).

Of course the effectiveness of optimizations applied to indirect operations depends on the quality of alias analysis: if the analysis is poor the compiler will be forced to ”play safe”, and in practice the values will have ”zero version” most of the time, so few or no optimizations will be applied to them. On the other hand, for all indirect operations where the alias analyzer determines that there are no interferences caused by aliasing all optimizations can happen naturally, like in the scalar case.

(explain the tradeoff of ”factored HSSA”, which is also explained in the gcc chapter...)

Author: Mantione

This citation is to make chapters without citations build without error. Please ignore it: [**?**].

References