Transformation-Based Register Optimization in High-Level Synthesis

Ramesh Karri and Alex Orailoğu
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093

Abstract

Rapid advances in VLSI technology have made it feasible to fabricate self-testable and/or self-recovering microarchitectures. Registers in these design methodologies are a precious resource. Consequently, strategies for minimizing the register requirements of a microarchitecture are crucial. In this paper, we propose a novel approach to temporary register minimization that exploits the power of behavioral transformations. Behavioral transformations allow us to reduce the register requirements below the inherent lower bound imposed by the structure of the input flow graph. Specifically, the transformations minimize the lifetimes of registers in a scheduled flow graph. The transformations are applied across clock cycle boundaries with peak register usage. Preliminary results on the benchmark examples show a significant reduction in the number of registers.

1 Introduction

The flexibility afforded by VLSI to pack more circuitry in a smaller, reliable, and less expensive package has made it feasible to realize self-testable and self-recovering microarchitectures on ICs. Whereas self-recovering microarchitectures provide detection and fault-recovery within the IC itself, self-testable microarchitectures enhance the testability of complex ICs. Registers in these design methodologies are a precious resource. For example, self-recovery can be accomplished via checkpointing and rollback. At a checkpoint, duplicate computations are compared using fault-tolerant voting circuitry which is expensive. The number of voters is equal to the number of registers at the checkpoint. Moreover, the lifetimes of these registers need to be extended until the next checkpoint. Consequently, strategies for minimizing the voting requirements of a self-recovering microarchitecture are crucial. Another scenario where minimizing registers is an important consideration is when they constitute a sizeable percentage of the hardware used to synthesize a microarchitecture. A transformation-based strategy to minimize the register requirements of the resulting microarchitecture, possibly below the derivable lower bounds, forms the subject matter of this paper.

Previously, techniques for temporary register minimization have been applied on a flow graph either before or during the scheduling phase of high-level synthesis. A major shortcoming of the previous approaches is that they, at best, use only the minimum number of registers as determined by the lower bound of the specific input flow graph.

1.1 Research Contributions of This Paper

The minimum number of temporary registers required to synthesize a microarchitecture is determined by the specific input flow graph representation. Alternate flow graph structures can result in smaller number of temporary registers. For example, figures 1a and 1b are different representations of the same algorithm with different register requirements. Assuming one adder per clock cycle, the schedule in figure 1b requires one temporary register, while 1a requires two temporary registers. This key observation motivated a transformation-based strategy to minimize temporary register requirements of the resulting microarchitecture. The research contributions of our work can be summarized as follows:

1. Identifying Register Minimizing Transformations: We identify a set of transformations that minimize the number of temporary registers required across a clock cycle boundary. Additionally, we formulate the triggering conditions for such transformations.

2. Transformation-Based Register Minimization: The transformations are selectively applied on a scheduled flow graph. The transformations are guided
by the criticality of register usage across clock cycle boundaries.

The rest of the paper is organized as follows. In section 2 we describe the assumptions. In section 3 we present transformation-based register minimization. The system has been used to schedule several benchmark flow graphs. In section 4, we present experimental results. Finally, section 5 concludes this paper.

2 Assumptions

In our model of flow graph representation, inputs to and outputs from the flow graph are read from or written into registers that are not overwritten during the computation. Consequently, these registers (also known as the architectural registers) are not the subject of the minimization strategies proposed in this paper. Furthermore, these inputs are assumed available in every clock cycle in a schedule without the ensuing register penalty. Throughout the paper, the term register implies a temporary register that is used to store an intermediate result during the computation. Additionally, we assume that there is no chaining of operators.

The input performance constraints include the maximum number of clock cycles, \( C \), while the hardware constraints include number of functional units of each type of operation \( (FU_i) \). The technique is applied after scheduling but prior to register allocation and hence is easily integrable into any high-level synthesis framework.

3 Transformation-Based Register Minimization

The problem of transformation-based register minimization is formulated as follows: Given a scheduled flow graph \( DFG^S \) which uses \( C \) clock cycles, \( FU_i \) functional units of the \( i^{th} \) type and \( R \) registers, apply transformations to minimize the registers without either sacrificing performance or increasing the hardware cost. The algorithm is outlined in figure 2. Initially, \( L \) is set to 1, and \( H \) is set to \( C \), the input clock cycle constraint.

