Behavioral Optimization Using the Manipulation of Timing Constraints

Miodrag Potkonjak, *Member, IEEE*, and Mani Srivastava

***Abstract*— We introduce a transformation, named rephasing, that manipulates the timing parameters in control-data-flow graphs (CDFG's) during the high-level synthesis of data-path-intensive applications. Timing parameters in such CDFG's 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. 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 data-path-intensive applications either have 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 data-path 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 such as area, throughput, latency, and power. 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. We have also investigated the relationship and interaction of rephasing with other high-level synthesis tasks.

***Index Terms*—Behavioral synthesis, transformations.**

I. Introduction

ALGORITHM transformations have emerged as important tools in the process of high-level synthesis. By restructuring the user-specified algorithm prior to the data-path 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: 1) data-flow optimizations based on algebraic identities (distributivity, commutativity, associativity, the inverse element law, etc.) and redundancy manipulation (common subexpression elimination and replication, copy propagation, constant propagation [9], etc.), 2) reorganization of control-flow structures such as conditionals and loops (retiming, loop unfolding, loop permutation, etc.), and 3) 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 toward optimizing various design metrics or meeting certain design goals.

Rephasing is targeted at computationally intensive applications that are processing streams of data. Typical examples include many of digital signal processing (DSP), video, control, and communications designs. The control-data-flow graphs (CDFG's) that are used during the high-level synthesis of these application-specific integrated-circuit (ASIC) computationally intensive systems have associated timing parameters such as sampling 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 DSP CDFG's 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 constraint, 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 either have 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 data-path allocation/scheduling step in the case of newer schedulers that use techniques like software pipelining and overlapped scheduling to generate complex time shapes.

Manuscript received October 27, 1994; revised March 9, 1998. This paper was recommended by Associate Editor M. C. McFarland.

M. Potkonjak is with the Computer Science Department, University of California, Los Angeles, CA 90095 USA (e-mail: miodrag@cs.ucla.edu).

M. Srivastava is with the Electrical Engineering Department, University of California, Los Angeles, CA 90095-1594 USA (e-mail: mbs@ee.ucla.edu).

Publisher Item Identifier S 0278-0070(98)08487-5.

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.

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 [2], [8], [24], most high-level synthesis work, and in particular the work in DSP and numerically intensive applications, has been based on the synchronous data-flow 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) [33], [34], [40] 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 [12], [25], [38].

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.* [20] 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 data-flow optimizations. Perhaps the first direct effort at directly manipulating the phases as part of an algorithm transformation was described by Srivastava and Potkonjak [43], who applied it to optimize throughput and latency simultaneously for linear DSP systems. Besides targeting a narrower set of goals for the limited domain of linear systems, Srivastava and Potkonjak [43] 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).

Several recent scheduling efforts proposed techniques that remove all artificial constraints induced by the boundary of iterations [10], [16], [21]. Unfortunately, it is done within the scheduling phase of the synthesis process and therefore prevents the application of other transformations.

Also of interest to us is the analogous problem of timing in logic synthesis. In fact, Iqbal *et al.* [20] 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 [1] and theoretically studied [6] since the mid-1960's. More recently, attention has been given to this transformation from a logic-synthesis viewpoint [26], [46]. A comprehensive coverage of the theoretical and practical issues in wave pipelining is given in [13]. 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 [11], [18]. Ishii *et al.* provide a comprehensive survey of optimization of multiphase circuitry [19]. Note that these techniques are in some sense analogous to rephasing at this level of design-process abstraction.

II. Rephasing: Definition and Intuition

A. Representation of DSP Systems by CDFG

The DSP systems that we are interested in have multiple inputs, multiple outputs, and a 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 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 CDFG's are of course causal; this means that they are equivalent to a function that expresses the ith set of output samples in terms of the ith 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*

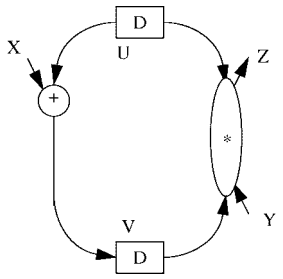

Fig. 1. An example CDFG.

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, for most of this paper, to single-rate systems where the data rate is identical on all the inputs, outputs, and data edges. In the rest of this paper, we use just the term CDFG to refer to such a "single-rate synchronous CDFG." However, in Section VII, we briefly discuss how rephasing might be extended to multirate systems.

