Dynamic Single Assignment

Peter Vanbroekhoven
Department of Computer Science
Katholieke Universiteit Leuven
B-3001 Leuven, Belgium
peterv@cs.kuleuven.ac.be

Multimedia and network applications are becoming immensely popular. When one wants to optimize these kinds of applications, that typically process large amounts of data, the classic compiler optimizations are insufficient. New optimizations have been devised that can achieve much higher gains on these applications, in terms of execution speed as well as memory size and energy consumption. These optimizations are much more complex and an important way of mastering this complexity is by converting the application to dynamic single assignment (DSA) form prior to applying the optimizations themselves. The idea behind this is to abstract the reuse of memory elements so that optimizations need not deal with them. In a sense they get more explicit freedom. After the optimizations the issue of reuse can then easily be handled. While methods exist for conversion to DSA, they are limited in both applicability as well as scalability. Hence a new method is being developed that overcomes both limitations by computing a DSA form of the input program that is different from the one computed by the existing methods. This new method appears to be more scalable and more generally applicable without endangering the possibility for optimizations.

1 DSA, what and why?

Everyone who ever followed a compiler construction course will probably remember an interesting intermediate representation called static single assignment or SSA [ASU86, CFR+91]. Because a lot of parallels can be drawn between SSA and DSA and because SSA is well-known and easier to understand than DSA, first the ideas behind SSA are introduced and then extended to DSA.

In essence the idea of SSA is to discard information about the reuse of variables before doing optimizing transformations by allowing only one definition of each variable to appear in the program text. After those transformations, the idea of reuse is reintroduced, more specifically during register allocation[1] where multiple variables can be allocated to the same register or possibly spilled to the same memory location. The effects of the transformation to SSA form can be perceived in two ways. Most often it is found to give an increase in accuracy for analyses and hence more optimizations can be done, or in other words there is increased freedom for transformations. A maybe more accurate perception is that analyses can be more accurate without an increase in complexity for those analyses and the transformations that use them. The reason is that the analyses do not need to take the reuse of variables into account, yet indirectly it is taken into account in the SSA form. A little example will illustrate this. Suppose the original program contains this piece of code:

```
s1: a = 5;
s2: b = f(a);
```

[1] this term is a bit misleading here in the sense that register allocation also does spilling, i.e. variables are also allocated to memory, not only to registers

```
s3: a = 6;
s4: c = f(a);
```

Here `f` represents some function defined in the rest of the program, such that it has no side-effects. Suppose an analysis shows that it would be advantageous to swap statement `s2` and `s3`. It is easy to see that it is not necessarily possible to just do the swap without changing the meaning of the program. The code above assigns to `b` the value of `f(5)`, but when `s2` and `s3` are swapped, the value assigned is `f(6)` which could be something quite different. The problem arises because of the reuse of variable `a` for storing both the value `5` and `6`. This means we can swap statement `s2` and `s3`, but we need to eliminate the reuse of variable `a`. This is done in the following version of the code above:

```
s1: a1 = 5;
s2: b = f(a1);
s3: a2 = 6;
s4: c = f(a2);
```

Obviously the rest of the code may need to be changed too to account for the renaming of `a`. In this version of the code it is perfectly allowed to swap statement `s2` and `s3`. The approach described here consists of first doing analyses (taking reuse into account) that decide what transformations are useful, then removing reuse where necessary to enable those transformations and in the end possibly introducing reuse where possible. It makes sense however to remove all reuse before all analyses and transformations, i.e. to convert the program to SSA form, and keep the analyses and transformations simpler. There is of course also a little downside to this approach, namely that the effects of a transformation need to be estimated because they are influenced by the effects of variable reuse and this is only known when it is too late. However this fact is not considered a killer.

Although SSA is a very useful intermediate form in compiler construction, it is not sufficient for certain applications that are emerging nowadays. With the rise of the internet, the number of applications that need to process huge amounts of data starts to increase, and so does their importance. Just think of video conferencing, portable MP3-players, medical imaging, and network servers and routers. There is an urgent need to optimize these applications, both for speed as well as energy usage of the memories used for data storage. This second one is especially important in portable systems with limited battery capacity. Methods to do this exist, e.g. Data Storage and Transfer Exploration (DTSE in short) described in [CWD+98].

For methodologies like DTSE however, SSA form is not sufficient. DTSE does, among others, global loop transformations. To get real advantage from a single assignment form, something stronger than SSA is needed. The following example will show this:

```
for (i = 0; i < 1000; i++) {
  s1: t = f(a[i]);
  s2: b[i] = g(t);
}
```

Suppose some analysis shows that it would be advantageous to split this loop in two loops, one containing statement `s1` and one containing statement `s2`. This however can not be done just like that. The reason is that there is still reuse of the variable `t` in the code above, although textually only 1 assignment is present. However splitting the loop is not a textual reordering of the statements, but rather a dynamical one since the executions of statement `s1` will no longer be interleaved with the executions of `s2` at runtime. To solve this, we need a single assignment form that is dynamic. The code above in DSA form is this:

```
for (i = 0; i < 1000; i++) {
  s1: t[i] = f(a[i]);
  s2: b[i] = g(t[i]);
}
```

This code will have no more than 1 assignment to every element of array `t` during execution of the program and hence no dynamical reuse is present. Now there is clearly no problem to split the loop. The reasoning behind the use of DSA is the same as behind SSA. First we discard all information about dynamical reuse of variables, then we do all optimizations and we conclude with reintroducing the reuse as one of the last steps in the DTSE methodology. The reason we do this is the same: we can increase the accuracy of analyses and transformations without increasing the complexity. In case of DTSE, the increase in accuracy is huge.

2 Related work and our own contributions

Methods exist for converting a specific subset of programs to DSA form, e.g. [Fea88]. However these are limited to this subset and an extension of these methods to general programs seems a dead-end. The only extension is found in [Kie00] which does not only allow linear expressions in the indexation of arrays but also modulo and integer division. Even with this extension, there are still a lot of elements in common programs that are not handled, like data-dependent indexing and control flow, while-loops, function calls, ... Also the existing methods do not scale well to the large applications that occur in the field of multimedia.

To overcome these problems, a new method is being devised that essentially chooses another program in DSA form as result. It is easy to see that there exist many programs in DSA form that are equivalent with the original program, but calculating them for the programs we consider is not always equally feasible or even possible. So we need to choose a DSA form for which this is not true. Of course this comes at a price: more memory accesses. However practice often shows that removing the last bit of overhead, in this case extra memory accesses, requires the most work and is often not worth the effort. The idea of conversion to a DSA form where extra memory accesses are introduced to make the conversion feasible and the resulting program still acceptable is part of currently ongoing work and the first results are expected in the near future.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Inc., 1986.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CWD+98] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.

[Fea88] P. Feautrier. Array expansion. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, 1988.

[Kie00] B. Kienhuis. Matparser: An array dataflow analysis compiler. Technical report, University of California, Berkeley, February 2000.