Transformations are applied on a scheduled flow graph across a clock cycle boundary, \( C_{\text{crit}} \), with the peak register utilization \( (R) \) guided by (a) the distribution of the utilized hardware and consumed registers derived from the schedule, and (b) the effects of preceding transformations. Transformations are not applied on an unscheduled flow graph as utilized hardware and register distributions are not available to trigger them. Initially, the input flow graph is scheduled. The scheduling algorithm uses a heuristic to balance the utilization of the available hardware in addition to minimizing the register consumption. Subsequently, the transformation-based register minimization algorithm is repeatedly invoked until no further register minimization is possible. The recursive invocation within the register minimization routine ensures that reduced register counts across clock cycle boundaries with peak

![Figure 2: Transformation-Based Register Minimization](image)

register utilization, due to an earlier invocation, are not incremented.

3.1 Register Minimizing Transformations

Flowgraph transformations have been extensively studied, in the context of programming language compilers. A set of basic transformations can be applied to any arithmetic and/or boolean operations; such transformations can be found in most programming language/compiler textbooks [1]. Specialized compositions of basic transformations, like tree height reduction, are explored in architecture textbooks to exploit parallelism [6]. Furthermore, both basic and specialized transformations have been applied in the context of high-level synthesis either to improve performance or to reduce hardware costs [9, 10, 12, 5].

We explore the register minimizing capabilities of a set of basic transformations such as associativity, distributivity, and duplication. Additionally, we outline specialized transformations to minimize the register requirements. These transformations are easy to implement, simple to manage, and effectively accessible from within an interactive synthesis framework. A detailed discussion of the transformations follows.

3.1.1 Associativity

Associativity can affect the critical path of a design, by reducing the overall height of the flowgraph. Such height reduction can be utilized either to induce additional mobility into the flowgraph or alternatively to reduce the number of control steps needed for instantiation. Not all arithmetic operations are associative; for example, operations like subtraction are nonassociative. An alternate form of associativity may apply to such operations, as shown in Fig. 3.

Selective use of associativity reduces register requirements. Registers are reduced both in terms of the sum of the lifetimes of the intermediate values involved and in terms of peak utilization across clock cycle transitions. Peak utilization is not only reduced but is also shifted; such shifts can balance register usage across clock cycles, leading to a drastic reduction in peak register usage. In cases when register counts are of added
importance, as in determining rollback registers during self-recovering data path synthesis, such effects can significantly impact chip real estate.

Figure 4: Associativity and its effect on register usage

Associativity-induced flowgraph rearrangements reduce register lifetime by equalizing the wait time of intermediate values before subsequent operations can be activated. We illustrate such effects of associativity in figure 4. The edges of the flow graph are annotated with the projected lifetimes. Associativity can have global effects too. The far-reaching mobility injections and shifts in peak usage make it an important transformation for register minimization. In applications where peak register usage is of critical importance associativity can be very helpful as it not only reduces register usage but also shifts register distribution.

3.1.2 Distributivity

To apply distributivity, a distributing operation, usually a multiply or a divide, needs to be directly sourced to a distributed operation, usually an add or a subtract. Furthermore, the intermediate value should not be multiply utilized. When distributivity is applied to noncritical subpaths, inversion of the dependency relationship allows dispersion of the duplicated elements into clock cycles with underutilization of the distributing hardware type. As a result, localized applications of distributivity can be used to improve performance and/or reduce hardware requirements.

Although the effect of distributivity on registers is usually negative, register requirements can decrease when the difference between the control steps that the two candidate nodes have been assigned to is large. The inverted dependency order can be used to reduce the register requirements, in such a scenario. The distributivity transformation can also be used as a pre-cursor to associativity or tree-height reduction. Frequently, when the critical path needs to be reduced, distributivity may generate a flowgraph dependency order, amenable to critical path reduction.

Inverse distributivity, on the other hand, can help reduce registers as a result of the reduction in the number of paths needed to carry intermediate values. Unless the new flowgraph dependencies result in scheduling into widely separated clock cycles, the register requirements will undergo a reduction.

3.1.3 Common Subexpression Elimination

This transformation is illustrated in Fig 6. The left-to-right transformation corresponds to Common Subexpression Elimination. Common subexpression elimination reduces register requirements, since it merges two paths into one, thereby resulting in a single register at the fanout point.

