Integrating Program Transformations in the Memory-Based Synthesis of Image and Video Algorithms*

David J. Kolson Alexandru Nicolau Nikil Dutt
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425

Abstract

In this paper we discuss the interaction and integration of two important program transformations in high-level synthesis—Tree Height Reduction and Redundant Memory-access Elimination. Intuitively, these program transformations do not interfere with one another as they optimize different operations in the program graph and different resources in the synthesized system. However, we demonstrate that integration of the two tasks is necessary to better utilize available resources. Our approach involves the use of a "meta-transformation" to guide transformation application as possibilities arise. Results observed on several image and video benchmarks demonstrate that transformation integration increases performance through better resource utilization.

1 Introduction

Tree height reduction (THR) [1, 12] is a well-known technique for reducing the critical path length and increasing the parallelism of expressions and/or recurrences through the introduction of redundant computation. THR has been applied to the synthesis of DSP applications [8, 11, 13, 19] and will continue to play an important role in system and architectural level synthesis. The synthesis of memory-intensive behaviors has been identified as important for application areas such as image and video algorithms [7, 16, 20].

Typically, designs synthesized for memory-intensive behaviors contain a secondary (slower) memory, due to the large data size, which is explicitly represented and accessed in the behavior by arrays. It then becomes crucial to optimize memory access in order to obtain acceptable performance. Redundant memory-access elimination (RME) [4, 9, 16] is a technique to remove memory operations (possibly on the critical path) that access locations previously loaded from and/or stored to the memory within and across loop iterations.

Although both RME and THR share the common goal of reducing the critical path length and do not compete in terms of the resource utilization that they optimize, it is important to consider the combined effect of both optimizations to obtain quality schedules for a set of given resources. Previously, no work has adequately addressed this interaction.

As an illustration, consider the code in Fig. 1(a). In Figs. 1(b) and 1(c) two distinct versions of the dataflow graph for the inner loop appear scheduled with the resource constraints of one pipelined two-cycle latency adder and one non-pipelined memory port with two-cycle latency[1]. Both versions consist of eight operations and the height of the tree (i.e., the length of the critical dependency chain) in (b) is four, while in (c) it is five. Although the tree in (c) has greater height, the performance of the resulting schedule is faster than in (b). In (b) operations add redundant[2] and non-redundant memory values (C and D, for instance) whereas in (c) operations add memory values with the same redundancy types (E and D, for instance) resulting in different schedule lengths *given the available resources.*

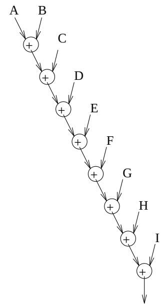
The key to automatically deriving the graph in Fig. 1(c) is to integrate the transformations rather than allowing them to work individually as opportunities arise. In this paper we propose the use of a "meta-transformation" to guide the application of transformations so as to make better decisions concerning the utilization of resources and allow trade-offs between transformations.

*Research partially supported by NSF grant CCR8704367, ONR grant N0001486K0215 and a UCI Faculty Research Grant.

[1] For clarity, no restriction on the number of registers (i.e., temporaries) is imposed and the division of the sum by nine is omitted.

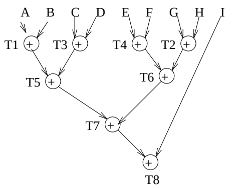
[2] Throughout this paper we use the term *redundant* to refer to computation on memory values loaded and/or stored redundantly in loop execution.

```
For i = 1 to M
  For j = 1 to N
    B[i][j] = (A[i-1][j-1]   /* A */
             + A[i-1][j]     /* B */
             + A[i-1][j+1]   /* C */
             + A[i][j-1]     /* D */
             + A[i][j]       /* E */
             + A[i][j+1]     /* F */
             + A[i+1][j-1]   /* G */
             + A[i+1][j]     /* H */
             + A[i+1][j+1])/9;  /* I */
```



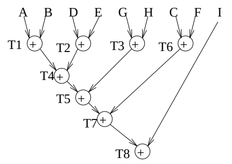
(a)

1	T1 = A + B	Load C	
2	T2 = G + H		A = B
3	T3 = C + D	Load F	B = C, G = H
4			D = E
5	T4 = E + F	Load I	
6	T5 = T1 + T3		E = F
7	T6 = T2 + T4		H = I
8			
9	T7 = T5 + T6		
10			
11	T8 = T7 + I		
12			



(b)

1	T1 = A + B	Load C	
2	T2 = D + E		A = B
3	T3 = G + H	Load F	B = C, D = E
4	T4 = T1 + T2		G = H
5	T6 = C + F	Load I	E = F
6	T5 = T3 + T4		
7			H = I
8	T7 = T5 + T6		
9			
10	T8 = T7 + I		
11			



(c)

Figure 1: Schedules for THR graphs.

2 Previous Work

Previous work has been done in three areas: removal of redundant memory operations, tree height reduction, and integration of program transformations.

