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, such as image and video algorithms, have been identified as important application areas [7, 16, 20].

Typically, designs synthesize 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. 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 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 CCR8701367, ONR grant N0001480K0215 and a UCI Faculty Research Grant.

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

2Throughout this paper we use the term redundant to refer to computation on memory values loaded and/or stored redundantly in loop execution.
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
Figure 2: Determining redundancy candidates.

Figure 3: Propagating redundancy information.

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.

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-
cor) 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.