Fig. 1 shows an example CDFG with two inputs X and Y, and one output Z. The two delay node U and V are represented by boxes with the letter D.

B. 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

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

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

$S[n] = (S_1[n]\ S_2[n]\ \cdots\ S_R[n])^T$ be the vector of nth 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 $n = 1, 2, 3, \cdots$.

Various timing parameters are associated with a CDFG, as shown in Fig. 2. Since we have restricted ourselves to single-rate synchronous CDFG's, the data rates are identical at all nodes so that the intersample 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 the *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.

A second set of timing parameters associated with a CDFG is the set of pair-wise latencies. The *latency* $T_L(i, j)$ from the ith input node to the jth output node is the delay between the arrival of a sample at the ith input and the production of the corresponding sample at the jth output. The sample correspondence is defined by the initial CDFG given by the user to specify the system—the nth sample at an output corresponds to the nth 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 nth output samples. The pair-wise 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 [43] has attention been paid to latency in synthesis of DSP systems.

The relative arrival times of the nth 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 nth sample at the ith input, relative to the arrival of $X_1[n]$, the nth sample at the first input. We call $\Phi_I(i)$, which may be negative, the *phase* associated with the ith input. The interface to the external world may require that the input phases be constrained to specific values. However, many high-level synthesis systems [40] simplify scheduling by assuming that the nth sample at each input arrives at the same time, i.e., $\Phi_I(i) = 0$ for all $i = 1 \cdots P$.

It must be noted that the timing parameters T_S, $T_L(i, j)$, and $\Phi_I(i)$, for $i = 1 \cdots P$, $j = 1 \cdots Q$, 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 nth sample at the ith delay node of the CDFG, relative to the arrival of $X_1[n]$, the nth sample at the first input. We call $\Phi_D(i)$ the phase associated with the ith 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.

[1] We use the term "system" interchangeably to mean "the system behavior specification," "an algorithm that realizes the system behavior specification," and "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 (CDFG's) that express the specified behavior, and there can be multiple implementations (hardware allocation and schedule) of an algorithm (CDFG).

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

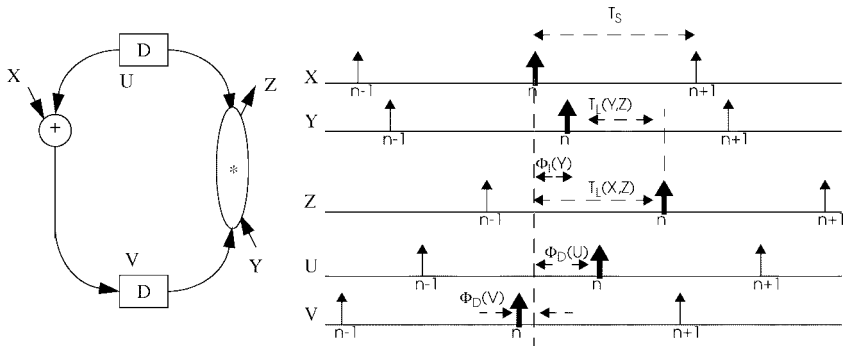

Fig. 2. Timing parameters associated with a CDFG.

What makes the delay node phases $\Phi_D(i)$ for $i = 1 \cdots R$ 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 delay node phases. Let

- $P_{ID}(i, j)$ length of the CDFG path from the ith input node to the jth delay node;
- $P_{IO}(i, j)$ length of the CDFG path from the ith input node to the jth output node;
- $P_{DD}(i, j)$ length of the CDFG path from the ith delay node to the jth delay node;
- $P_{DO}(i, j)$ length of the CDFG path from the ith delay node to the jth 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

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

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 affect the CDFG path lengths P_{ID}, P_{IO}, P_{DD}, and P_{DO}.

C. 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 very-large-scale-integration (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. Fig. 3 introduces the basic idea behind our new transformation. The CDFG shown has two operations: multiplication, which takes three control cycles, and addition, which takes one 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 two cycles. The initial best sampling rate is obviously three 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 cannot 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 the multiplication operator, which requires at least three control cycles and cannot be subdivided using delay nodes.

