Design Optimization and Space Minimization Considering Timing and Code Size via Retiming and Unfolding *

Qingfeng Zhuge, Chun Xue, Zili Shao, Meilin Liu,
Meikang Qiu, Edwin H.-M. Sha
Department of Computer Science
University of Texas at Dallas
Richardson, Texas 75083, USA

Abstract

The increasingly complicated DSP processors and applications with strict timing and code size constraints require design automation tools to consider multiple optimizations such as software pipelining and loop unfolding, and their effects on multiple design parameters. This paper presents an Integrated Framework for Design Optimization and Space Minimization (IDOM) toward finding the minimum configuration satisfying timing and code size constraints. We show an effective way to reduce the design space to be explored through the study of the fundamental properties and relationships among retiming function, unfolding factor, timing, and code size. We present theorems to show that a small set of feasible unfolding factors can be obtained effectively to produce high-quality designs. The IDOM algorithm is proposed to generate a minimal configuration of the design by integrating software pipelining, unfolding, and code size reduction techniques. The experimental results on a set of DSP benchmarks show the efficiency and effectiveness of our technique.

*This work is partially supported by TI University Program, NSF EIA-0103709, Texas ARP 009741-0028-2001, NSF CCR-0309461, NSF IIS-0513669, Microsoft, USA.

1 Introduction

DSP processors and their applications usually have strict requirements in time, code size, hardware cost, area, etc. Many signal processing applications are computation-intensive programs depend on time-critical sections consisting of a loop of instructions. The increasingly complicated embedded signal processing systems require extensive design automation and optimization tools. One of the most challenging problems in high-level synthesis is how to explore a wide range of design options to achieve high-quality designs within a short time [2, 6, 14]. For high-performance embedded systems with limited on-chip memory resource, the design exploration problem is to find the minimum number of functional units or processors for executing an application with timing and code size requirements.

To achieve high execution rate, optimization techniques such as loop unfolding and software pipelining are commonly used to optimize the schedule length [3, 5, 9, 12, 13, 17]. It is necessary that the designer is able to choose the proper unfolding factor and software pipelining degree to satisfy the performance constraint. On the other hand, these optimization techniques tend to incur a large code size expansion [26], which is not desirable due to a very restricted memory resource. Therefore, a good design method should be able to exploit both performance and code size optimization techniques to produce a high-quality design solution. Without this capability, a design exploration method either easily fails for finding feasible solutions, even with exhaustive search, or cannot locate the superior solutions.

A number of research results have been published on timing optimization [1,3–5,7,10–12,17–20] or code size optimization problems [8,21,25]. However, we are unaware of any previous work that addresses the combination of various optimization techniques and their effects on design. The previous works of design space exploration and high-level synthesis [2, 6, 16, 23, 24] rarely consider the effectiveness of reducing the design space through the study of relationships between optimization parameters such as software pipelining degree, unfolding factor, and the design criteria such as timing and code size. We strongly believe that an effective way to reduce the design space is through the study of the fundamental properties and relations among multiple design parameters, such as retiming value, unfolding factor, time, and code size.

In the following, we use the example in Figure 1(a) to show the performance effects and de-

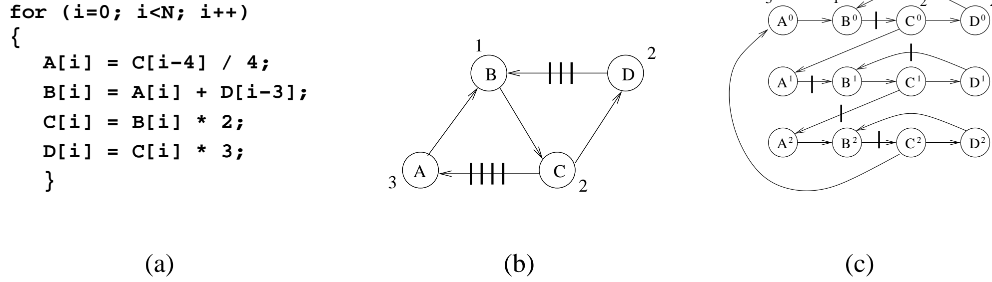

Figure 1: (a) A data flow graph. (b) The unfolded and retimed graph.