3.1.4 Duplication

Interestingly, the inverse of common subexpression elimination, namely duplication (the right-to-left transformation in figure 6), can also be used to reduce register requirements. When some of the multiple destinations of an operation are spaced apart, and the operation inputs have long lifetimes, it may be preferable to occasionally regenerate a value, instead of saving it in a critical register. The ensuing register reduction is illustrated in figure 7. The duplicated operation node \(X_d\) reduces register requirements by exploiting the long lifetimes of its ancestors; the intermediate result of operation \(X\) need not be saved until \(Y\) is ready to execute. It is especially worthwhile to search for such transformational contexts in specialized design methodologies like design for testability and design for self-recovery where the registers consume a sizable portion of the chip real-estate.

\(^1\) This may happen as a result of tight hardware criticalities for the distributing node type at intermediate clock cycles.
3.1.5 Specialized Transformations

Previously specialized transformations such as tree height reduction and redundant operation creation [9] have been proposed to improve performance. Here we present three specialized transformations that will help reduce register consumption. All three are illustrated in figure 8. The first, Special Transform 1, in addition to reducing the long lifetime of "A", also reduces the length of the path, and moves the multiplication one step down. It not only minimizes the peak register count but also induces additional mobility, and changes the possible clock cycles where the multiplier can be scheduled. Similarly, special transform 2 and special transform 3 and their effects are illustrated in figure 8. Intuitively, the specialized transformations eliminate repeated usage of a variable, thereby decreasing its lifetime. Moreover, these transformations are composed using the basic ones.

A high-level description of the algorithm for applying the register minimizing transformations is given in figure 9. A critical clock cycle boundary has peak register utilization. Candidate transformations help reduce/shift such peak register utilization, while such transformations can be applied on candidate flow graph structures.

```plaintext
Transform (DFG', R, Ccrit)
{
    while (register usage of Ccrit >= R)
    {
        Identify candidate transformations
        Estimate their effects
        Apply a transform to reduce peak register usage
        If register usage at Ccrit does not reduce
            return FALSE
    }
    return TRUE
}
```

Figure 9: Application of register minimizing transformations

4 Experience with the Framework

We present preliminary results of transformation-based register minimization on two scheduled flow graphs. The examples include a fifth order elliptic filter and an AR filter. A time-and-resource-constrained scheduler has been used to schedule the flow graphs. This can be replaced with the schedulers proposed in literature.

In the following experiments, we assume that: (i) the cost of a multiplier = 2 units, cost of an adder = 1 unit, and cost of a register = 1 unit. The duration of a control step = the delay through a multiplier = the delay through an adder. For simplicity, we do not include the overhead due to interconnect in the analyses.

Initially transformation-based register minimization is applied on two schedules of the elliptic filter. The first schedule uses two adders and a multiplier, while the second schedule uses three adders and a multiplier. The original schedules are shown in figures 10a, 10c respectively, while the results of transformation-based register minimization are shown in figures 10b, 10d. The schedules without transformations use 8 and 7 registers respectively. The transformations improved upon these values (6 and 6 respectively), without compromising on performance or increasing the functional unit costs. The transformations applied include associativity, duplication, and Special Transform 1. The reduction in register costs is 25% and 14.3% respectively, while the reduction in total costs (register + functional units) is 16.66% and 8.33% respectively.

For clarity, we will outline the order in which the transformations are applied on the flow graph in figure 10c to transform it to the flow graph form in figure 10d. Initially, the clock boundary 12-13 with a register count of eight registers is targeted for minimization.
Figure 10: Fifth Order Elliptic Filter: Results of transformation-based register minimization. The clock cycle boundaries are annotated with the number of registers.

Associativity (on the output of node +x16), followed by two steps of special transform 1, reduced the peak register consumption to six. Additionally, two steps of duplication are applied to minimize the total register lifetime.

A similar experiment was performed on an AR filter flow graph. The input schedule used 2 adders and 2 multipliers. The improvement in the register costs was 16.6%, while the improvement in the cost of the total design was 8.3%. In this example, two steps of associativity were invoked. The results are shown in figure 11. Each clock cycle boundary is annotated with the register usage across it.

5 Concluding Remarks

A major contribution of this research involved exploiting the register minimizing capability of transformations. We provide a register minimization module that can be used in conjunction with a variety of scheduling algorithms with remarkable ease. Furthermore, our technique can be applied on scheduled flow graphs that have already been optimized for registers to further minimize the register requirements. Although the proposed approach overcomes the inherent lower bound imposed by the structure of the initial specification, some of the transformations may have undesirable side-effects such as finite word length effects.

References