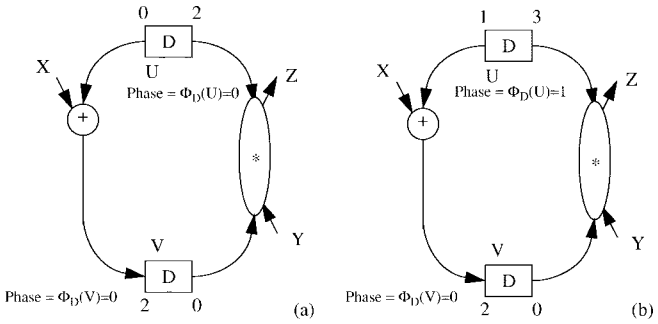

Fig. 3. Rephasing: the introductory example. The example is also used to illustrate how rephasing can be used to eliminate granularity bottlenecks.

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

Fig. 4. Functional dependencies for the first three iterations of the example in Fig. 3.

Among previously proposed transformations, only unfolding or unfolding combined with functional pipelining and retiming can resolve the throughput (sampling rate) constraint. However, it is well known that unfolding often results in increased hardware requirements due to a need for a significantly more complex controller and the loss of regularity in the structure of the data path.

Now consider the same CDFG, as shown in Fig. 3(b). The only difference is that the phase, or timing constraints, associated with the delay node U is moved by one control step relative to the 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 Fig. 4, and from the graphically represented schedule in Fig. 5. In the remainder of this paper, we will demonstrate systematic methods for exploring this mechanism of phase alteration to improve a number of design metrics.

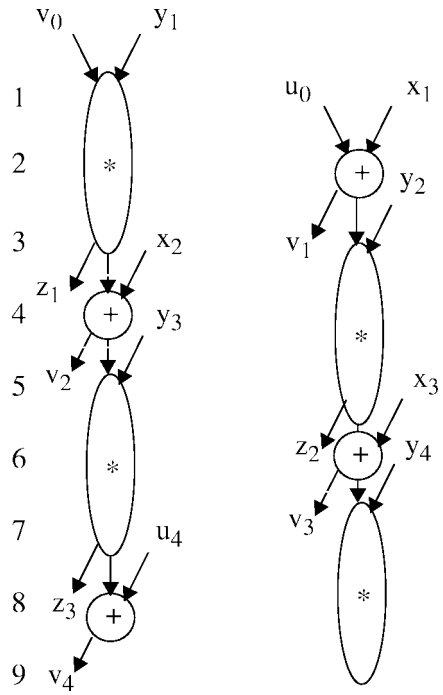

Fig. 5. Using rephasing to improve throughput by eliminating granularity bottlenecks. Schedule for the example shown in two different representations in Figs. 3 and 4.

III. Properties of Rephasing

A. 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, the 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 [27].

B. Rephasing Versus Range Scheduling (Blocking and Cycle-Static Solutions)

Range scheduling, cyclo-static scheduling, and blocking are among the several recently proposed scheduling techniques that remove artificially induced constraints due to the boundary of iterations [10], [16], [21]. It is important to note that the range scheduling (and blocking) and rephasing are not equivalent. All these scheduling techniques operate at a different level of abstraction in synthesis and compilation process than rephasing. This by itself has numerous ramifications. However, this is not the only point of difference. The range scheduling and other proposed techniques do not answer the question of when the introduction of transfer units is required, and may result in time and hardware overhead related to computation of initial values for delays. More important, the fact that rephasing does not involve scheduling enables combination of rephasing with other transformations. Coordinated application of several transformations often greatly outperforms the application of individual transformations.

There are two major aspects related to proper (minimal and complete) capturing of dependencies among operations and data transfers for computations defined on semiinfinite (or long) streams of input data. The first is definition of iteration and the second is proper calculation of the minimal set of dependences relationship among all operations and data transfers. Note that the notion of iteration is a fundamental one in computations done on semiinfinite data streams. The notion of iteration greatly facilitates minimization of the controller and calculation of initial values for delays and, maybe most important, enables effective application of transformations.

Classical scheduling techniques in both behavioral synthesis and software compilation have a complete and fully defined notion of iteration. However, this is established at the expense of producing an often too pessimistic set of dependences among operations belonging to different iterations instead of using the real minimal set of dependences as imposed by the specification of the computation.