sign issues when using unfolding and software pipelining. In this paper, we use the data flow graph (DFG) to model an iterative program, as shown in Figure 1(b). A node in a DFG represents an operation, such as summation, multiplication, etc. The weight of a node represents the computation time of an operation. An edge represents a dependence relationship between two computations. The weight of an edge, indicated by the short black bars on an edge, represents the dependence distance between two nodes. Loops are represented by cycles. To obtain the execution time of a data flow graph, we need to calculate the summation of the node weight along the longest path with zero weight (also known as the critical path). The execution time of the DFG in Figure 1(b) is the summation of the node weights along the path $A \rightarrow B \rightarrow C \rightarrow D$, that is 7 time units. The lower bound of the execution time, also known as the iteration bound, of the DFG can also be computed as 5/3 [22]. The iteration bound can be achieved by using unfolding only. The unfolding times is also called unfolding factor. It is known that the iteration bound can be achieved by unfolding factor that is equal to the Least Common Multiplier of the delays of all cycles in a DFG [17]. For our example, the unfolding factor is 12. As a result, the unfolded data flow graph will be 12 times larger. And the code size will be expanded for 12 times also. While combining software pipelining and unfolding, we only need to unfold for 3 times to achieve the iteration bound, as shown in Figure 1(c). The critical path is $C^0 \rightarrow A^1$. The computation time of the critical path is 5 time units. Because there are three iterations to be executed in parallel within 5 time units, the average execution time of one iteration is reduced to 5/3, The superscripts show the number of the duplicated nodes by unfolding. This example shows that the unfolding factor chosen in design can be much smaller by combining with software pipelining. In this pa-

per, we use the retiming concept [13] to model software pipelining [25]. The basic concepts and principles related to this work will be presented in details later.

When retiming and unfolding are applied to optimize the performance, a critical question is: What are the feasible unfolding factors that can achieve the timing requirement by combining with retiming? Instead of searching all the design options with unfolding factors 2 to 12 as in our example, and performing expensive computations on software pipelining and scheduling for all the unfolding factors, the effective unfolding options can be a very small part of all the options. In this paper, we show that we can produce *the minimum set of unfolding factors* for achieving timing requirement. Many unfolding factors can be proved to be infeasible, and eliminated immediately *without performing real scheduling*. Thus, the design space and search cost are significantly reduced. Intuitively speaking, the obtained interrelation *reduces the points to be selected from a higher-dimensional volume to a small set of lower-dimensional planes.* Since retiming and unfolding greatly expand the code size [17, 25, 26], it's possible that the generated code is too large to be fit into the on-chip memory. Thus, the relationship between schedule length and code size must be studied. As we found in our research, the relationship between code size and performance can be formulated by mathematical formula using retiming functions. It provides an efficient technique for reducing the code size of any software-pipelined loops.

In this paper, we propose an Integrated Framework for Design Optimization and Space Minimization (IDOM). It distinguishes itself from the traditional design exploration in the following ways: First, it significantly reduces the design space and the exploration cost by exploiting the properties of the unfolded and retimed data flow graph and the relationship between design parameters. Second, it combines several optimization techniques, namely, unfolding [17], extended retiming [15] and code size reduction [25, 26], to produce the minimum configuration that satisfies both timing and code size constraints,

Theories are presented to reveal the underlying relationship between unfolding, retiming, performance, and code size. IDOM algorithm is proposed and compared with the traditional approaches. Our experimental results on a set of DSP benchmarks show that the search space and search cost are greatly reduced. For example, the search space of 4-stage Lattice Filter is reduced from 909 design points to 55 points, and the search cost using IDOM is only 4% of that using the standard method. The average search cost using IDOM is only 3% of that

using the standard method for our benchmarks. Our experiments also show that IDOM algorithm constantly generates the minimal configurations.

