Advanced Copy Propagation for Arrays

Peter Vanbroekhoven* Gerda Janssens Maurice Bruynooghe Henk Corporaal
Francky Catthoor

Abstract

The focus of this work is on a data flow-transformation called advanced copy propagation. After an array is assigned, we can, under certain conditions, replace a read from this array by the right hand side of the assignment. If so, the intermediate assignment can be skipped. In case it becomes dead code, it can be eliminated. Where necessary we distinguish between the different elements of arrays as well as the different runtime instances of statements, allowing us to do propagation over global loop and condition scopes. Running our prototype implementation on some multimedia kernels shows that we can get a decrease in memory accesses from 22% up to 94%.

1 Introduction

Since the rise of the World Wide Web, the number of multimedia and network applications has been growing rapidly. These applications process large amounts of data. In *e.g.*, a C-program this data will be stored in large arrays. Techniques exist to allocate arrays to registers[12], but most arrays will still be in memory. This is however a problem because of the exponentially growing gap between processor speed and main memory speed[8]. Another problem with large, high-speed memories is that they consume much power and hence have a large heat dissipation[3]. Current hardware technology is unable to solve this problem.

This is especially a problem for data intensive applications on embedded systems. The solution is to transform these programs such that the memory accesses and the power consumption are reduced. This is exactly the aim of the Data Transfer and Storage Exploration (DTSE) methodology [3]. One of the steps proposed there is to do advanced copy propagation for arrays[1], and more specifically on programs in dynamic single assignment (DSA) where every variable (thus also every array element) is written only once during the execution of the program. This is a stronger form of single assignment than the well-known static single assignment[1, 12]. Removing copy operations is important since they occur readily in multimedia applications. This can either be because an operation in the program is inherently a copy operation, like a swap of two array elements or a transposition of a matrix, or because introducing extra copy operations simplifies writing a program or reasoning about a program.

The idea of removing copy operations already exist as copy propagation in the classic compiler literature[1, 12]. The idea there is that after an assignment `f = g`, if possible, `g` is used instead of `f`. If then the assignment becomes dead code, it can be eliminated. In that case we save two memory accesses and a memory location. Nonetheless, simple copy propagation fails on the first piece of code in Fig. 1. First note that this piece of code can be part of a larger program, the only thing of importance is that there are no further reads from array `a`, so `S4` is the last read from `a`. We would now like to propagate copy operation `S2` to `S3`. One bump in the road is that the copy propagation from [12] does not take subscripts into account. But even if it did, we cannot just propagate `S2` to `S3` since elements 0 through 49 read by `S3` are not written by `S2` but by `S1` which has a different right hand side. Also if we did propagate, `S2` would not become dead code yet because `S4` still reads `a[50]`. Yet only `a[50]` should be written, *i.e.*, `S2` is dead code for all iterations except for `i` equal to 0.

We show in this work *how to overcome the limitations of classic copy propagation by distinguishing between the different runtime executions of a statement*, called instances. For *e.g.*, `S2` in Fig. 1 we distinguish between 50 instances, one for each value of iterator `i` smaller than 50. This allows us to propagate only part of the instances of a statement and then remove only the instances of the statement that have become dead code. Propagating `S2` to `S3` in the example above, and removing dead code, gives us the second program in Fig. 1. As can be seen, `S2'` is only executed for $\texttt{i} = 0$ and `S3` is split into two statements `S3'` and `S3''` that are conditionally executed to allow propagation to `S3''` only, as required. The net effect is that 49 instances of `S2` are removed and thus 98 memory accesses are removed (49 for `a` and 49 for `b`). To do this we need to introduce extra conditions and there is some code duplication, but executing these is less expensive than memory accesses as far as power consumption is concerned. Also note that elements 51 through 99 are no longer accessed, so the array can be shrunk. This decreases power consumption some more since smaller memories use less power.

*Supported by a specialization grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT)

[1] In [3] it is called advanced signal propagation.

```
for (i = 0; i < 50; i++)
  a[i] = b[i] * b[i];           // S1
for (i = 0; i < 100; i++) {
  if (i < 50) a[i + 50] = b[i]; // S2
  c[i] = a[i];                  // S3
}
temp = a[50];                   // S4
```

```
for (i = 0; i < 50; i++)
  a[i] = b[i] * b[i];           // S1
for (i = 0; i < 100; i++) {
  if (i == 0) a[i + 50] = b[i]; // S2'
  if (i < 50) c[i] = a[i];      // S3'
  else c[i] = b[i - 50];        // S3''
}
temp = a[50];                   // S4
```

Figure 1: *Left*: an example on which classic copy propagation fails. *Right*: Propagated `S2` to `S3` using advanced copy propagation

Name	Δacc	Δacc_r	Δmem	t (s)
LU1	32%	26%	40%	0.191
LU2	37%	26%	45%	0.332
im1	44%	43%	56%	0.367

Name	Δacc	Δacc_r	Δmem	t (s)
im2	95%	94%	100%	1.700
mp3	30%	22%	37%	0.929
divx	31%	31%	100%	0.134

Table 1: Results of advanced copy propagation

To be able to easily distinguish between the runtime instances of statements, we apply our methods to programs in dynamic single assignment (DSA) form. Since array elements are then assigned a value only once, they become equivalent to the values they were assigned, allowing much easier transformations on the data flow. Currently only a subset of imperative programs can be automatically converted to DSA form[6, 9]. Fortunately many multimedia kernels belong to that subset. Hence DSA is already widely used in the context of parallelization [7] and systolic array design [11]. We are currently working on a new method that will be able to extend the subset of programs that can be transformed to DSA. A second issue is that transformation to DSA can have array sizes grow considerably [6, 9]. Advanced methods exist for compacting arrays[5, 15]. When used in combination with loop transformations[4], they can reduce the arrays again – often beyond their original size.