The range scheduling (and other techniques such as unfolding and the blocked scheduling), on other hand, has almost complete information about data dependencies. These scheduling techniques, at least in their current form, do not resolve issues about when transfer units are required for the correct implementation and do not address how initial values of the states can be calculated. However, this comes at the expense of essentially complete elimination of information about iterations.

The power of rephasing comes from its ability to simultaneously, properly, and in a very simple and clean way capture the least restrictive set of data and control dependences and to provide complete information about the boundaries of iterations.

C. Rephasing Versus Retiming

There is a close and easily seen relationship between rephasing and retiming [30] 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 the following.

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 [45] and sometimes infeasible task, particularly for arbitrary CDFG's not restricted to traditional operators of $+$ and $*$. Rephasing has an edge over retiming 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 [30]. 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 [38]. Leiserson and Saxe were the first to algorithmically study retiming [30]. 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 that 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 [30].

An iteration bound is the bound imposed by data dependences and the length of computations in cycles. Interestingly, the iteration bound was first discovered and discussed in operation research literature under the picturesque name of "tramp steamer problem" [7]. 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 the name of minimum (cost-to-time) ratio cycle problem [22], [27]. Currently, the most efficient algorithm is one proposed by Hartmann [15]. Reiter [41] was the first to point out the relevance of iteration bound to maximal speed of computation in recursive computations. In the early 1980's, the iteration bound was rediscovered, and thus named, in the VLSI DSP community [42].

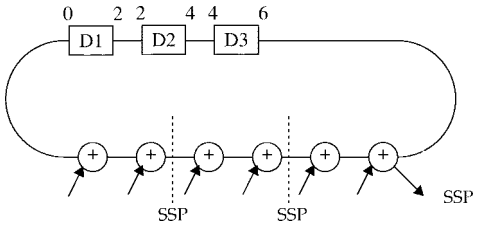

Fig. 6. Breaking input–output bottleneck using rephasing. While retiming is unable to reduce critical path of six cycles, the application of rephasing reduces the critical path to two cycles.

Fig. 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 two ($= 6/3$) 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 cannot be moved to a new position that reduces the critical path because the delay node has to be moved across a multiinput operator that does not have a delay node on at least one of its inputs. Fig. 6 again provides the illustration. Although the iteration bound is only two cycles, input/output bottlenecks dictate the critical path and therefore the best sample period of six 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, the graph always can be retimed such that iteration bound is achieved [30]. Potkonjak and Rabaey showed that the addition of pipelining as a preprocessing step to retiming always removes the input/output bottleneck [37].

Fig. 3(a) 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 three 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 two 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 an attractive

option from the implementation point of view, retiming cannot improve the critical path.

Fig. 3(b) 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 Fig. 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 situations, 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 the 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 Fig. 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.

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 a long finite-duration impulse response (FIR) filter in direct form [Fig. 7(a)], which is often retimed to the transposed form so that the critical path is reduced. Fig. 7(b) 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 an identical number of bits, after retiming there is often a need to allocate significantly more bits for storing the new delay node values. Fig. 7(c) 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.

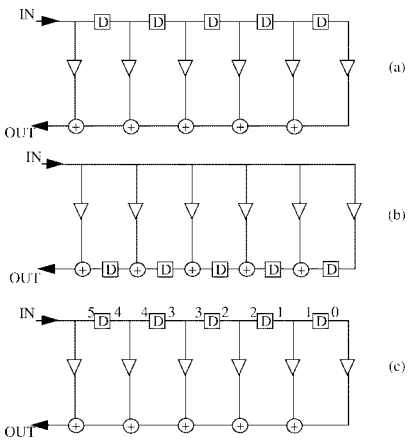

Fig. 7. FIR filter—retiming t versus rephasing. (a) Initial direct form, (b) transposed form obtained after the application of retiming, and (c) direct form after rephasing (see Section II-C).

IV. 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 dataflow transformations (e.g., algebraic and redundancy manipulation) are not used. In this section, we demonstrate how rephasing can be used as an effective means 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 V.

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

A. 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 [27], 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. The SCC can be identified using the stand graph-theoretic approach [5].)
4) Assign to the selected node phase zero. Assign to each delay node a phase that is equal to the 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 operation nodes in the CDFG.

For the sake of brevity, we omit the proof. The proof also can be 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.

B. 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 that is needed is to run the Bellman–Ford algorithm on the CDFG. The distances from the delay nodes in $G2$ of edges that 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 [30], 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 the 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 pseudocode.