In the next section, we present the basic concepts and principles related to our research, including data flow graph model, retiming, unfolding, and code size reduction. Section 3 presents the theoretical foundation of design space minimization. Section 4 describes the algorithms and computation costs for several design space exploration algorithms. An example for design optimization and space exploration is demonstrated in Section 5. In Section 6, we present the experimental results on a set of DSP benchmarks. Finally, concluding remarks are provided in Section 7.

2 Optimization Techniques

In this section, we provide an overview of data flow graph model, the basic principles of retiming and unfolding. An in-depth analysis for these optimization techniques based on data flow graph model reveals underlying relations between the design parameters. We also describe an code size optimization technique that reduces the code size for software-pipelined programs.

2.1 Data Flow Graph Model

Many DSP applications can be modeled as **data flow graphs (DFG)**. Then, optimization problems, such as retiming and unfolding, can be regarded as graph transformation problems. A data flow graph $G = (V, E, d, t)$ is a node-weighted and edge-weighted directed graph, where V is a set of computation nodes, $E \subseteq V \times V$ is a set of edges, d is a function from E to $\mathbb{N}$ representing the number of delays on each edge, and $t(v)$ represents the computation time of each node.

Programs with loops can be represented by cyclic DFGs. An *iteration* is the execution of each node in V exactly once. Inter-iteration dependencies are represented by weighted edges. For any iteration j, an edge e from u to v with delay $d(e)$ conveys that the computation of node v at iteration j depends on the execution of node u at iteration $j - d(e)$. An edge with no delay represents a data dependency within the same iteration. An iteration is associated with a static schedule. A static schedule must obey the precedence relations defined by the DFG. The ***cycle***

period $c(G)$ of a data-flow graph G is the computation time of the longest zero-delay path, which corresponds to the schedule length of a loop without resource constraint. The data flow graph of a loop is shown in Figure 2(a) with cycle period $c(G) = 2$.

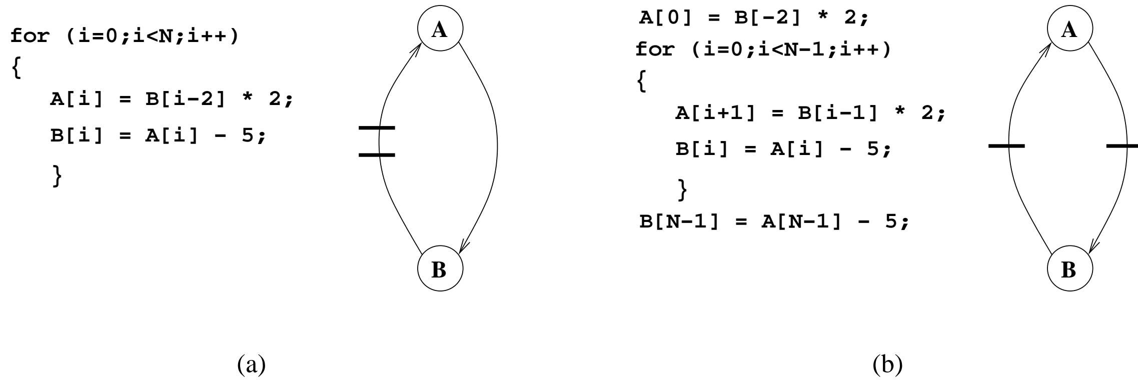

Figure 2: (a) The original code and DFG. (b) The code and the retimed DFG with $r(A) = 1$ and $r(B) = 0$.

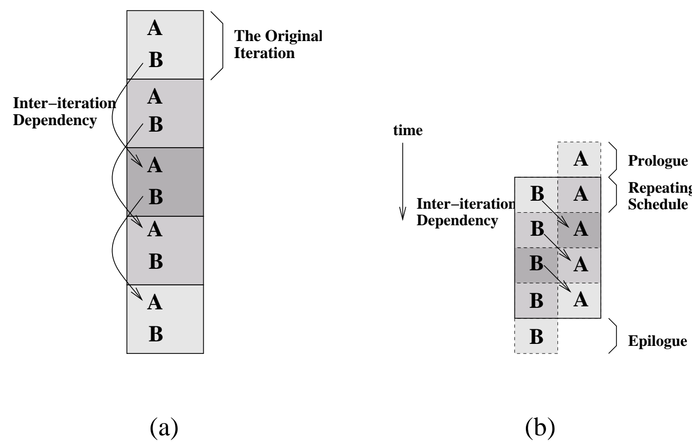

Figure 3: (a) A static schedule of original loop. (b) The pipelined loops.