Redundant Memory-access Optimization
Techniques which specifically reduce memory accessing by eliminating memory operations are described in [4, 5, 16]. In [6] iteration distance relationships between memory operations are formulated for memory analysis. In [9] we present a technique for removing memory operations that are redundant over loop execution. Our technique uses memory anti-aliasing theory so as to detect redundancy in a general manner.

Tree Height Reduction
Tree height reduction was first studied [1, 12] as a method for reducing critical dependency chains to increase parallelism and was later extended in [3]. Various synthesis systems [8, 19] include support for THR, but do not specifically factor resource availability into the process, or do so in an exhaustive manner [11]. In [13] an *incremental* reduction technique is presented which globally (i.e., over multiple expressions in the program) exploits unused resources by factoring resource availability into the THR conditions.

Integrating Transformations
Work on integrating transformations in compiler theory addresses both coarse-grain (source-code level) and fine-grain (register-transfer level) aspects. In [21] a tool for studying the ordering of coarse-grain transformations is discussed. However, this approach does not allow for trade-offs between transformations. In fine-grain compilation, the thrust is to integrate register allocation and instruction scheduling. Proposed techniques [2, 15] typically make allocation decisions based on the parallelism automatically detected. In [14] resource trade-offs are made dynamically during scheduling but this work does not address transformation interaction.

3 Our Approach

Our approach to integrating THR and RME is to use a "meta-transformation" that makes trade-offs between individual transformations during scheduling. In this way, a global view of the effects of a transformation is maintained and is useful in assessing whether a transform should be applied. Due to the complexity of this task, however, this assessment must rely on heuristics.

3.1 Memory Analysis

In a pre-scheduling phase, all memory operations in the program are analyzed for redundancy to determine which are candidates for removal. This information is propagated throughout the program once it is exposed.

Determining Candidates
Memory operations in our model contain *symbolic expressions* [9] which represent the referenced address in as reduced a form as is possible. Essentially, program variables used in address calculation are "normalized" in terms of a new looping variable. This allows easy dependency testing of memory operations without having to maintain complex relationships between program variables.

Fig. 2 contains an algorithm to determine memory operations which are candidates for removal and is essentially built on top of the techniques presented in [9]. The symbolic expression of each memory operation in the program is compared with all other memory operations' symbolic expressions that reference the same

```
Procedure determine_candidates(program)
begin
   /* collect all memory operations */
   foreach /* memory operation - m1 */ do
      Forall /* other memory ops - m2 */
         if (/* m1 and m2 access same array */) then
            /* dist = m2's sym. expr. - m1's sym. expr. */
            set iters to dist / element_size
            if (/* iters is integer */) then
               /* tag m1 as a candidate for removal */
            endif
         endif
      end
   end
end determine_candidates
```

Figure 2: Determining redundancy candidates.

```
Procedure propagate_redundancy_info(program)
begin
   changes = false
   while (changes) do
      foreach node in program
         foreach op in node
            def_ops = /* ops which define vars that op reads */
            tmp_tags = /* union of red. tags for def_ops */
            changes = changes or tmp_tags ≠ op's tags
            /* tag op with tmp_tags */
         end
      end
   end
end propagate_redundancy_info
```

Figure 3: Propagating redundancy information.

array to determine the distance between two references. When this distance is normalized by the array element size, the number of iterations over which redundancy spans is known and operations are tagged appropriately.

Propagating Redundancy Information

Once redundancy in memory interaction has been determined, that information is propagated throughout the program. Fig. 3 contains an algorithm to propagate redundancy information and is essentially patterned after flow analysis routines. Initially, all operations have no tagging information, except for those memory operations that are redundant, tagged with "r," and those memory operations that are non-redundant, tagged with "n." Then, for each operation in the program, all of the operations that define any variable used by that operation are collected. All of the redundancy tags of those operations are unioned to derive the local redundancy information on that sub-expression.

```
Function META-Transformation(xform, its arguments)
begin
  case xform is:
    Tree-height reduction:
      if ( /* paired args redundancy tags match */) then
        /* recursively descend tree tagging redundant */
        /* memory operations "ready" */
        Return Go_Ahead
      else
        /* "rotate-down" any redundant subtrees */
        Return Inhibit
    Redundant Memory Elimination:
      if ( /* memory op is tagged "ready" */) then
        Return Go_Ahead
      else
        Return Inhibit
  end case
end META-Transformation
```

Figure 4: The META-Transformation

3.2 The META-Transformation

The heuristic that we have selected for integrating THR and RME is a simple greedy scheme that only allows THR to progress when both operands of a new THR operation are either redundant or non-redundant sub-trees and allows RME to eliminate redundant memory operations once they become part of a redundant sub-tree.

The META-Transformation appears in Fig. 4. Because our goal is to pair values with the same redundancy tags, the THR transformation is given the "go ahead" when the tags match. In this case the sub-trees are recursively descended to tag redundant memory operations as "ready" for elimination. If the redundancy tags mismatch, then the redundant sub-trees are rotated. This has the effect of moving computation on redundant values towards the leaves and non-redundant computation towards the root allowing the schedule to better tolerate the latency of memory operations which will not be removed. The RME transformation is inhibited from removing redundant memory operations until they become part of redundant sub-trees (i.e., tagged as "ready"). This strategy has the effect of minimizing the live ranges of redundant memory values stored in the register file.