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 the 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 VII.

V. Optimizing Area Using Rephasing

In this section, we demonstrate the application of rephasing for minimization of area. The application of rephasing to other popular design metrics, such as power, testability, and permanent and transient fault tolerance, can be found in [39].

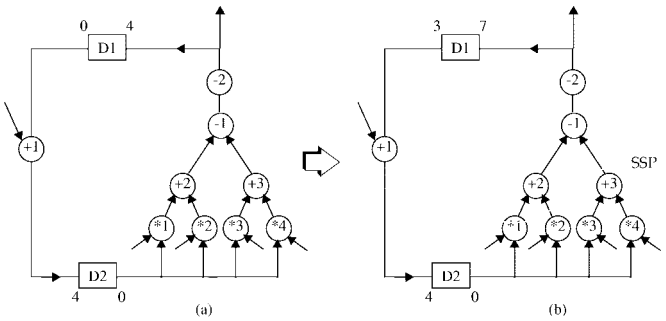

Fig. 8. Area minimization using rephasing. While for the initial CDFG four multipliers, two adders, and one subtracter is required, for the transformed CDFG, one multiplier, one adder, and one subtracter are sufficient. The corresponding schedules are shown in Table I.

TABLE I
A Possible Schedule for the Rephased Example in Fig. 8(b)

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

A. 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 [38]. However, the effectiveness of retiming for area optimization is often severely limited by granularity bottlenecks and input/output bottlenecks. Fig. 8(a) shows a typical example of this situation. It is assumed that the available time is four control steps. Retiming cannot 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 four multipliers, two adders, and one subtracter.

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

It has also been recognized that the constraints imposed only by input and output positioning can be overcome by the use of pipelining [37]. However, there are at least two type of designs 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 cannot 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 [38]. 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 a result as reported for retiming [38] functional pipelining [37] using Hyper always will be 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

$$u_i(k) = \frac{1}{\mathrm{ALAP}_i - \mathrm{ASAP}_i + 1}$$

for each time $t \leq k < t + \mathrm{Duration}_i$. 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

$$U(t) = \sum_{i=1}^{\mathrm{Num\ Operations}} u_i(t). \tag{1}$$

Last, 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 that will 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 [38]. 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 [36] where at each step, the most critical part of the predicted cost is targeted. The following pseudocode 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 [40]. As soon as the requirement for some type of unit reaches the min-bound, further improvements in the value of this component are not considered as the improvements of the OF, since the minimum along this dimension has been reached.

Last, 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 was done in [38].

VI. Relationship with Other High-Level Synthesis Tasks

A. 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 that is needed is to find for a given rephasing the corresponding functional pipelining on the corresponding modified CDFG (see Section III) and denote which operation nodes in the CDFG are now aligned at the beginning (and therefore also at the end) of the iteration period. In Figs. 5 and 7, those nodes are marked by the scheduling starting point (SSP) cuts.

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.

B. 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 transformation 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 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 [20] can be easily modified to the 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.

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 [12], [25] 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 important, the result of the sequence of transformations can be analyzed independent of the scheduling.

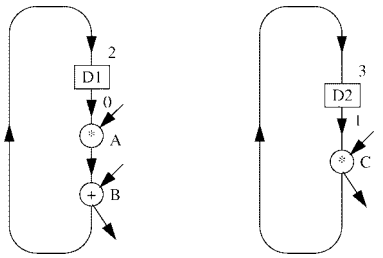

Fig. 9. A multirate CDFG and its equivalent single-rate CDFG.

VII. Rephasing in Multirate Systems

So far, we focused on systems with single-rate synchronous CDFG's, where the data rate is identical on all the inputs, outputs, and internal edges. As a result, every computation node is executed only once during an iteration: it consumes a single data sample from each of its incoming data edges and produces a single sample on each of its output edge. However, single-rate CDFG's are not sufficient to represent the complexity of many present-day DSP designs, which often require multirate CDFG's [28], [35]. In multirate systems, the computation loop may require different nodes to be executed a different number of times in a single computation iteration. While we do not develop the theoretical framework underlying rephasing of multirate CDFG's, in this section, we show that rephasing is indeed applicable to such CDFG's and carries similar benefits as in rephasing of single-rate CDFG's.

