Behavioral Optimization using the Manipulation of Timing Constraints

Miodrag Potkonjak
Department of Computer Science, University of California, Los Angeles, CA 90024
Mani B. Srivastava
AT&T Bell Laboratories, Murray Hill, NJ 07974

Abstract

We introduce a transformation, named rephasing, that manipulates the timing parameters in control-dataflow graphs (CDFGs) during the high-level synthesis of datapath intensive applications. Timing parameters in such CDFGs include the sample period, the latencies between input-output pairs, the relative times at which corresponding samples become available on different inputs, and the relative times at which the corresponding samples become available at the delay nodes (delay nodes in the CDFG of an ASIC correspond to state variables in the state-space). While some of the timing parameters may be constrained by performance requirements, or by the interface to the external world, others remain free to be chosen during the process of high-level synthesis.

Traditionally high-level synthesis systems for datapath intensive have either assumed that all the relative times, called *phases*, when corresponding samples are available at input and delay nodes are zero (i.e., all input and delay node samples enter at the initial cycle of the schedule), or have automatically assigned values to these phases as part of the datapath allocation/scheduling step in the case of newer schedulers that use techniques like overlapped scheduling to generate complex time shapes.

Rephasing, however, manipulates the values of these phases as an algorithm transformation before the scheduling/allocation stage. The advantage of this approach is that phase values can be chosen to transform and optimize the algorithm for explicit metrics like area, throughput, latency, power, register cost etc. Moreover, the rephasing transformation can be combined with other transformations such as algebraic transformations. We have developed techniques for using rephasing to optimize a variety of design metrics, and our results show significant improvements in several design metrics (throughput, area, power, fault-tolerance and partial-scan testability overhead). We have also investigated the relationship and interaction of rephasing with other high level synthesis tasks.

1.0 Introduction

1.1 Manipulation of Timing Relationships During Algorithm Transformations

Algorithm transformations have emerged as important tools in the process of high-level synthesis. By restructuring the user specified algorithm prior to the datapath synthesis and scheduling steps, algorithm transformations help convert it to a functionally equivalent form that is better suited to meeting various design goals and constraints such as high throughput, low latency, low area, low power etc. For the most part algorithm transformations in high level synthesis and compilers have relied on the following three techniques: dataflow optimizations based on algebraic identities (distributivity, commutativity, associativity, the inverse element law, etc.) and redundancy manipulation (common sub-expression elimination and replication, copy propagation, constant propagation [Fis88] etc.), reorganization of control flow structures such as conditionals and loops (retiming, loop unfolding, loop permutation, etc.), and change in the algebraic structure underlying the computation (replacing multiplications by additions in logarithm domain, strength reduction, linear transformations, etc.).

However, the focus of our work is on an algorithm transformation, named *rephasing*, that belongs to a new category - it manipulates the timing relationships between different parts of a computation. While the precise positioning of operations along the time axis is the job of the scheduler, an algorithm transformation like rephasing can intelligently manipulate timing relationships without violating specified timing constraints so as to veer the synthesis process towards optimizing various design metrics or meeting certain design goals.

1.2 What is New?

Rephasing is targeted at computationally intensive applications which are processing streams of data. Typical examples include many of DSP, video, control, and communications designs. The control-dataflow graphs (CDFGs) that are used during the high-level synthesis of these ASIC computationally intensive systems have associated timing parameters such as sampling period, the latencies between input-out pairs, the relative times at which corresponding samples become available on different inputs, and the relative times at which the corresponding samples become available at the delay nodes (delay nodes in DSP CDFGs correspond to state variables in the state-space). While some of the timing parameters may be constrained by performance requirements, or by the interface to the external world, others remain free to be chosen during the process of high-level synthesis. For example, while the sample period is usually a specified performance constraints, the relative times at which corresponding samples become available at the delay nodes are internal timing parameters that are free to be chosen by an implementation.

Traditionally high-level synthesis systems have either assumed that the relative times, called *phases*, when corresponding samples are available at input and delay nodes are all zero (i.e., all input and delay node samples enter at the initial cycle of the schedule), or have automatically assigned values to these phases as part of the datapath allocation/scheduling step in the case of newer schedulers that use techniques like software pipelining and overlapped scheduling to generate complex time shapes.

As the name suggests, rephasing manipulates the values of these phases as an algorithm transformation before the scheduling/allocation stage. The advantage of our approach is that phase values can be chosen to transform and optimize the algorithm for explicit metrics like area, throughput, latency, power, register cost etc. Moreover, the rephasing transformation can be combined with other transformations such as algebraic transformations. Compared to retiming and functional pipelining, the rephasing transformation not only preserves their key advantages but also offers improvements such as elimination of input/output and granularity bottlenecks, preservation of numerical properties, and no need for initial state computation.

We have developed techniques for using rephasing to optimize a variety of design metrics, and our results show significant improvements in several design metrics (throughput, area, power, fault-tolerance and partial-scan testability overhead). In particular, we conducted a detailed investigation of the use of rephasing to optimize for area using a newly developed and statistically validated cost function. We have also investigated the relationship and interaction of rephasing with other high level synthesis tasks.

1.3 Related Work

Relevant to the manipulation of timing relationships by rephasing is the handling of timing constraints that is done by the schedulers during high level synthesis. While there are several notable exceptions [Ku92, Fil93, Ber93, Bha94]], most of high level synthesis work, and in particular the work in DSP and numerically intensive application, has been based on the synchronous dataflow model of computation. As pointed out earlier, most high-level synthesis systems for DSP, video and other numerically intensive applications either assume that all input and delay node samples are available at the same time (all phases are zero) [McF88, Not91, Rab91, Wal91], or indirectly assign values to the phases by using schedulers that incorporate techniques such as overlapped scheduling and software pipelining to generate complex time shapes [Goo89, Lee92, Pot94, Lam88].

However only recently has some limited work been done on relaxing the assumption that all phases are zero and explicitly manipulating the phases. In one such effort Iqbal et al. [Iqb93] proposed an algebraic speed-up algorithm that satisfies an arbitrary set of timing constraints on inputs and outputs. However, they restricted the application of a more general timing constraints scheme only to intermediate optimization steps that combine retiming and dataflow optimizations. Perhaps the first direct effort at directly manipulating the phases as part of an algorithm transformation was described by Srivastava and Potkonjak [Sri94] who applied it to simultaneously optimize throughput and latency for linear DSP systems. Besides targeting a narrower set of goals for the limited domain of linear systems, Srivastava and Potkonjak [Sri94] assumed that the corresponding samples at all the inputs are mutually aligned to each other (all input phases were zero), and the corresponding samples at all the delay nodes in CDFG were mutually aligned to each other (all delay node phases were a constant).

Also of interest to us is the analogous problem of timing in logic synthesis. In fact, Iqbal et al. [Iqb93] exploited techniques from the logic synthesis domain and applied them to high level synthesis. At a conceptual level it can be argued that the rephasing of delay nodes in a CDFG corresponds to *cycle borrowing* in logic synthesis, and that the rephasing of inputs corresponds to *wave pipelining*. Wave pipelining is a technique for pipelining digital designs that can increase the clock frequency without introduction of additional memory elements. In wave pipelining, multiple instances of input data are sent through a block of combinational logic at a rate faster than the maximum delay of the logic. It has been employed in commercial computers [And67] and theoretically studied [Cot65] since the mid-sixties. More recently attention has been given to this transformation from a logic synthesis viewpoint [Won89, Lam92]. Several implementations of wave pipelined designs have been also reported [Lin88, Won92]. A comprehensive coverage of the theoretical and practical issues in wave pipelining is given in [Gra93]. Cycle borrowing - also known as cycle stealing, retardation, slack steal, and transparent latch - is a clocking technique where signal propagation steals a portion of the next clock cycles without changing the functionality of the design. Its application has been discussed in several studies [Noi82, Gla85, Ung86, Ish90].

1.4 Organization of paper

The remainder of the paper is organized as follows. We first introduce key definitions and formally define rephasing. After that we discuss the relationship of rephasing to retiming and other transformations. Using the relationship to functional pipelining, we establish an alternative definition of rephasing which is subsequently used in many applications. Section 3 discussed the use of rephasing as a tool for throughput optimization. A polynomial time algorithm for achieving the iteration bound is presented. Section 4 is used to demonstrate the versatility of rephasing by showing how it can be used to optimize power consumption, area, permanent and transient fault tolerance overhead and testability overhead. Brief description of the interaction of rephasing with other transformations and with other high level synthesis tasks is presented in Section 5. We conclude by presenting a comprehensive experimental evaluation of rephasing, the proposed objective function, and the optimization algorithms.

2.0 Rephasing: Definition and Intuition

2.1 Representation of DSP Systems by CDFG

The DSP systems that we are interested in have multiple inputs, multiple outputs, and finite number of states. They accept streams of samples on each of the inputs, and produce streams of samples on each of the output ports. We represent an algorithm for a system by a hierarchical directed control-dataflow graph (CDFG). In a CDFG the nodes represent data operators or subgraphs, data edges represent the flow of data between nodes, and control edges represent sequencing and timing constraints between nodes.

We restrict ourselves to operators that are synchronous in that they consume at every input, and produce at every output, a fixed number of samples on every execution. This restriction has two interesting ramifications. First, the operators, and hence the system, are *determinate* in that a given set of input samples always results in the same outputs independent of the execution times. Second, the system is *well behaved* in that the data sample rate at any given data edge in the CDFG is independent of the inputs, and the ratio between any two data sample rates will be a statically known rational number. Mathematically, such a synchronous CDFG is equivalent to a continuous function over streams of data samples. Since CDFGs are of course causal, this means that they are equivalent to a function that expresses the i-th set of output samples in terms of the i-th and earlier sets of input samples.