For the technical details of this work, we refer to the full paper [16].

2 Experimental results

Our advanced copy propagation has been implemented using [14] as polyhedral library. The codes we have tested our implementation on have been manually transformed to DSA form and have been made pointer-free by translating pointer references to array references. Also due to current parser restrictions, all irrelevant code was removed so the remaining codes are the computational kernels only. The size of these kernels ranges from 29 to 91 lines of C-code. The results are shown in Table 1. Δacc refers to the decrease in memory accesses (both reads and writes) between the DSA version and the propagated version of the programs. Since DSA conversion was done by hand, we introduced copy operations to make it easier and less error-prone. Hence we also list Δacc_r which gives the "real" decrease in memory accesses between the original and the propagated version. This decrease in memory accesses is important because the power consumption of memories grows as the number of accesses increases. Δmem stands for the decrease in memory size given that the program stays in DSA form. The memory size can change when translated out of DSA form, so the number is only indicative rather than an exact measure. t is the execution time of the advanced copy propagation algorithm.

3 Discussion

In this work we have developed and implemented a method to do advanced copy propagation over global loop and condition scopes of imperative programs in dynamic single assignment form. We apply our methods to programs in dynamic single assignment form. This DSA form is also widely used in the context of parallelization [7] and systolic array design [11]. In [6], Feautrier presents an automated method for dynamic single assignment conversion, but it is limited in both applicability and scalability. We are currently working on a new method that tries to overcome those limitations.

The goals of our methods are similar to those of copy propagation and constant propagation[1, 12] that have been used for a long time in compilers. They often limit themselves to scalar data types. In [1] an extension to arrays is presented. A dependence analysis between array references is required but is left unspecified in [1]. In [12] Muchnick gives a number of dependence tests, ranging from Banerjee's GCD test[2] to Pugh's Omega test[13], which allow the data-flow analysis required for copy propagation to be made more accurate for array accesses. An extension to constant propagation that takes conditional branches into account is given by Wegman and Zadeck[17], but it is limited to branch conditions that evaluate to a constant true or false. To the best of our knowledge, our work is the first attempt to automate copy propagation and constant propagation for programs with arrays by treating each instance of each statement separately,

and thus allowing array propagation on an element basis. Each of the previously existing methods for copy and constant propagation consider all instances of a statement as a whole and hence they do these transformations on an all-or-nothing basis. We can also propagate over conditional statements and global loops that often hinder classic copy and constant propagation.

The intention of doing classic copy propagation is to create dead code that can then be eliminated. Dead-code elimination is described in [1, 12]. Here too compilers often limit themselves to simple data types, while technically the same dependence tests can increase the precision in case of arrays. Partial dead-code elimination is presented in [10] and allows to remove code that is dead on only part of the program paths by moving it down the paths along which the code is not dead. However the code is textually moved without changing the branching structure of the program. Again to the best of our knowledge, we are the first to automate dead-code elimination on a statement instance basis.

The further development of our prototype, together with its integration in the other steps in the framework of [3] will allow us to measure the effects of advanced copy propagation on large multimedia applications.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Inc., 1986.

[2] U. Banerjee. Dependence testing in ordinary programs. Master's thesis, Dept. of Comp. Sci., Univ. of Illinois, Nov. 1976.

[3] 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.

[4] K. Danckaert. *Loop transformations for data transfer and storage reduction on multiprocessor systems*. PhD thesis, K.U.Leuven, 2001.

[5] E. De Greef, F. Catthoor, and H. De Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. In *Proceedings of the Workshop on Parallel Processing and Multimedia of the International Parallel Processing Symposium*, pages 84–98, 1997.

[6] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[7] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. In *IEEE Computer*, December 1996.

[8] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

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

[10] J. Knoop, O. Ruthing, and B. Steffen. Partial dead code elimination. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 147–158, 1994.

[11] H. Kung and C. Leiserson. Systolic arrays for VLSI. In *Sparse Matrix Proceedings*, pages 245–282, Philadelphia, 1978.

[12] S. Muchnick. *Advanced compiler design & implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[13] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomputing '91*, Albuquerque, NM, 1991.

[14] P. Quinton, S. Rajopadhye, and T. Risset. On manipulating $\mathbb{Z}$-polyhedra. Technical report, Institut de Recherche en Informatique et Systemes Aleatoires, 1996.

[15] R. Tronçon, M. Bruynooghe, G. Janssens, and F. Catthoor. Storage size reduction by in-place mapping of arrays. In *Verification, Model Checking and Abstract Interpretation, VMCAI 2002, Revised Papers*, volume 2294 of *LNCS*, pages 167–181, 2002.

[16] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor. Advanced copy propagation for arrays. In *Languages, Compilers, and Tools for Embedded Systems LCTES'03*, pages 24–33, June 2003.

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