Fig. 9(a) shows an example of a multirate CDFG. Following the notation of [35], the numbers at the inputs of a node represent the number of samples consumed by it from that input on each invocation of the node. Similarly, the numbers at the outputs of a node represent the number of samples produced by it on that output on each invocation. In this example, node A fires twice for every invocation of node B.

Obviously, one simple approach to apply rephasing to multirate CDFG's is to make use of the observation [35] that any multirate CDFG can be expressed as an equivalent single-rate CDFG to which rephasing can then be applied. Fig. 9(b) shows the equivalent single-rate representation of the multirate CDFG in Fig. 9(a) in which odd and even invocations of node A in each iteration are executed by nodes A_o and A_e.

However, one can also extend the notion of rephasing to the case of multirate CDFG's directly and derive benefits similar to that for rephasing of single-rate CDFG's. For example, consider Fig. 9(a) again, and assume that node A has a computation delay of 12 time units and node B has a computation delay of four time units. We will intuitively compare retiming and rephasing.

If further pipelining is not allowed, then this CDFG cannot be retimed due to input/output bottleneck that prevents the delays nodes from being moved around. Even if extra levels of pipelining can be introduced, the retiming would suffer from the node operator granularity problem since node A has a much higher delay than node B. In particular, the best input sample rate that can be achieved is two samples per 12 time units, or a minimum sample period of six.

Rephasing of multirate CDFG's, just like in the single-rate CDFG case, does not suffer from input/output and operator

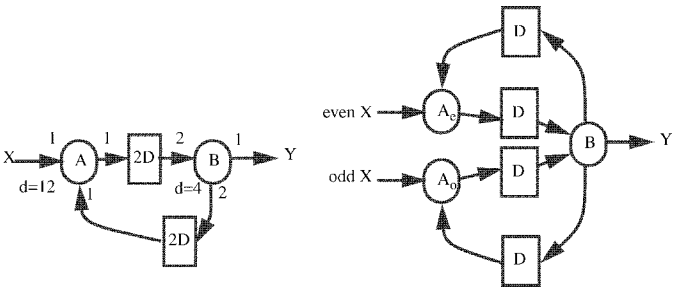

Fig. 10. Multirate CDFG before and after rephasing.

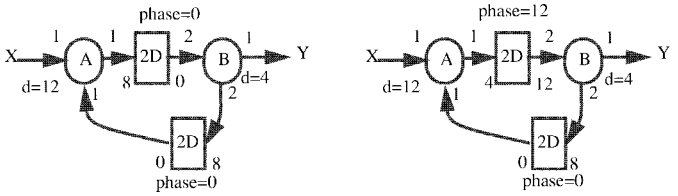

Fig. 11. Using rephasing to improve throughput of multirate CDFG's by eliminating granularity and input/output bottlenecks.

granularity bottlenecks. Fig. 10 shows the multirate CDFG of Fig. 9(a), both before and after rephasing. Fig. 11 shows the corresponding schedule, which shows that rephasing achieves an input sample rate of four samples per 16 (12 + 4) time units, or a sample period of four. This is the theoretically best possible sample period, corresponding to the iteration bound of the CDFG. The crucial thing to note is that in multirate CDFG's, the phases may be assigned to groups of delay nodes instead of individual delay nodes corresponding to least common multiple of the number of samples being produced and consumed by the upstream and downstream nodes on each invocation.

VIII. Experimental Results

Throughput optimization using rephasing is a problem of polynomial complexity. We presented the optimization algorithm in Section III. 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 three to 380 control cycles. The average critical path was 67 control cycles, while the median was 14. After rephasing the shortest critical path was one cycle long, and the longest was only 17 control cycles long. The average critical path was 3.6 cycles long, while the median critical path was only two control cycles long. So the median improvement was by a factor of seven, while the average was improved by more than 20 times.