The system state is represented in a CDFG by special delay operator nodes which are initialized to a user specified value. A delay operator node (often referred to as just *delay* or *state* in this paper) delays by one sample the stream of data on its sole input port. A CDFG corresponds to an algorithm for computing the output samples and the new samples to be stored at the delay nodes (i.e. the new state) given the input samples and the old (current) samples at the delay nodes (i.e. the old state). Intuitively, one can think of delay operators as representing registers holding states, and the other operators as combinational logic.

A system[1] is completely represented by a CDFG and the initial values for all the delays in the CDFG. We further restrict ourselves to single-rate systems where the data rate is identical on all the inputs. In the rest of the paper we use just the term CDFG to refer to such a "single-rate synchronous CDFG".

Figure 1 shows an example CDFG with two inputs X and Y, and one output Z. The two delay

1. We use the term "system" interchangeably to mean "the system behavior specification", "an algorithm that realizes the system behavior specification", "an implementation of the system" in different places - the three are not equivalent. A CDFG is a representation of a specific algorithm. There can be multiple algorithms (CDFGs) that express the specified behavior, and there can be multiple implementations (hardware allocation and schedule) of an algorithm (CDFG).

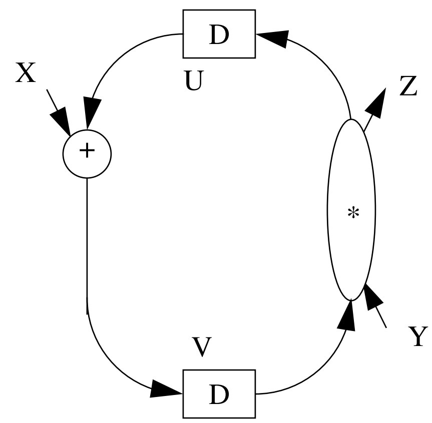

FIGURE 1. An Example CDFG

nodes, U and V, are represented by boxes with the letter 'D'.

2.2 Timing Relationships in a CDFG

Associated with a CDFG are also timing constraints specified by the user. These constraints arise from requirements of the interface to the external world and from performance requirements. Consider a CDFG with P inputs, Q outputs, and R delay nodes (state nodes). Let,

$X[n] = (X_1[n]\ X_2[n] ...\ X_P[n])^T$ be the vector of n-th samples at the P input nodes of the CDFG

$Y[n] = (Y_1[n]\ Y_2[n] ...\ Y_Q[n])^T$ be the vector of n-th samples at the Q output nodes of the CDFG

$S[n] = (S_1[n]\ S_2[n] ...\ S_R[n])^T$ be the vector of n-th samples at the R delay nodes of the CDFG

Given initial values S[0] of the samples in the delay nodes, the CDFG repeatedly computes output samples Y[n] and new samples S[n] for delay nodes from input samples X[n] and old samples S[n-1] at the delay nodes for n = 1, 2, 3...

Various timing parameters are associated with a CDFG as shown in Figure 2. Since we have restricted ourselves to single-rate synchronous CDFGs, the data rates are identical at all nodes so that the inter-sample time interval is identical and constant for all nodes. The maximum rate at which such a CDFG can process samples is called its **Throughput**, and the inverse of this rate, called **Sample Period,** is the minimum required time[2] between successive samples at a node in the CDFG. The sample period, denoted by T_S, is an important timing parameter that is usually constrained not to exceed a maximum value.

2. In high level synthesis and compiler literature time is usually measured in number of control steps.

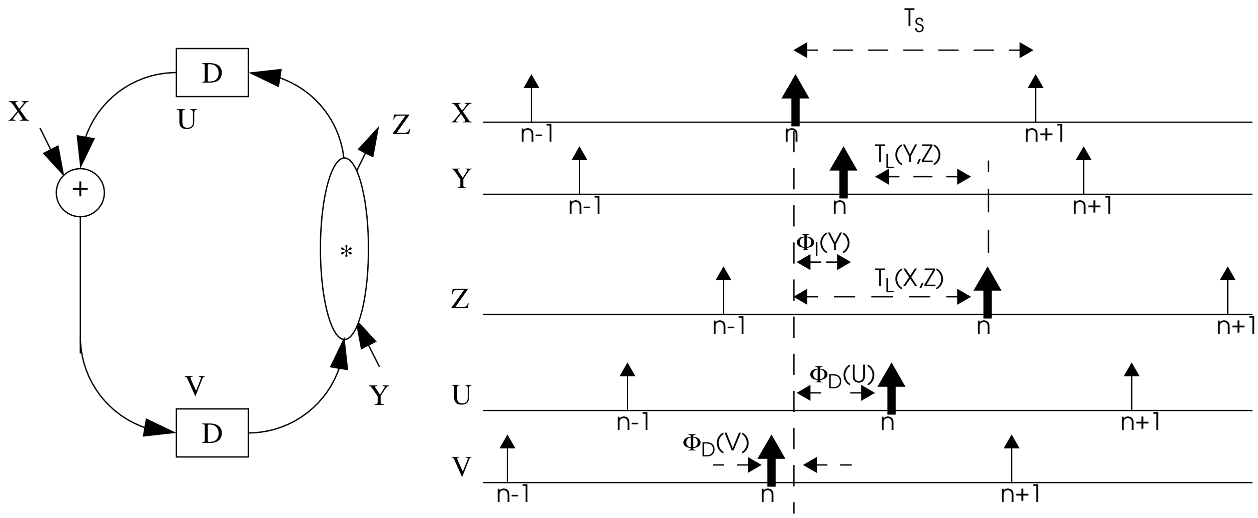

FIGURE 2. Timing Parameters Associated with a CDFG

A second set of timing parameters associated with a CDFG is the set of pairwise latencies. The **Latency**, $T_L(i,j)$, from the i-th input node to the j-th output node is the delay between the arrival of a sample at the i-th input and the production of the corresponding sample at the j-th output. The sample correspondence is defined by the initial CDFG given by the user to specify the system - the n-th sample at an output corresponds to the n-th sample at an input. However, adding or deleting pipeline stages during algorithm transformation changes this correspondence. For example, if one were to add one level of pipelining to the initial CDFG then the latencies will be defined in terms of the (n+1)-th input samples and the n-th output samples. The pairwise latencies $T_L(i,j)$ associated with a CDFG are also important timing parameters because in latency-critical systems they too are constrained not to exceed specified maximum values. Only recently [Sri94] has attention been paid to latency in synthesis of DSP systems.

The relative arrival times of the n-th sample at each of the P input nodes form the third important set of timing parameters for a CDFG. $\Phi_I(i)$ is the skew in the arrival of $X_i[n]$, the n-th sample at the i-th input, relative to the arrival of $X_1[n]$, the n-th sample at the 1-st input. We call $\Phi_I(i)$, which may be negative, the **Phase** associated with the i-th input. The interface to the external world may require that the input phases be constrained to specific values. However many high-level synthesis systems [Rab91] simplify scheduling by assuming that the n-th sample at each input arrives at the same time, i.e. $T_{IA}(i)=0$ for all i=1..P.

It must be noted that the timing parameters T_S, $T_L(i,j)$, and $\Phi_I(i)$, for i=1..P, j=1..Q, are sufficient to completely characterize the timing associated with the inputs and the outputs. In other words, the external timing behavior of a CDFG is completely characterized by them. The values of some of these externally visible timing parameters may be constrained by design requirements.

The final CDFG timing parameter of interest is $\Phi_D(i)$, the skew in the arrival of $S_i[n]$, the n-th sample at the i-th delay node of the CDFG, relative to the arrival of $X_1[n]$, the n-th sample at the 1-st input. We call $\Phi_D(i)$ the **Phase** associated with the i-th delay node. It is clear that $S_i[n]$ is not available earlier than $\Phi_D(i)$ after the arrival of $X_1[n]$, and $S_i[n]$ must be calculated no later than $T_S+\Phi_D(i)$ after the arrival of $X_1[n]$. Note that $\Phi_D(i)$ may be negative. Unlike T_S, $T_L(i,j)$, and $\Phi_I(i)$ which are external timing parameters that may be constrained by the user, $\Phi_D(i)$ is an internal timing parameter that is unconstrained by the user and may be chosen so as to satisfy design constraints while optimizing design cost metrics.

What makes the delay node phases $\Phi_D(i)$ for i=1..R interesting as free timing parameters is that, as shown below, T_S and $T_L(i,j)$ can be expressed in terms of various path lengths in the CDFG, input phases, and the delay node phases. Let:

$P_{ID}(i,j)$ = length of the CDFG path (in control steps) from the i-th input node to the j-th delay node

$P_{IO}(i,j)$ = length of the CDFG path (in control steps) from the i-th input node to the j-th output node

$P_{DD}(i,j)$ = length of the CDFG path (in control steps) from the i-th delay node to the j-th delay node

$P_{DO}(i,j)$ = length of the CDFG path (in control steps) from the i-th delay node to the j-th output node

k = number of pipeline stages that have been added (or removed if k<0) to the initial CDFG that was given by the user as a specification, and from which the current CDFG was obtained after some transformations

A little thought shows that:

$$T_S = \max \{P_{DD}(r,s)+\Phi_D(r)-\Phi_D(s)\ \forall r,s \in 1..R\} \cup \{P_{ID}(p,r)+T_{IA}(p)-\Phi_D(r)\ \forall p \in 1..P,\ \forall r \in 1..R\}$$

$$T_L(i,j) = k*T_S + \max \{P_{IO}(p,j)+\Phi_I(p)-\Phi_I(i)\ \forall p \in 1..P\} \cup \{P_{DO}(r,j)+\Phi_D(r)-\Phi_I(i)\ \forall r \in 1..R\}\ \forall i \in 1..P,\ \forall j \in 1..Q$$

From the above expressions it is clear that the values of T_S, T_L, Φ_I, and Φ_D are coupled - choosing some may place constraints on the achievable values of the remainder. Consequently an algorithm transformation can manipulate those timing parameters that are free. In fact, such a transformation may be combined with other algorithm transformations that effect the CDFG path lengths P_{ID}, P_{IO}, P_{DD}, and P_{DO}.