4 Experiments and Results

Our approach has been implemented in our Percolation-based scheduler [17] with which we conducted some experiments. Twelve core routines in image and video algorithms were used. The spatial filters, edge enhancements and blurring benchmarks were obtained from [10], while the compression benchmarks—wavelet and predictor-corrector (pred-

Table 1: Experimental results.

Benchmark	FUs.	w/	w/o	Impr.
Spatial filters				
General coeff.	2+,1*	18	20	11%
Low-pass				
median	2+, 1*	11	12	9%
powers of 2	2+	12	14	17%
High-pass				
median	2+, 1*	15	17	13%
powers of 2	2+	13	16	23%
Edge enhancement				
Laplace1	2+, 1*	10	12	20%
Laplace2	2+	16	19	19%
north-gradient	2+	15	18	20%
matched-edge	2+	21	25	19%
Blurring	2+	28	31	11%
Compression				
wavelet	2+, 1*	15	18	20%
pred-corr	2+	9	11	22%

corr)—were obtained from [18]. Although the basic structure of many of the benchmarks is similar—namely, the loading of a neighborhood of values, multiplication of those values by respective coefficients and then summation to produce a new value—the expression trees become vastly different due to positive and negative coefficients. Furthermore, resource utilization differs due to the possibility that a coefficient is a power of two (resulting in shifts).

Schedules were generated with latency parameters of two-cycles for add, three-cycles for multiply and two-cycles for load/store. The functional resources used included two adders, one multiplier[3] and a two-port memory. Table 1 presents our observed results. The column labelled "FUs" indicates the functional unit resources used. The columns labelled "w/" and "w/o" contains the number of cycles for the inner loop schedules produced with and without the META-transformation, respectively, while the last column indicates the percentage improvement of our technique.

Our results convincingly demonstrate that integration of the transformations results in better performance. Improved performance is due largely to the ability to perform a significant amount of computation on the redundant portions of the algorithm during the latency of non-redundant memory operations.

[3] In some cases the multiplier was not necessary due to shifting.

References

[1] J. L. Baer and D. P. Bovet. Compilation of Arithmetic Expressions for Parallel Computations. *Proc. of IFIP Congress*, pages 34–46, 1968.

[2] D. G.. Bradlee, S. J. Eggers, and R. R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. *ASPLOS*, 26(4), April 1991.

[3] R. P. Brent. The Parallel Evaluation of General Arithmetic Expression. *Journal of the ACM*, 21(2), 1974.

[4] D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Architectures. *ICPP*, 1987.

[5] J. W. Davidson and S. Jinturkar. Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses. *PLDI*, 29(6), June 1994.

[6] E. Duesterwald, R. Gupta, and M. Soffa. A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimizations. *PLDI*, 28(6), June 1993.

[7] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers. Norwell, MA., 1992.

[8] Z. Iqbal, M. Potkonjak, S. Dey, and A. Parker. Critical Path Minimization Using Retiming and Algebraic Speed-Up. *30th DAC*, 1993.

[9] D. J. Kolson, A. Nicolau, and N. Dutt. Minimization of Memory Traffic in High-Level Synthesis. *31st DAC*, June 1994.

[10] J. S. Lim. *Two-Dimensional Signal and Image Processing*. Prentice Hall Signal Processing Series, 1990.

[11] D. A. Lobo and B. M. Pangrle. Redundant Operator Creation - An Optimized Scheduling Technique. *28th DAC*, 1991.

[12] Y. Muraoka. *Parallelism Exposure and Exploitation in Programs*. PhD thesis, Univ. of Ill. Urbana-Champagne, 1971.

[13] A. Nicolau and R. Potasman. Incremental Tree Height Reduction for High-Level Synthesis. *28th DAC*, 1991.

[14] S. Novack and A. Nicolau. Mutation Scheduling: A Unified Approach to Compiling for Fine-Grain Parallelism. *Proc. 7th Int'l Wksp on Lang. and Comp. for Par. Computing*, 1994.

[15] S. S. Pinter. Register Allocation with Instruction Scheduling: A New Approach. *PLDI*, 1993.

[16] P. Pöchmüller, M. Glesner, and F. Longsen. High-Level Synthesis Transformations for Programmable Architectures. *Euro-DAC '93*, 1993.

[17] R. Potasman, J. Lis, A. Nicolau, and D. Gajski. Percolation Based Synthesis. *27th DAC*, 1990.

[18] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition, 1992.

[19] H. Trickey. Flamel: A High-Level Hardware Compiler. *IEEE Trans. on CAD*, 6(2), March 1991.

[20] J. Vanhoof, K. Van Rompaey, I. Bolsens, G. Goossens, and H. De Man. *High Level Synthesis for Real Time Digital Signal Processing*. Kluwer Academic Publishers. Norwell, MA., 1993.

[21] D. Whitfield and M. L. Soffa. Investigating Properties of Code Transformations. *ICPP*, 1993.