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

TABLE II
Area Optimization Using Rephasing

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

IX. 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.

References

[1] C. Anderson, J. Earle, R. Glodschmidt, and D. Powers, "The IBM system/360 model 91 floating point execution unit," *IBM J. Res. Develop.*, vol. 11, no. 1, pp. 34– 53, 1967.
[2] R. A. Bergamaschi and A. Kuehlmann, "A system for production use of high-level synthesis," *IEEE Trans. VLSI Syst.*, vol. 1, no. 3, pp. 233–243, 1993.
[3] A. P. Chandrakasan, M. Potkonjak, J. Rabaey, and R. Brodersen, "Hyper-LP: A design system for power minimization using architectural transformations," in *Proc. ICCAD*, 1992, pp. 300–303.
[4] K. T. Cheng and V. D. Agarwal, "A partial scan method for sequential circuits with feedback," *IEEE Trans. Comput.*, vol. 39, no. 4, pp. 544–548, 1990.
[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* Cambridge, MA: MIT Press, 1990.
[6] L. W. Cotten, "Circuits implementation of high-speed pipeline systems," in *Proc. AFIPS Fall Joint Conf.*, 1965, vol. 27, pp. 489–504.
[7] G. B. Danzig, W. Blattner, and M. R. Rao, "Finding a cycle in a graph with minimum cost to time ratio with application to a ship routing problem," in *Theory of Graphs*, P. Rosenstiehl, Ed. New York: Paris and Gordon and Breach, Dunod, 1967, pp. 77–84.
[8] D. Filo, D. C. Ku, C. N. Coehlo, and G. De Micheli, "Interface optimization for concurrent systems under timing constraints," *IEEE Trans. VLSI Syst.*, vol. 1, no. 3, pp. 268–281, 1993.
[9] C. N. Fischer and R. J. Le Blank, *Crafting a Compiler.* Menlo Park, CA: Benjamin/Cummings, 1988.
[10] P. R. Gelabert and T. P. Barnwell, "Optimal automatic periodic multiprocessor scheduler for full specified flow graphs," *IEEE Trans. Signal Processing*, vol. 41, no. 2, pp. 858–888, 1993.
[11] L. A. Glasser and D. W. Dobberpuhl, *The Design and Analysis of VLSI Circuits.* Reading, MA: Addison-Wesley, 1985.
[12] G. Goossens, J. Wandewalle, and H. DeMan, "Loop optimization in register-transfer scheduling for DSP-systems," in *Proc. DAC-89*, 1989, pp. 826–831.
[13] C. T. Gray, W. Liu, and R. K. Cavin, *Wave Pipelining: Theory and CMOS implementation.* Boston, MA: Kluwer, 1993.
[14] L. Guerra, M. Potkonjak, and J. Rabaey, "High level synthesis for reconfigurable datapath synthesis," in *Proc. ICCAD-93*, 1993, pp. 26–29.
[15] M. Hartman, "On cycle means and cycle staffing," Univ. of North Carolina at Chapel Hill, Tech. Rep. UNC/OR/TR-90/14, June 1990.