2.3 The Rephasing Transformation

Our transformation *rephasing* is based on the idea of assigning values to the input and delay node phases that are free so as to improve desired design characteristics. Many VLSI DSP and high level synthesis schedulers assume that all input samples and old delay node samples are available simultaneously at the beginning of a sample period (i.e. all phases are zero), and that the new delay node samples are to be calculated by the end of the sample period. Rephasing plays around with the free phases so as to stagger the computation suitably - hence the name rephasing. Figure 3 introduces the basic idea behind our new transformation. The control-data flow graph (CDFG) shown has two operations: multiplication which takes 3 control cycles and addition which takes 1 control cycle. The CDFG also has two delay nodes: U and V. The delay nodes are annotated by their phases. In addition, the pair of numbers above the box corresponding to a delay node denote the earliest time at which the delay node sample is available and the latest time by which the next delay node sample has to be calculated. The goal is to obtain a minimal area implementation under the sampling rate constraint of 2 cycles. The initial best sampling rate is obviously 3 control cycles. Algebraic and redundancy manipulation transformations are not effective since only one operation is available between delay nodes. Retiming is also not effective - actually it can not be applied at all since no delay nodes can be moved due to the blocking inputs and the blocking output. Even functional pipelining, which overcomes the problem retiming has with the lack of delay nodes on the inputs and the outputs by introducing new delay nodes, is ineffective due to the atomic nature of multiplication operator which requires at least 3 control cycles and cannot be sub-divided using delay nodes.

Amongst previously proposed transformations, only unfolding or unfolding combined with functional pipelining and retiming can resolve the throughput (sampling rate) constraint. However,

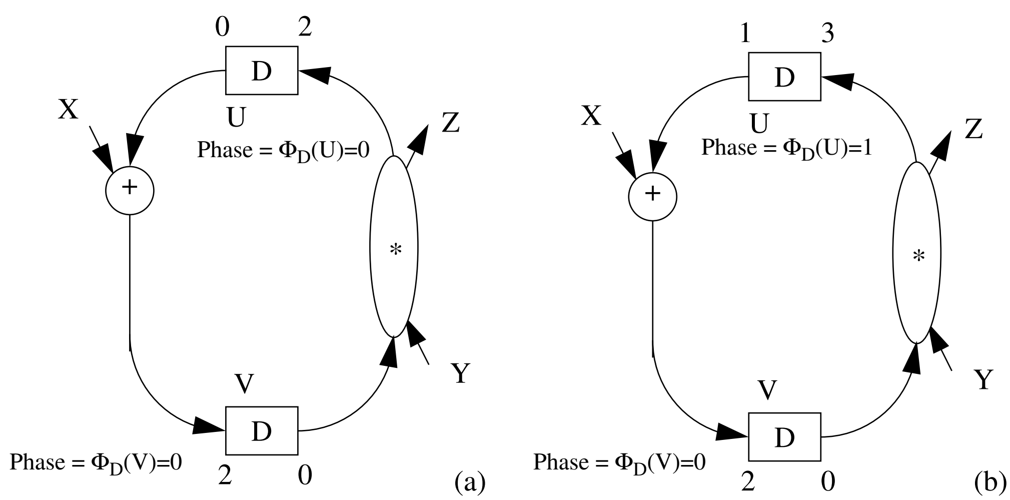

FIGURE 3. Rephasing: The introductory example. The example is also used to illustrate how rephasing can be used to eliminate granularity bottlenecks.

$$Z_1 = U_1 = V_0 * Y_1;$$
$$V_1 = X_1 + U_0;$$

1st iteration

$$Z_2 = U_2 = V_1 * Y_2;$$
$$V_2 = X_2 + U_1;$$

2nd iteration

$$Z_3 = U_3 = V_2 * Y_3;$$
$$V_3 = X_3 + U_2;$$

3rd iteration

FIGURE 4. Functional dependences for the first three iteration of the example shown in Figure

it is well known that unfolding often result in increased hardware requirements due to a need for a significantly more complex controller and the loss of regularity in the structure of the datapath.

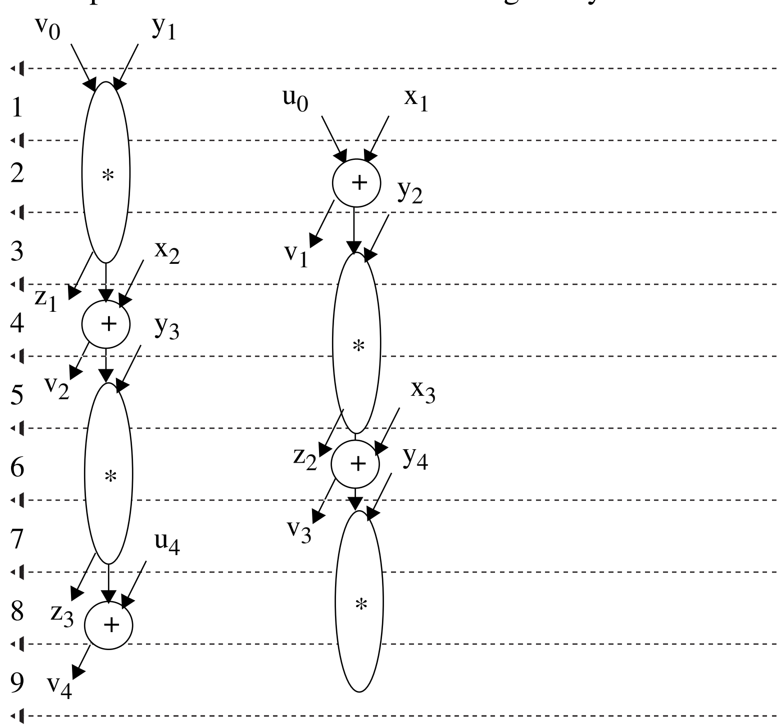

FIGURE 5. Using rephasing to improve throughput by eliminating granularity bottlenecks. Schedule for the example shown in two different representations in Figures 3 and 4.

Now consider the same CDFG, as shown in Figure 3b. The only difference is that the phase, or timing constraints, associated with the delay node U is moved by one control step relative to phase of the delay node V and inputs. The resulting throughput (critical path) is now only two cycles, since the maximal difference between the availability and the required times for each delay node is still two. That everything is correctly done can be checked by analyzing the corresponding functional dependences shown in Figure 4, and from the graphically represented schedule in Figure 5. In the remainder of the paper we will demonstrate systematic methods for exploring this mechanism of phase alteration to improve a number of design metrics.

3.0 Properties of Rephasing

3.1 Feasibility Checking

The first fundamental question about rephasing is the following. Given a CDFG and a set of phase values for each delay node and each input, is the computation well defined i.e., is there a schedule that does not violate timing constrains? Recall that in each cycle of the CDFG sum of differences between the required times and the arrival times of the various delay (state) nodes has to be at least equal to the sum of the computation delays of all the operations.

It is easy to see that this problem is equivalent to the problem of detection of negative cycles in the CDFG. The detection of negative cycles is a well studied problem in the theoretical computer science literature. It can be solved in cubic run time by using the Bellman-Ford or Yen algorithms [see Law76, Section 11, pages 90-91].

3.2 Rephasing vs. Retiming

There is a close and easily seen relationship between rephasing and retiming [Lei91] of a CDFG. There are however several important but less obvious differences between rephasing and retiming. Interestingly almost all of the differences result in giving an edge to rephasing for use in optimizing compiler or as a high level synthesis transformation. The advantages include:

1. There is no need for recomputation of initial states (initial values of delay node samples)

In retiming of a CDFG it is essential that proper initial values be assigned to the delay nodes after the retiming transformation is applied. Not doing so can alter the functional behavior. Unfortunately recomputing the initial values for the delay nodes after retiming is a difficult [Tou93] and sometimes infeasible task, particularly for arbitrary CDFGs not restricted to traditional operators of + and *. Rephasing has an edge over rephasing in this regard because the need for recomputation of initial values of delay nodes is eliminated. Unlike retiming, there is no movement of delay nodes across operator nodes in rephasing.

2. There is no blocking input/output problem and no operator node granularity bottleneck

Retiming is a transformation that relocates delay nodes so that the functionality is preserved [Lei91]. Basic retiming law can be defined in the algebraic framework as the distributivity of the delay node operator over an arbitrary other operator, i.e. the statement D(a) * D (b) is functionally equivalent to D (a * b), where * denotes an arbitrary operator [Pot94]. Leiserson and Saxe were the first to algorithmically study retiming [Lei83, Lei91]. Their best known result with respect to retiming are polynomial time algorithms for the minimization of the critical path and the number of delay nodes. Less known results from their papers include the important theorem that there exist only three retiming bottlenecks which can prevent retiming from achieving an arbitrarily high reduction in the length of the critical path. The bottlenecks are iteration bound, input/output bottlenecks and granularity bottlenecks [Lei91, page 21, Theorem 11].

Iteration bound is the bound imposed by data dependences and the length of computations in cycles. Interestingly, iteration bound was first discovered and discussed in operation research literature under the picturesque name of "tramp steamer problem" [Dan67]. An extensive study of algorithms for efficient calculation of iteration bound was done in theoretical computer science where the problem was most often studied under name of the minimum (cost-to-time) ratio cycle problem [Kar78, Law76]. Currently, the most efficient algorithm is one proposed by Hartmann [Har90]. Reiter [Rei68] was the first to point out the relevance of iteration bound to maximal speed of computation in recursive computations. In early eighties the iteration bound was rediscovered, and thus named, in the VLSI DSP community [Ren81].