2.2 Retiming and Software Pipelining

Retiming [13] is one of the most effective graph transformation techniques for optimization. It transforms a DFG to minimize its cycle period in polynomial time by redistributing delays in the DFG. The commonly used optimization technique for DSP applications, called *software pipelining*, can be correctly modeled as a retiming.

A *retiming r* is a function from V to $\mathbb{Z}^+$ that redistributes the delays in the original DFG G, resulting a new DFG $G_r = \langle V, E, d_r, t \rangle$ such that each iteration still has one execution of each node in G. The delay function changes accordingly to preserve dependencies, i.e., $r(v)$ represents delay units pushed into the edges $v \rightarrow w$, and subtracted from the edges $u \rightarrow v$, where $u, v, w \in G$. Therefore, we have $d_r(e) = d(e) + r(u) - r(v)$ for every edge $u \rightarrow v$ and $d_r(\ell) = d(\ell)$ for every cycle $\ell \in G$. Figure 2(b) shows the updated code and retimed DFG of Figure 2(a) with retiming functions $r(A) = 1$, $r(B) = 0$.

When a delay is pushed through node A to its outgoing edge as shown in Figure 2(b), the actual effect on the schedule of the new DFG is that the ith copy of A is shifted up and is executed with $(i-1)^{th}$ copy of node B. Because there is no dependency between node A and B in the new loop body, these two nodes can be executed in parallel. The schedule length of the new loop body is then reduced from two control steps to one control steps. This transformation is illustrated in Figure 3(a) and Figure 3(b).

Software pipelining can be clearly modeled by using retiming concept. In fact, every retiming operation corresponds to a software pipelining operation. When one delay is pushed forward through a node u in V, every copy of this node is moved up by one iteration, and the fi rst copy of the node is shifted out of the fi rst iteration into the prologue as shown in Figure 2(b). Consequently, the last copies of the other nodes in the data flow graph become the epilogue to fi nish the computations of a loop. With the *normalized* retiming function r, [1] we can measure the size of prologue and epilogue. When $r(v)$ delays are pushed forward through node v, there are $r(v)$ copies of node v appeared in the prologue. The number of copies of a node in the epilogue can also be derived in a similar way. If the maximum retiming value in the data flow graph is

[1]The value of normalized retiming function equals to retiming value $r(v)$ subtracts $min_v r(v)$ for every node v in V.

$max_u r(u)$, there are $max_u r(u) - r(v)$ copies of node v in the epilogue. For example, the retiming value of node A is 1 in Figure 2(a). Then, there is one copy of node A in the pipelined schedule as shown in Figure 3(b). Thus, the maximum retiming value equals to the software pipelining degree.

The *extended retiming* allows the delays to cut the execution time of a multiple-cycle node to achieve optimal schedule length [15]. Due to the space limitation, we will not illustrate the extended retiming in details. Readers interested in the extended retiming technique can refer to [15] for details.

2.3 Unfolding

Unfolding is another effective technique for improving the average cycle period of a static schedule [5,17]. Unfolding technique replicates a loop body for several times to increase the instruction-level parallelism in the unfolded loop body. The number of copies of the original loop body in an unfolded loop body is called *unfolding factor* f. From the data flow graph point of view, the nodes of the original DFG G is replicated for f times in the unfolded graph G_f. Accordingly, the schedule of an unfolded graph with unfolding factor f contains the nodes from f iterations of the original DFG. The average computation time of an iteration is called the *iteration period* P. The iteration period of an unfolded graph G_f with unfolding factor f is computed as $P = c(G_f)/f$, where $c(G_f)$ is the cycle period of the original DFG. That is, the average execution time of one iteration of the original loop is improved by a factor of f. For a cyclic DFG, the iteration period is bounded from below by the iteration bound of the graph [22] which is defi ned as follows:

Definition 2.1. *The **iteration bound** of a data-flow graph G, denoted by $B(G)$, is the maximum time to delay ratio of all cycles in G. This can be represented by the equation $B(G) = \max_{\forall \ell} T(\ell)/D(\ell)$, where $T(\ell)$ and $D(\ell)$ are the summation of all computation times and the summation of delays in a cycle ℓ.*

The iteration bound can be achieved by unfolding. An unfolding factor to achieve the iteration bound is called a rate-optimal unfolding factor. It is known that the rate-optimal unfolding factor is upper bounded by the least common multiple of the delay counts $lcm(D(\ell))$ of all the cycles ℓ in a DFG G [5, 17]. But it's likely that this rate-optimal unfolding factor is too large for the

design requirement in some cases. For example, the code size of the unfolded graph will be f times larger than the original code size. For DSP processors with limited on-chip memory resources, the unfolding factor can be too large to satisfy the code size constraint. Therefore, it is important for designers to find the minimal unfolding factor f efficiently to satisfy the timing and code size constraints.

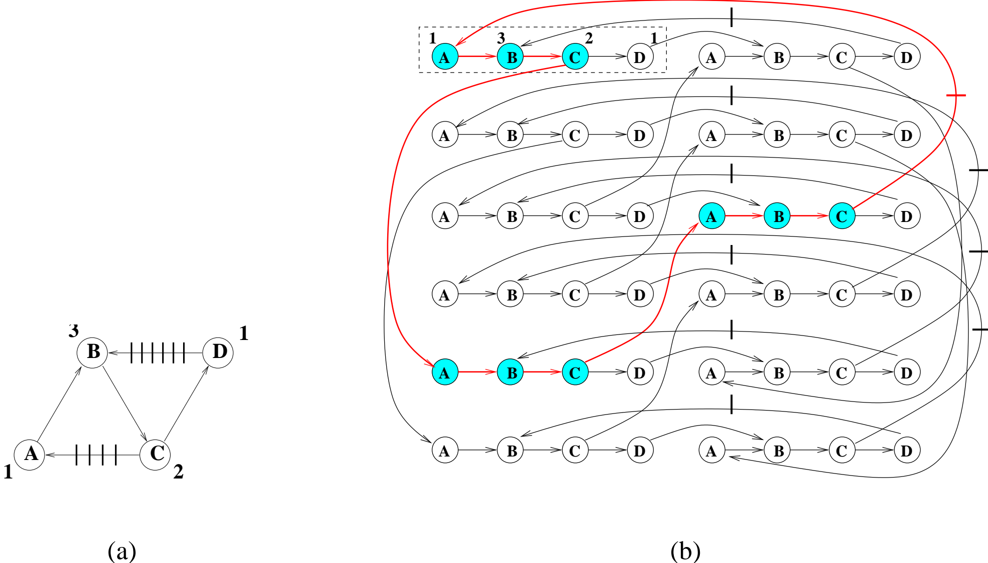

Figure 4: (a) A data flow graph. (b) The unfolded graph.

2.4 Combining Retiming and Unfolding

As we discussed earlier, unfolding can achieve iteration bound but may violate code size constraint. We will show that retiming alone may not achieve iteration bound. By combining retiming and unfolding, we are able to find a smaller rate optimal unfolding factor.

Consider the data flow graph G in Figure 4(a). The critical cycle is $B \rightarrow C \rightarrow D \rightarrow B$. Thus, the iteration bound is $B(G) = 6/4 = 3/2$. The static schedule of the original loop is shown in Figure 5(a). A rate optimal unfolding factor is the least common multiplier of delay counts of all cycles, that is 12. The unfolded graph G_f is shown in Figure 4(b). Each iteration of the unfolded loop contains twelve copies of nodes of the original loops. Assuming there is