[16] S. M. Heemstra de Groot, S. H. Gerez, and O. E. Herrman, "Range-chart-guided iterative data-flow graph scheduling," *IEEE Trans. Circuits Syst. Part I*, vol. 39, no. 5, pp. 351–364, 1992.
[17] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* San Mateo, CA: Morgan Kaufmann, 1989.
[18] A. T. Ishii and C. E. Leiserson, "A timing analysis of level-clocked circuitry," in *Advance Research in VLSI: Proc. 6th MIT Conf.*, 1990, pp. 113–130.
[19] A. T. Ishii, C. E. Leiserson, and M. C. Papaefthymiou, "Optimizing two-phase level-clocked circuitry," *J. ACM*, vol. 44, no. 1, pp. 148–199, 1997.
[20] Z. Iqbal, M. Potkonjak, S. Dey, and A. Parker, "Critical path minimization using retiming and algebraic speed-up," in *Proc. 30th ACM/IEEE DAC*, 1993, pp. 573–577
[21] K. Ito, L. E. Lucke, and K. K. Parhi, "Module Selection and data format conversion for cost-optimal DSP synthesis," in *Proc. ICCAD'94*, 1994, pp. 322–329.
[22] R. Karp, "A characterization of the minimum cycle mean in a digraph," *Discrete Math.*, vol. 23, pp. 309–311, 1978.
[23] R. Karri and A. Orailoglu, "Transformation-based high-level synthesis of fault-tolerant ASIC's," in *29th ACM/IEEE Design Automation Conf.*, 1992, pp. 662–665.
[24] D. Ku and G. De Michelli, *High Level Synthesis of ASIC's Under Timing and Synchronization Constraints.* Norwell, MA: Kluwer, 1992.
[25] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *SIGPLAN'88 Conf. Programming Language Design and Implementation*, 1988, pp. 318–328.
[26] W. C. K. Lam, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Valid clocking in wavepipelined circuits," in *Proc. ICCAD*, 1992, pp. 518–525.
[27] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids.* New York: Holt, Rinehart and Winston, 1976.
[28] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, pp. 1235–1245, Sept. 1987.
[29] T. C. Lee, N. K. Jha, and W. H. Wolf, "Behavioral synthesis of highly testable data path under nonscan and partial scan environments," in *Proc. DAC'93*, 1993, pp. 616–619.
[30] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 3–36, 1991.
[31] P. Marwedel, "Cooperation of synthesis, retargetable code generation, and test generation in the MSS," in *Proc. Eur. Design and Test Conf.*, 1993, pp. 63–69.
[32] ———, "Tree-based mapping of algorithms to predefined structures," in *Proc. IEEE ICCAD'93*, 1993, pp. 586–593.
[33] M. C. McFarland, A. C. Parker, and R. Camposano, "Tutorial on high-level synthesis," in *Proc. 25th Design Automation Conf.*, 1988, pp. 330–336.
[34] S. Note, W. Geurts, F. Catthoor, and H. De Man, "'Cathedral-III' architecture-driven high level synthesis for high throughput DSP applications," in *Proc. Design Automation Conf.*, 1991, pp. 597–602.
[35] K. K. Parhi, "Algorithm transformation techniques for concurrent processors," *Proc. IEEE*, vol. 77, pp. 1879–1895, Dec. 1989.
[36] P. G. Paulin and J. P. Knight, "Scheduling and binding algorithms for high-level synthesis," in *Proc. DAC-89*, 1989, pp. 1–6.
[37] M. Potkonjak and J. Rabaey, "Pipelining: Just another transformation," in *Proc. Int. Conf. Application Specific Array Processors*, 1992, pp. 163–177.
[38] ———, "Optimizing resource utilization using transformations," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 277–292, Mar. 1994.
[39] M. Potkonjak and M. B. Srivastava, "Behavioral optimization using the manipulation of timing constraints," Dept. of Computer Science, Univ. of California, Los Angeles, Tech. Rep. 950057, 1995.
[40] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast prototyping of datapath-intensive architectures," *IEEE Design Test Comput. Mag.*, vol. 8, pp. 40–51, June 1991.
[41] R. Reiter, "Scheduling parallel computations," *J. ACM*, vol. 15, no. 4, pp. 590–599, 1968.
[42] M. Renfors and Y. Neuvo, "The maximum sampling rate of digital filters under hardware speed constraints," *IEEE Trans. Circuits Syst.*, vol. 28, no. 2, pp. 178–195, 1981.
[43] M. B. Srivastava and M. Potkonjak, "Transforming linear systems for joint latency and throughput optimization," in *Proc. EDAC'94*, 1994, pp. 267–271.
[44] C. H. Stapper, "A new statistical approach for fault-tolerant VLSI systems," in *Proc. 22nd Ann. Int. Symp. Fault-Tolerant Computing*, Boston, MA, 1992, pp. 356–365.
[45] H. J. Touati and R. K. Brayton, "Computing the initial states of retimed circuits," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 157–162, 1993.
[46] D. Wong, G. De Micheli, and M. Flynn, "Inserting active delay elements to achieve wave pipelining," in *Proc. ICCAD*, 1989, pp. 270–273.

Miodrag Potkonjak (S'90–M'91) received the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1991.

In September 1991, he joined C&C Research Laboratories, NEC USA, Princeton, NJ. Since 1995, he has been an Assistant Professor in the Computer Science Department of the University of California, Los Angeles. His research interests include intellectual-property-protection techniques, system design, collaborative design, integration of computations and communications, and experimental algorithmics.

Dr. Potkonjak received the NSF Career award in 1998.

Mani Srivastava received the B.Tech. degree in electrical engineering from IIT Kanpur, India, and the M.S. and Ph.D. degrees from the University of California, Berkeley.

He is an Assistant Professor of electrical engineering at the University of California, Los Angeles. From 1992–1996, he was a Member of Technical Staff at Bell Laboratories in networked computing research. His current research interests are in mobile and wireless networked computing systems, low-power systems, and hardware and software synthesis of embedded DSP systems.

Dr. Srivastava received the NSF Career award in 1998 and the President of India Gold Medal in 1985.