Figure 6 illustrates the first two bottlenecks: iteration bound and I/O bottleneck. Assuming that each addition takes one cycle, the iteration bound is equal to 2 (= 6/3) cycles.

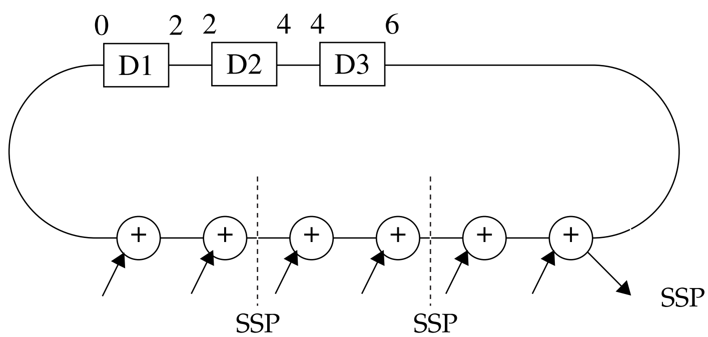

FIGURE 6. Breaking input-output bottleneck using rephasing. While retiming is not unable to reduce critical path of 6 cycles. The application of rephasing reduces the critical path to 2 cycles.

The second bottleneck - the input/output bottleneck - often prevents retiming from achieving the iteration bound. This bottleneck refers to the situation where a delay node can not be moved to a new position that reduces the critical path because the delay node has to be moved across a multi-input operator that does note have a delay node on at least one of its inputs. Figure 6 again provides the illustration. Although the iteration bound is only 2 cycles, input/output bottlenecks dictates the critical path and therefore the best sample period of 6 cycles. Note that the delay nodes cannot be moved across any of the addition nodes using retiming, due to the presence of blocking inputs and blocking outputs.

Leiserson and Saxe proved that if the last two bottlenecks, iteration/bound and granularity, are removed, graph can be always retimed such that iteration bound is achieved [Lei91, page 21]. Potkonjak and Rabaey showed that the addition of pipelining as a preprocessing step to retiming always removes the input/output bottleneck [Pot92].

Figure 3a illustrates the granularity bottleneck faced by retiming (even when supported by pipelining to eliminate any input/output bottleneck), and shows the ability of rephasing to avoid these granularity bottlenecks. Suppose that all input/output bottlenecks are eliminated by pipelining (i.e. by addition of delay nodes on primary inputs). The CDFG has a loop with two operations: one addition and one multiplication. It is assumed that while the addition takes one control step, multiplication takes 3 control steps. This is quite a common scenario in fixed point computations. The loop also has two delay (state) nodes. The iteration bound indicates that a sample period of 2 control steps is feasible. However, it is easy to see that this cannot be done by retiming because achieving iteration bound in this case will require positioning of at least one delay node *inside* the multiplication node. Since high level synthesis assumes that all operations are atomic, and since addition of registers inside execution units is not attractive option from the implementation point of view, retiming can not improve the critical path.

Figure 3b shows the rephased CDFG so that the delay node U is positioned in the first control cycle. Now we see that the resulting CDFG achieves the iteration bound - a new data sample can be accepted every second control step. That it is indeed so is demonstrated by a schedule for the transformed computation in Figure 5. Note that inputs are consumed every two cycles, and the output is delivered every two cycles, while no data dependence is violated.

3. Elimination of the need for transfer units

In many situation it is required to replace delay nodes by transfer units. Transfer units denote that a data value needs to be moved from one register to another without applying any operator to the value. This is done in order to preserve the periodicity of the schedule and module assignment. Transfer units often require additional hardware and always impose additional timing constraints. If two delay nodes are next two each other, and the first delay node sends data only to second delay node, the need for transfer units is eliminated when rephasing is used. For example, we do not need transfer units in the example shown in Figure 6 because the data from delay node D3 can be directly send to delay node D1, with no need for intermediate storing.

4. Preservation of numerical properties

Many design parameters are influenced by the numerical properties of CDFG, and transformations often result in significant changes in the numerical properties. For example, the bit-width is often dictated by numerical stability. Note that bit-width directly influences both the size and the speed of the arithmetic units. In many cases, the application of transformations is avoided solely to prevent their often unpredictable consequences on numerical properties.

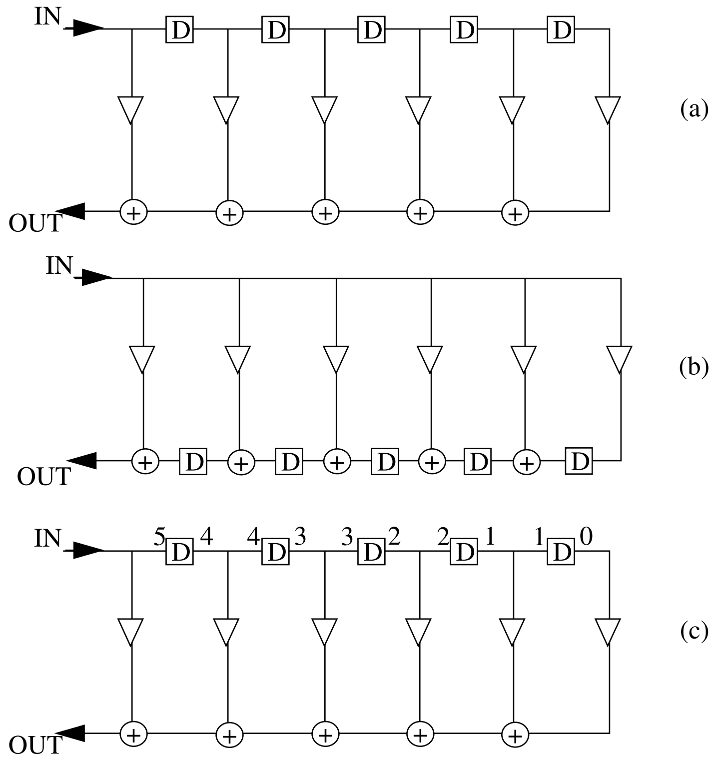

FIGURE 7. FIR filter - Retiming t vs. Rephasing: (a) initial direct form; (b) transposed form obtained after the application of retiming; (c) direct form after rephasing (see Section 2.3)

Although retiming, when applied in isolation, has in some situations relatively mild influence or no influence on bit-width, rephasing, even in comparison to retiming, has superior preservation of numerical properties. In fact, since rephasing is related only to changes in timing relationships between the various parts of a computation, it does not influence numerical properties and the required bit-width at all. When all operations have identical bit-widths, neither retiming nor functional pipelining alter the required bit-width. However, when the bit-width requirements are not uniform, retiming can result in a need for increased bit-width. A typical example is long FIR filter in direct form (Figure 7a), which is often retimed to the transposed form so that the critical path is reduced. Figure 7b shows the graphical illustration of retiming. There is a long chain of additions when high order FIR filters are used, and therefore it is often required that the intermediate results close to output as well as the output be computed using extra bits. While initially all delay node values can be stored using identical number of bits, after retiming there is often a need to allocate significantly more bits for storing the new delay node values. Figure 7c shows the same example rephased, so that while the critical path is reduced to be the same as with retiming, the number of bits needed is the same as that in the original direct form FIR filter.

4.0 Using Rephasing to Optimize Throughput

Throughput is one of the most important design metrics. As we already mentioned, the iteration bound imposes a fundamental limit on the achievable throughput when data-flow transformations (e.g. algebraic and redundancy manipulation) are not used. In this section we demonstrate how rephasing can be used as an effective mean to achieve iteration bound. All results can be directly derived using graph theoretic approaches. However, after a very brief treatment of rephasing using direct graph-theoretic techniques, we will take an alternative, more insightful approach. We will establish a relationship between rephasing and a modified version of functional pipelining, and use this relationship to derive the theoretical results and to design efficient algorithms. The second approach not only simplifies the description of our methods, but also provides a valuable insight into the relationship between the two transformations. This relationship is a basis for the integration of rephasing with other high level synthesis tasks, e.g. scheduling, as presented in Section 5.

The following algorithm in polynomial times applies rephasing on an arbitrary CDFG so that a throughput corresponding to the iteration bound is achieved.

Rephasing for Throughput:

1. *Find the iteration bound using Hartman's minimum ratio cycle algorithm;*
2. *Assign to each delay node a delay equal to the negative of the iteration bound (for example, if iteration bound is k, assign delay equal to -k);*
3. *Using the Bellman-Ford algorithm [Law76] find the longest path between one node and all other nodes. The node is arbitrarily randomly selected; (We assume that the CDFG has only one nontrivial strongly connected component (SCC). If the CDFG has more than one SCC the rephasing for throughput algorithm should be applied in each nontrivial SCC separately. SCC can be identified using stand graph-theoretic approach [Cor90].)*
4. *Assign to the selected node phase 0. Assign to each delay node the phase which equal to distance from the selected node. Assign to each input a phase that is equal to some large negative number M. M has a magnitude larger than the sum of delays of all operations nodes in the CDFG.*

For sake of brevity we omit the proof. The proof can be also easily established using the functional pipelining - rephasing relationship theorem presented bellow. The run time is dominated by the Bellman-Ford algorithm which is $O(n^3 \log n)$.

The relationship between function pipelining and rephasing is established by the following theorem:

Functional Pipelining-Rephasing Relationship Theorem: *Rephasing on a given CDFG G1* *is equivalent to functional pipelining on a modified CDFG G2. G2 is obtained from G1 by replacing each node with k multicycle delay by a chain of k nodes with unit cycle delays.*

The proof is constructive, and therefore can be directly used for conversion between functional pipelining and rephasing. Suppose that the CDFG G2 is given. To find phases of all delay nodes, all what is needed is to run the Bellman-Ford algorithm on the CDFG. The distances from the delay nodes in G2 of edges which correspond to delay nodes in G1 are the required phases. This is so because we will not change any timing constraint if we assign these phases to the delay nodes. The second part of the proof, going from G1 to G2, follows similarly.

The following theorem from [Lei91, page 21, Theorem 11], which has been slightly modified for the needs of high level synthesis, is the basis for an alternative view of the algorithm for rephasing to optimize the throughput:

Let G be a unit-delay CDFG, and let c be any positive integer. Then there is a retiming r of G such that critical path of CDFG is at most c if and only if G-1/c contains no cycles having negative weight.

G-1/c is, by definition, the CDFG where each delay node has delay of (-c). The new rephasing algorithm for throughput optimization can be now formulated using the following pseudo-code:

Throughput Optimization using Rephasing

1. *Substitute a given CDF G1 with CDFG G2, where all operation with k cycles delay, are replaced with k operation of one cycle delay;*
2. *Apply the modified Leiserson-Saxe functional pipelining algorithm on G2;*
3. *Find timing of all nodes in the modified CDFG using the Bellman-Ford algorithm;*
4. *Find the corresponding phase for all delay nodes and inputs.*

The effectiveness of the algorithm for throughput optimization using rephasing is discussed in Section 7.

5.0 Optimizing Area, Power, Transient and Permanent Fault Tolerance Overhead and Testability Overhead Using Rephasing

5.1 Optimizing Area using Rephasing

It has recently been demonstrated that retiming at behavioral level can be effectively used to improve resource utilization of the targeted design, and therefore reduce the implementation area {Pot94]. However, the effectiveness of retiming for area optimization is often severely limited by granularity bottlenecks and input/output bottlenecks. Figure 8a shows a typical example of this situation. It is assumed that the available time is four control steps. Retiming can not be applied on the initial CDFG because any movement of the delay nodes is prevented by either the output or the inputs. Since all operations, except addition +1, are on the critical path, it is obvious that implementation requires at least 4 multipliers, 2 adders and 1 subtracter.

The application of rephasing significantly improves the resource utilization. If delay node D1 is shifted in time by 3 cycles, as shown by Figure 8b, it is easy to generate schedule and assignment under significantly lower allocation constraints. Table 1 shows one of the possible resulting schedules - only 1 execution of each type is now required.

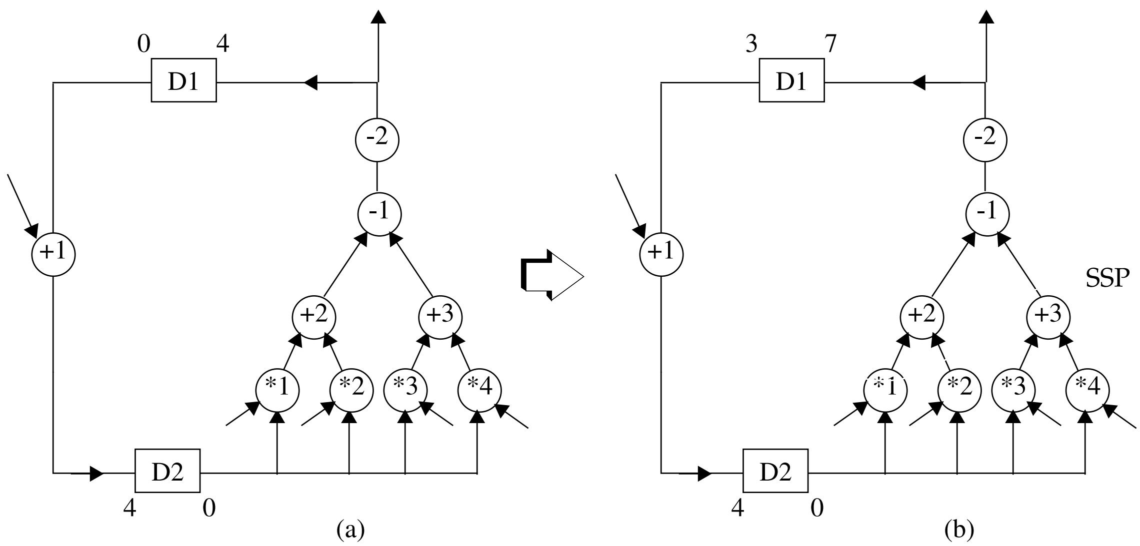

FIGURE 8. Area minimization using rephasing. While for the initial CDFG 4 multipliers, 2 adders, and 1 subtracter is required, for the transformed CDFG only 1 multiplier, 1 adders and 1 subtracter are sufficient. The corresponding schedules are shown in Table 1.

It has also been recognized that the constraints imposed only by inputs and output positioning can be overcome by the use of pipelining [Pot92]. However, there are at least two type of designs

	multiplier	adder	subtracter
1. control step	*3	+2	
2. control step	*4	+1	-1
3. control step	*1	+3	-2
4. control step	*2		

Table 1: A Possible Schedule for the Rephased Example in Figure 8b

where rephasing has significant advantage not just over retiming, but also over functional pipelining. The first situation is related to the ability of rephasing to reduce the number of transfer units.The second, and more important, situation is related to the potential of rephasing to efficiently handle granularity bottlenecks. Note that functional pipelining can not remove granularity bottlenecks.

An efficient and effective approach for area optimization requires two components: an objective function to estimate (predict) the area without the time consuming application of scheduling, assignment, and physical design tools; and an optimization algorithm for minimizing the objective function.

A simple option for solving these two goals was to adapt the objective function and the optimization algorithm used in the high level synthesis system Hyper for area optimization using retiming and functional pipelining [Pot94]. While the objective function can be directly transferred, only minimal changes are needed to adapt the probabilistic sampling optimization algorithm. It is sufficient to change the definition of the basic move from an atomic retiming move to rephasing a delay node by one cycle. This approach guarantees that at least as good result as reported for retiming [Pot94] functional pipelining [Pot92] using Hyper will be always produced.

However, recent developments in high level synthesis indicate that an even more powerful optimization algorithm, and a more accurate objective function can be developed without penalizing the run time.

The new objective function for area optimization is calculated using the following approach.

For each operation *i* with as-soon-as-possible time $ASAP_i$, as late as possible time $ALAP_i$, and the length $\mathrm{Duration}_i$ we define the function

$u_i(k) = \frac{1}{ALAP_i - ASAP_i + 1}$ for each time slot $t \le k < t + \mathrm{Duration}_i$. Next, we calculate the values *U(t)* of the function that describes the likelihood that some operation of the specified type is scheduled in a given time step. This is done by accumulating the contributions of all operations of the specified hardware type:

$$U(t) = \sum_{i=1}^{\mathrm{NumOperations}} u_i(t) \qquad (1)$$

Finally, as an estimation of the requirements for execution units of a specific type, we take the maximum of the function U(t) over all control steps for the corresponding type of operations. Similarly, by considering each type of transfer of data from a particular type of execution unit *i* to a particular execution unit of type *j,* we calculate a function indicating the need for interconnect of type *ij*. We estimate the lifetime of each variable by assuming that it is alive from the most likely control step that its producing operation is scheduled to the expected moment when the latest operation which consume the variable is scheduled. Expected times for both operations are calculated as the middle of the ASAP-ALAP interval. The objective function (OF) is the weighted sum of three components: execution units, interconnect, and registers. The weights are proportional to the cost of each component.

We compared the quality of the new function with the one used in the Hyper system using a benchmark set of DSP, image, video, control, information theory and communication applications. The statistical analysis indicates that the new measure consistently outperforms the objective function used by Hyper during retiming [Pot94]. For example, Hyper uses the number of delay (state) nodes as a predictor of the number of registers. This approach has a correlation of 88% to the number of registers in the final implementation, while our new register prediction function achieved correlation of 97% on a set of 40 real-life examples that we used to validate our approach. Similarly, on the same set of examples, correlations of 93% and 92% were found for the number of execution units and interconnect respectively when the new objective function is used.

The new optimization algorithm is based on the force-directed paradigm [Pau89] where at each step the most critical part of the predicted cost is targeted. The following pseudo-code describes the optimization algorithm:

Area Optimization using Rephasing:

while (it is 1st pass, or there was improvement in the previous iteration) {

find objective function (OF) for each one clock step change of the phase for each delay node and input;

find the best OF, and the corresponding one clock steps change of phase;

apply the best selected one clock step rephasing;

}

An important addition that we made to the force-directed optimization is the use of min-bounds [Rab91]. As soon as the requirement for some type of unit reaches the min-bound, further improvements in value of this component are not considered as the improvements of the objective function OF, since the minimum along this dimension has been reached.

Finally, it is important to note that the effectiveness of rephasing for area, power, and fault tolerance optimization can be easily combined with the power of algebraic transformations (e.g. commutativity, associativity, distributivity and the inverse element low) and redundancy manipulation transformations (e.g. common subexpression elimination and replications) by simply combining basic alternations of the CDFG as the set of moves, as it was done in [Pot94].

5.2 Optimizing Power using Rephasing

Although area and throughput are still widely used as the primary optimization objectives during high level synthesis, the importance of some other design metrics has also recently been recognized. For example, power optimization has become important [Cha92] due to the increasing desire for portability and mobility. In this subsection we present a rephasing algorithm for power optimization under throughput constraint. This set of goals and constraints is typical in communications, video, and image applications.

Power minimization at the behavioral level can to the first order of approximation be treated as a simultaneous optimization of throughput, area and number of operations [Cha92]. The extensive experimental studies confirm the validity of this approximation [Cha92]. The objective function for power prediction is more involved than the one for area due to a need for taking into account the energy consumption not only in the data path components but also in the control logic. We used the behavioral level power prediction tools [Cha92] in Hyper for this task.

Rephasing does not alter the number of operations, with the only exception being that it can reduce the number of transfer operations that are needed. This change is usually relatively small. Extensive experimental studies as well as theoretical analysis indicate that power vs. throughput function has unimodal shape, i.e. there exist a single minimum [Cha92].

Based on this observation we developed an algorithm for power minimization using rephasing. The algorithm conducts a search along the throughput axis. At each step the area, and therefore the effective capacitance, is minimized by applying the algorithm for area minimization using rephasing.

5.3 Optimizing Testability Overhead using Rephasing

Another design metrics which has recently gained popularity is testability. For example, a number of high level synthesis system algorithms has been proposed that simultaneously target both area and the partial scan overhead [Lee93] In this subsection we present the key idea on how rephasing can be used to reduce the hardware overhead when synthesis for testability using partial scan and resource utilization are targeted.

It is known that presence of loops in a sequential circuit makes sequential ATPG very hard. Partial scan is an effective technique to break loops by scanning a subset of flip-flops/registers [Che90]. It has been demonstrated that it is possible to generate designs that do not compromise resource utilization while significantly reducing partial scan overhead by paying early attention to formation of loops in the datapath.

Consider the example shown in Figure 9 with an available time of 2 control steps. Therefore

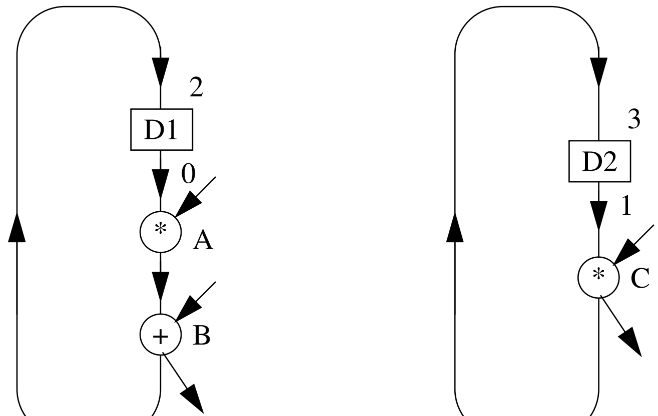

FIGURE 9. Reducing partial scan overhead using rephasing. If both delay nodes are phased identically, at least two scan registers are needed in the register file model to eliminate all loops in the datapath (Section 4.4). However, after rephasing delay node D2 the lifetime of variable *f* is reduced so that it does not overlap with the lifetime of variable *g*. Now two variables can share the same scan register.

all operations are on the critical path. For each operation, we indicate the control step in which the operation is bound to be scheduled. The example has two CDFG loops. The dedicated register file is assumed [Rab91]. It is easy to see that the design requires at least two scan registers in order to ensure that all CDFG loops are broken, regardless of the scheduling and assignment. This is so because it is impossible to share the same scan register to break both loops. However, after rephasing of the right loop as shown in Figure 9 the lifetime of variable D2 in the register file of the multiplier is reduced. Now the variables D1 and D2 can be stored in the same scan register while breaking both the CDFG loops. Therefore, the partial scan overhead is reduced

The algorithm for the simultaneous optimization of area and partial scan overhead using rephasing can be easily derived by modifying the algorithm for the same task which explored the addition of deflection operation [Pot94]. However, the algorithm is involved, with a number of technical details, and is beyond the scope of this paper.

5.4 Optimizing Transient and Permanent Fault Tolerance overhead using Rephasing

With the increased level of integration it is, interestingly, easy to reduce hardware overhead when either transient or permanent fault tolerance is targeted. We will illustrate the basic ideas behind two approaches using the examples shown in Figures 10 and 11.

Figure 10 left of the dotted line shows the initial design. When the duplication [Kar92] is used as the transient fault tolerant technique, it is required to compute two copies of the initial computation, i.e. to use a CDFG that has two identical components as shown in Figure 10. Of course, the

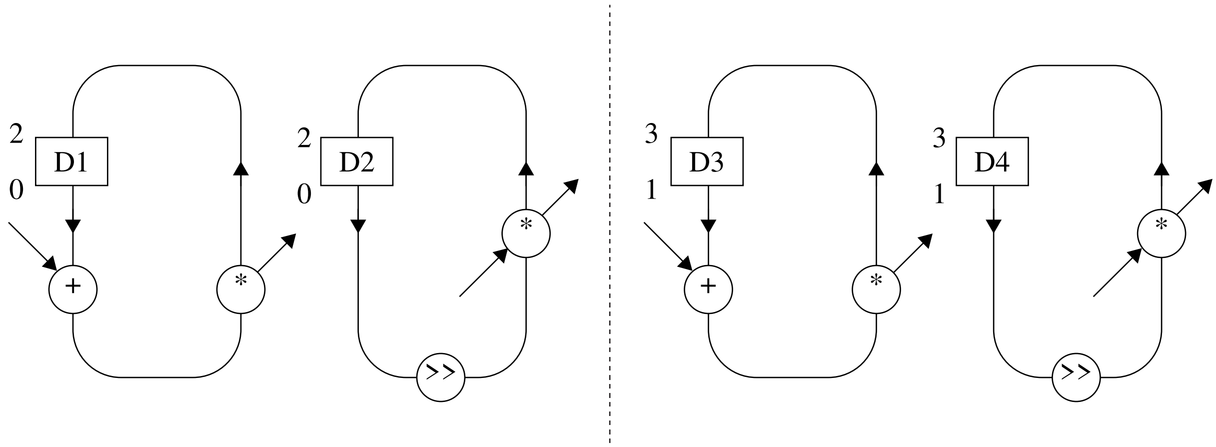

FIGURE 10. Reducing transient fault tolerance overhead using rephasing. As fault-tolerance technique 2-redundancy replication is used.

goal during duplication is to reduce area overhead while preserving throughput.

It is evident that the optimization of the fault tolerance overhead is equivalent to area minimization when both the initial and the replicated CDFG are simultaneously considered for resource allocation, scheduling, and assignment. So, the force directed rephasing algorithm and structural properties-based objective function directly address the problem of minimization the overhead of the duplication-based fault tolerance. The algorithm in only two steps yield the solution shown in Figure 10. It is assumed that each operation takes one control cycle, and that the available time is 2 control cycles. If no fault tolerance is requested the initially 2 multipliers, 1 adders, and 1 shifter are needed.

When duplication, without use of transformations, is applied the minimal hardware resources increase to 4 multipliers, 2 adders and 2 shifters. After the application of rephasing, again only 2 multipliers, 1 adder, and 1 shifter are needed. So, we see that in this case fault tolerance is achieved with no increase in hardware.

The observation that the optimization of the fault tolerance overhead is equivalent to area minimization when both the initial and the replicated CDFG are simultaneously considered, enables the direct application of the algorithm for area optimization using rephasing. The only modification is the inclusion of CDFG duplication as a preprocessing step.

An efficient BISR (Build-In-Self-Repair) fault-tolerance technique based on exploiting the flexibility provided by design space exploration has been proposed recently [Gue93]. BISR is a widely used technique for yield and permanent fault tolerance enhancement where in addition to operational modules, a set of spare module is also provided. If a faulty module is detected, it is replaced with a spare module. The main idea in the high level synthesis BISR technique is to reduce overhead by using different schedules or differently transformed CDFGs for the various scenarios of failed units. Therefore, in some sense, the same backup module enables replacement of several types of failed units and reduces BISR overhead.

Figure 11 and Table 2 show an example of how rephasing can be used to reduce the BISR

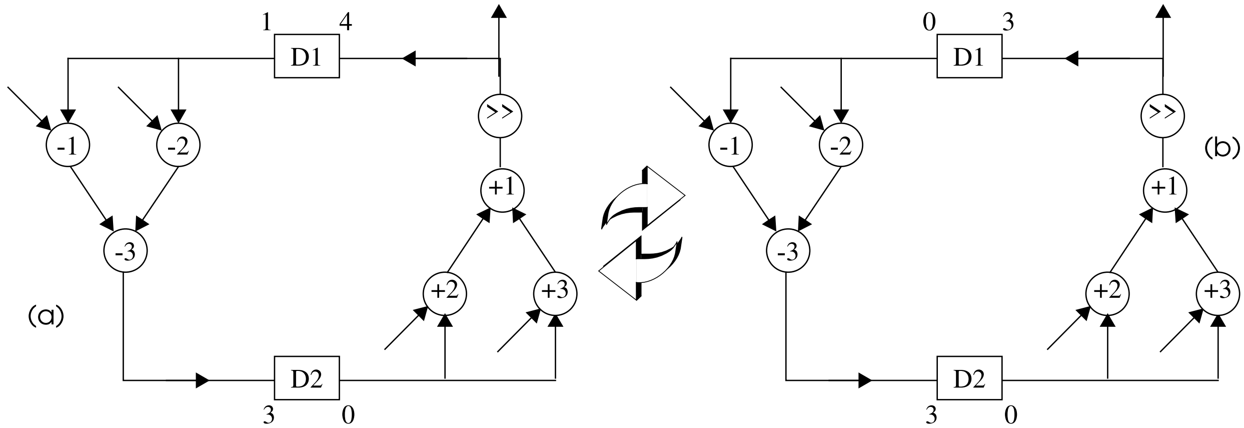

FIGURE 11. Reducing BISR (permanent fault tolerance Build-in-Self-Repair) overhead using rephasing. The transformed CDFG shown in Figure (a) is used when one of the adders is faulty. The transformed CDFG shown in Figure (b) is used when one of the subtracters is faulty.

overhead. It is assumed that BISR scheme targets resilience against a single fault at any type of execution unit. It is easy to see that standard (non-BISR) requires 1 shifter, 2 adders, and 1 subtracter. If rephasing is not used, BISR scheme requires one spare unit for each type of operations. However, if rephasing is applied as shown in Figure 11 the overhead can be reduce, so that 2 shifters, 2 adders, and 2 subtracters are sufficient regardless of which unit is faulty. Note that the existence of blocking inputs and the output prevents use of retiming for this type of optimization.

Failed Unit	Adder			Subtracter			Shifter		
Control Step	>>	-	+	>>	-	+	>>	-	+
1	>>		+2		-1	+2, +3		-1,-2	+2,+3
2		-1, -2	+3		-2	+1		-3	+1
3		-3	+1	>>	-3		>>		

Table 2: Potential Schedules for Example of Fig. 10

For the BISR overhead minimization using rephasing the retiming BISR algorithm from the Hyper high level synthesis system can be used. The only difference is the change of the basic move, from local retiming to local rephasing. This analogy is already explained in Subsection 4.1.

6.0 Relationship with other High Level Synthesis Tasks

6.1 Scheduling

Besides conceptual simplicity perhaps the most influential factor behind the current widespread popularity of assigning the same phase to all states and inputs is that this approach results in very straightforward timing constraints for scheduling. The scheduling, in this approach, starts from the first control cycle assuming that all primary inputs and all states are available, and finishes at the last control step of the iteration, at which time all outputs and all states must be computed.

Without a careful analysis one may think that rephasing complicates the scheduling by enforcing more suitable but also more complicated sets of timing constraints on states and the inputs. However, using the functional pipelining-rephasing relationship theorem, it is easy to derive the same uniform timing constraints on scheduling after the application of arbitrary rephasing. All what is needed is to find for a given rephasing, the corresponding functional pipelining on the corresponding modified CDFG (see Section 3.0), and denote which operation nodes in the CDFG are now aligned at the beginning (and therefore also at the end) of the iteration period. On Figures 5 and 7 those nodes are marked by the SSP cuts (scheduling starting points).

Another simple, but useful observation is that when the available time is T, then it is convenient to think about the schedule of a node A in a control step S in the following way. If S is smaller than T, then use the standard reasoning about scheduling. If S is greater than T then one should take S modulo T for the scheduling step as the correct interpretation of the selected scheduling step after the application of rephasing.

6.2 Relationship to Other Transformations or How to Enable Rephasing

It is well known that organized application of several transformations is usually superior to the application of isolated transformations. Recently, a powerful principle of enabling has been established. The enabling principle establishes the transitive ordering relationship between transformations in a such way that if a particular transformation has higher precedence, it is always more beneficial that this transformations is applied first. For example, it has been demonstrated that common subexpression replication can be used to significantly improve performances of algebraic transformations. Deriving the position of rephasing in the transformation ordering is simple. It has the exactly the same position as retiming and functional pipelining. This observation, enables an easy approach for development of several transformation ordering scripts for efficient use of rephasing.

For example, the retiming enabling algorithms such as ERB [Iqb93], can be easily modified to rephasing enabling algorithm. The only difference is that instead of constraints on how much a particular state can be retimed, now we have the corresponding constraints on how much the state can rephased. In both cases, satisfaction of constraints guarantees the reduction of the iteration bound, and therefore improvement of the throughput.

6.3 Software Rephasing

The importance of merging compiler and high level synthesis techniques has been emphasized in some recent work [Mar93a, Mar93b]. Software pipelining is a popular compiler transformations [Lam88, Goo89, Lee92] that integrates scheduling and functional pipelining for optimization of control loops. It is widely used in compilers for throughput optimization under resource constraints [Lam88]. Rephasing targets time loop. If the target is changed to control loop, a new transformation, software rephasing is developed. There is a significant level of similarity and conceptual equivalence between software pipelining and software rephasing - they are like different views of the same object. However an advantage of the rephasing approach is that it, unlike software pipelining, keeps scheduling decoupled from transformation. Therefore rephasing can be easily combined with other transformations such as unfolding and retiming, and more importantly, the result of the sequence of transformations can be analyzed independent of the scheduling.

7.0 Experimental Results

Throughput optimization using rephasing is a problem of polynomial complexity. We presented the optimization algorithm in Section 3.0. Therefore, the relevant question is not how well the algorithm is performing, but how much improvement can be achieved in the length of the critical path by using rephasing. To analyze the improvements we analyzed 55 DSP, video, and telecommunications examples. The initial critical paths ranged from 3 to 380 control cycles. The average critical path was 67 control cycles, while the median was 14. After rephasing the shortest critical path was 1 cycles long, and longest only 17 control cycles long. The average critical path was long 3.6 cycles, while the median critical path was long only 2 control cycles. So the median improvement was by factor 7, while the average was improved by more than 20 times.

Table 3 shows the performance of rephasing for area optimization on the set of 10 examples. The examples are: 8X8 DCT- 2-dimensional discrete cosine transform, 9 FWT - 9 point Winograd fast Fourier transform (FFT), 11 FWT - 11 point Winograd FFT, 11FIR - 11th order FIR filter, 7IIR - 7th order cascade IIR filter and 11IIR - 11th order cascade IIR filters, Volterra - 2nd order Volterra filter, DAC- digital-to-analog converter, lin5 - 5th order linear controller, 2D FIR - two dimensional 10th order FIR image filter. The average reduction of area was by 31.3%, the median reduction of area was by 35.5%. The application of rephasing on the 5th order elliptical wave digital filter, resulted in the implementation which requires only 2 multipliers and 2 adders for the available time of 17 control cycles. It is best known implementation, where excellent numerical properties of the filter are fully preserved.

Design	Initial Area [mm^2]	Final Area [mm^2]	Final/Initial [%]
8X8 DCT	40.46	22.19	54.8
9FWT	51.72	33.62	65.0
11WFT	54.08	25.94	66.5
11FIR	7.67	4.92	64.1
7IIR	18.27	16.25	88.9
Volterra	34.42	20.48	59.5
Lin5	37.71	30.45	80.7
DAC	46.66	27.98	60.0
2D FIR	39.98	20.06	50.,2
11 IIR	22.65	22.07	97.4

Table 3: Area Optimization Using Rephasing

Table 4 shows the performance of rephasing for power optimization on the set of 10 examples. The average reduction of power was by 4.18 times, the median reduction of power was by 4.12 times.

Design	Initial Power [nJ/sample]	Final Power [nJ/sample]	Final/Initial
8X8 DCT	284.42	49.27	5.77
9FWT	76.52	18.32	4.18
11WFT	80.02	20.00	4.00
11FIR	21.17	5.21	4.06
7IIR	66.20	34.27	1.93
Volterra	81.77	40.06	2.04
Lin5	39.91	9.21	4.33
DAC	48.02	7.52	6.39
2D FIR	39.56	5.88	6.73
11 IIR	78.45	33.01	2.38

Table 4: Power Optimization Using Rephasing

Table 5 shows the performance of rephasing for transient fault tolerance optimization on the set of 5 examples. The average overhead was by 20%, the median overhead of area was by 14%.

Design	Initial Area [mm^2]	Overhead [%]
8X8 DCT	40.46	20
9FWT	51.72	11
11WFT	54.08	9
11FIR	7.67	14
7IIR	18.27	47

Table 5: Overhead in duplication based fault-tolerance after the application of rephasing

The nominal overhead was duplication, and fault tolerance is 100%.

Tables 5 and 6 show the performance of rephasing for area overhead optimization in BISR fault tolerance scheme on the set of five examples. For the yield and productivity calculations, we followed the state-of the-art procedure presented in [Sta92]. The initial yield is set to 10%, as was assumed in Stapper's paper. Similar data was also used in [Pat89]. We calculated changes in both

Example	IU	FU	NT	Non-BISR Area	BISR Area	Area Overhead (%)
9WFT	9	12	4	33.62	39.37	17.1
11FIR	9	11	4	4.92	5.53	12.3
7 IIR	17	19	4	16.25	17.54	7.9
DAC	30	32	4	27.98	29.51	5.5
LIN4	21	23	3	36.00	38.49	6.9

Table 6: Physical characteristics of examples used during validation of rephasing for BISR: IU - # of EXU units in non-BISR implementation; FU - # of EXU units in BISR implementation; NT - # of hardware classes

yield and productivity, for various values of the variability parameter μ. This parameter gives an indication of the assumed probability of clustered defects, which are the most common sources of chip malfunctions. Large values of μ correspond to smaller levels of clustering, and therefore lower processing variability [Sta92]. The average yield improvement were between 48% and 169% depending on the variability factor.The median yield improvement were between 49% and 154% depending on the variability factor. The average productivity improvement were between 35% and 147%

Example	Yield					Productivity				
	μ=0.5	μ=1.0	μ=2.0	μ=5.0	μ=Inf	μ=0.5	μ=1.0	μ=2.0	μ=5.0	μ=Inf
9WFT	14.40	15.39	16.47	17.95	20.90	1.230	1.314	1.406	1.533	1.785
11FIR	15.34	16.67	18.19	20.37	25.35	1.366	1.484	1.619	1.813	2.257
7IIR	15.07	16.32	17.82	20.24	28.65	1.397	1.513	1.652	1.876	2.655
DAC	14.54	15.76	17.08	19.30	30.50	1.378	1.494	1.619	1.829	2.891
LIN4	14.92	16.13	17.56	19.93	29.49	1.396	1.509	1.643	1.864	2.759

Table 7: Yield (in%) and Relative Productivity change due to use of rephasing for BISR for 7 examples from Table 2 for various values of the variability parameter μ. The initial yield is 10%.

depending on the variability factor.The median productivity improvement were between 38% and 166% depending on the variability factor.

8.0 Conclusion

We introduced a new type of transformation - *rephasing*. Rephasing changes the timing constraints associated with delay nodes, inputs, and outputs, and can be used to address a variety of different design goals. We demonstrated the effectiveness of rephasing in optimizing a number of design metrics: throughput, area, power, permanent and transient fault tolerance, and testability.

9.0 References

[And67] C. Anderson, J. Earle, R. Glodschmidt, D. Powers: “The IBM System/360 Model 91 Floating Point Execution Unit”, IBM Journal of Research and Development, Vol. 11, No. 1, pp. 34-53, 1967.

[Ber93] R.A. Bergamaschi, A. Kuehlmann: “A System for Production Use of High-Level Synthesis”, IEEE Trans. on VLSI Systems, Vol. 1, No. 3, pp. 233-243, 1993.

[Bha94] S. Bhattacharya, S. Dey, F. Brglez: “Performance Analysis and Optimization of Schedules for Conditional and Loop-Intensive Specifications”, 31st ACM/IEEE Design Automation Conference, pp. 491-496, 1994.

[Cha92] A.P. Chandrakasan, M. Potkonjak, J. Rabaey, R. Brodersen: “Hyper-LP: A Design System for Power Minimization using Architectural Transformations”, ICCAD, 300-303, 1992

[Che90] K.T. Cheng, V.D. Agarwal: “A Partial Scan Method for Sequential Circuits with Feedback”, IEEE Trans. on Computers, Vol. 39, No. 4, pp. 544-548, 1990.

[Cor90] T.H. Cormen, C.E. Leiserson, R.L. Rivest: “Introduction to Algorithms”, The MIT Press, Cambridge, MA, 1990.

[Cot65] L.W. Cotten: “Circuits Implementation of high-speed pipeline systems, 1965 AFIPS Proceedings of Fall Joint Conference, Vol. 27, pp. 489-504, 1965.

[Dan67] G.B. Danzig, W. Blattner, M.R. Rao: “Finding a cycle in a graph with minimum cost to time ratio with application to a ship routing problem”, Theory of Graphs, ed. by P. Rosenstiehl, pp. 77-84, Dunod, paris and Gordon and Breach, New York, NY, 1967.

[Fil93] D. Filo, D.C. Ku, C.N. Coehlo, G. De Micheli: “Interface Optimization for Concurrent Systems Under Timing Constraints”, IEEE Trans. on VLSI Systems, Vol. 1, No. 3, pp. 268-281, 1993.

[Fis88] C.N. Fischer, R.J. Le Blank: “Crafting a Compiler”, The Benjamin/Cummings Publishing Co., Menlo Park, CA, 1988

[Gla85] L.A. Glasser, D.W. Dobberpuhl: “The Design and Analysis of VLSI Circuits”, Addison-Wesley, Reading, MA, 1985.

[Goo89] G. Goossens, J. Wandewalle, H. DeMan: “Loop Optimization in register-transfer scheduling for DSP-systems”, DAC-89, pp. 826-831, 1989.

[Gra93] C.T. Gray, W. Liu, R.K. Cavin: “Wave Pipelining: Theory and CMOS implementation”, Kluwer, Boston, MA, 1993.

[Gue93] L. Guerra et al. “High Level Synthesis for Reconfigurable Datapath Synthesis”, ICCAD-93, pp. 26-29, 1993.

[Har90] M. Hartman: “On cycle means and cycle staffing”, Technical Report UNC/OR/TR-90/14, University of North Carolina at Chappel Hill, June 1990.

[Hen89] J.L. Hennessy, D.A. Patterson: “Computer Architecture: A Quantitative Approach”, Morgan Kaufmann Publishers, San Mateo, CA, 1989.

[Ish90] A.T. Ishii, C.E. Leiserson: “A Timing Analysis of Level-Clocked Circuitry”, Advance Research in VLSI: Proceedings of the Sixth MIT Conference, pp. 113-130, 1990.

[Iqb93] Z. Iqbal, M. Potkonjak, S. Dey, A. Parker: “Critical Path Minimization Using Retiming and Algebraic Speed-Up”, pp. 573-577, 30th ACM/IEEE DAC, 1993.

[Kar78] R. Karp: “A characterization of the minimum cycle mean in a digraph”, Discrete Mathematics, Vol. 23, pp. 309-311, 1978.

[Kar92] R. Karri and A. Orailoglu, “Transformation-Based High-Level Synthesis of Fault-Tolerant ASICs,” 29th ACM/IEEE Design Automation Conference, pp. 662-665, 1992.

[Ku92] D. Ku, G. De Michelli: “High Level Synthesis of ASICs Under Timing and Synchronization Constraints”, Kluwer, Norwell, MA, 1992.

[Lam88] M. Lam: “Software Pipelining: An Effective Scheduling Technique for VLIW Machines”, SIGPLAN’88 Conf. on Programming Language Design and Implementation, pp. 318-328, 1988.

[Lam92] W.C.K. Lam, R.K. Brayton, A.L. Sangiovanni-Vincentelli: “Valid Clocking in Wavepipelined Circuits”, ICCAD, pp. 518-525, 1992.

[Law76] E.L. Lawler: “Combinatorial Optimization: Networks and Matroids”, Holt, Rinehart and Winston, New York, NY, 1976.

[Lee92] T-F. Lee, A.C-H. Wu, D.D. Gajski, Y-L. Lin: “An Effective Methodology for Functional Pipelining”, ICCAD-92, pp. 230-233, 1992.

[Lee93] T. C. Lee, N.K. Jha, W.H. Wolf: “Behavioral Synthesis of Highly Testable Data Path under Non-Scan and Partial Scan Environments”, DAC93, pp. 616-619, 1993.

[Lei83] C.E. Leiserson, F.M. Rose, J.B. Saxe: “Optimizing synchronous circuitry by retiming”, 3rd Caltech Conference on VLSI, pp. 87-116, 1983.

[Lei91] C.E. Leiserson, J.B. Saxe: “Retiming Synchronous Circuitry”, Algorithmica, Vol. 6, No. 1, pp. 3-36, 1991.

[Lin88] Q. Lin, P. Xia: “The design and implementation of very fast experimental pipelining computer”, Journal of Computer Science and Technology, Vol. 3, No. 1, 1988.

[Mar93a] P. Marwedel: “Cooperation of Synthesis, Retargatable Code Generation, and Test Generation in the MSS”, European Design and Test Conference, pp. 63-69, 1993.

[Mar93b] P. Marwedel: “Tree-Based Mapping of Algorithms to Predefined Structures” IEEE ICCAD-93, pp. 586-593, 1993.

[MvF88] M.C. McFarland, A.C. Parker, R. Camposano: “Tutorial on High-Level Synthesis”, 25th Design Automation Conference, pp. 330-336, 1988.

[Noi82] D. Noice, R. Matheus, J. Newkirk: “A Clocking Discipline for Two-Phase Digital Systems”, IEEE International Conference on Circuits and Computers, pp. 108-111, 1982.

[Not91] S. Note, W. Geurts, F. Catthoor, H. De Man: “Cathedral-III” Architecture-Driven High Level Synthesis for High Throughput DSP Applications”, Design Automation Conference, pp. 597-602, 1991.

[Pau89] P.G. Paulin, J.P. Knight: “Scheduling and Binding Algorithms for High-Level Synthesis”, DAC-89, pp. 1-6, 1989.

[Pot92] M. Potkonjak, J. Rabaey: “Pipelining: Just Another Transformation”, Proceedings of the International Conference on Application Specific Array Processors, pp. 163-177, 1992.

[Pot94] M. Potkonjak, J. Rabaey: “Optimizing Resource Utilization Using Transformations” IEEE Transactions on CAD, Vol. 13, No. 3, pp. 277-292, March 1994.

[Rab91] J. Rabaey, C. Chu, P. Hoang, M. Potkonjak: “Fast Prototyping of Datapath-Intensive Architectures”, IEEE Design and Test of Computers, Vol. 8, No. 2, pp. 40-51, June 1991.

[Rei68] R. Reiter: “Scheduling Parallel Computations”, Journal of the ACM, Vol. 15, No. 4., pp. 590-599, 1968.

[Ren81] M. Renfors, Y. Neuvo: “The maximum sampling rate of digital filters under hardware speed constraints, IEEE Trans. on Circuits and Systems, Vol. 28, No. 2, pp. 178-195, 1981.

[Sri94] M.B. Srivastava, M. Potkonjak: “Transforming Linear Systems for Joint Latency and Throughput Optimization”, EDAC-94, pp. 267-271, 1994.

[Sta92] C. H. Stapper, “A New Statistical Approach for Fault-Tolerant VLSI Systems”, The 22nd Annual International Symposium on Fault-Tolerant Computing, pp. 356-365, Boston, MA, 1992.

[Tou93] H.J. Touati, R.K. Brayton: "Computing the initial states of retimed circuits", IEEE Trans. on CAD, Vol. 12, pp. 157-162, 1993.

[Ung86] S.H. Unger, C-J. Tan: "Optimal Clocking Schemes for High Speed Digital Systems", IEEE Transactions on Computers, Vol. 35, No. 10, pp. 880-895, 1986.

[Wal91] R.A. Walker, R. Camposano: "A Survey of High-Level Synthesis Systems", Kluwer Academic Publishers, Boston, Ma, 1991.

[Won89] D. Wong, G. De Micheli, M. Flynn: "Inserting Active Delay Elements to achieve wave pipelining", ICCAD, pp. 270-273, 1989.

[Won92] D.C. Wong, G. De Micheli, M. Flynn, R.E. Houston: "A Bipolar Population Counter Using Wave Pipelining to Achieve 2.5X Normal clock Frequency", IEEE Journal of Solid-State Circuits, Vol. 27, No. 5, pp. 745-753, 1992.