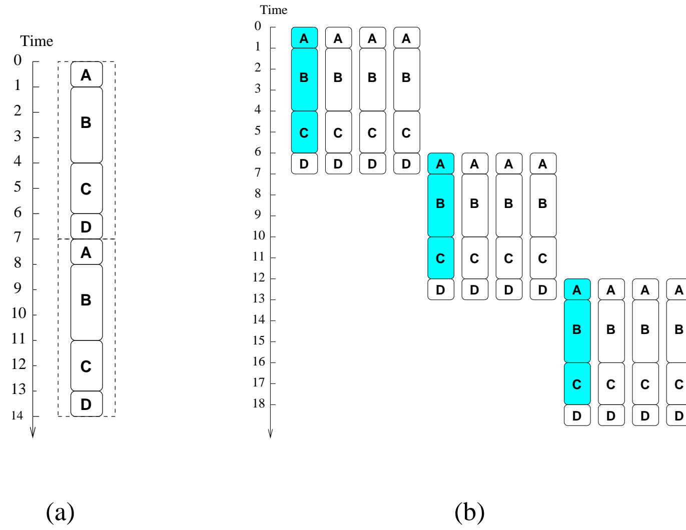

Figure 5: (a) The static schedule of the DFG in Figure 4(a). (b) A static schedule of the unfolded DFG in Figure4(b).

no resource constraint, the critical cycle indicated by the shaded nodes is finished within 18 time units for the unfolded loop. Therefore, the average execution time of one iteration of is reduced from 7 cycles to only $18/12 = 3/2$ cycles. The static schedule of the unfolded loop is shown in Figure 5(b). The shaded nodes belong to a critical cycle.

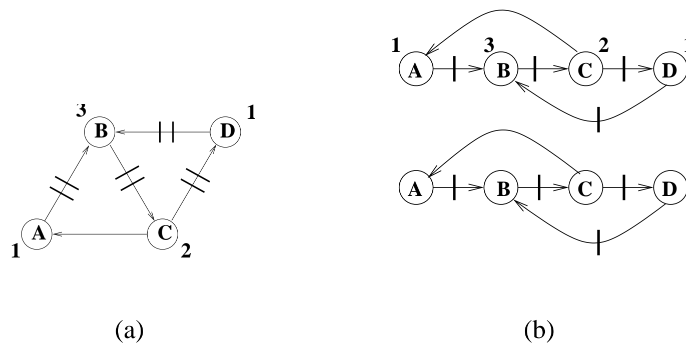

Figure 6: (a) The retimed graph. (b) The unfolded retimed graph.

By combining retiming and unfolding, the unfolding factor can be further reduced to 2, which is the minimal rate-optimal unfolding factor. The retimed graph G_r with retiming values $r(A) =$

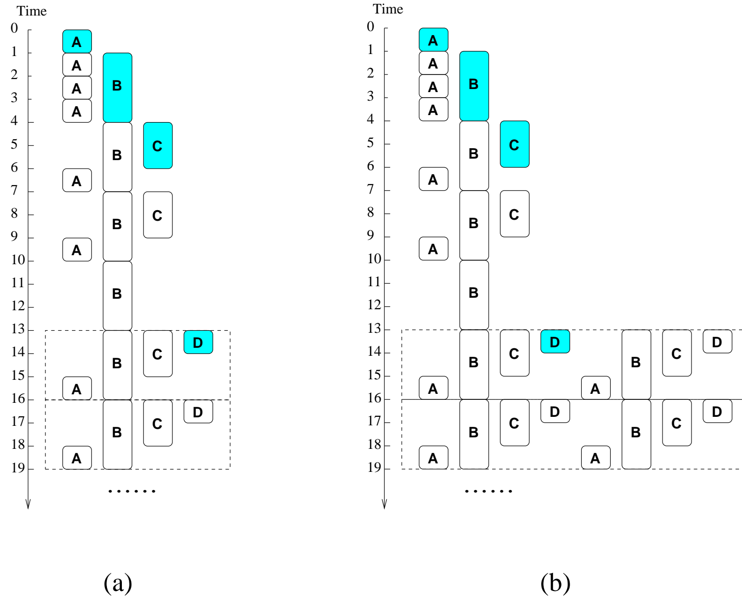

Figure 7: (a) The static schedule of the retimed DFG in Figure 6(a). (b) The static schedule of the retimed, unfolded DFG in Figure 6(b).

6, $r(B) = 4$, and $r(C) = 2$ is shown in Figure 6(a). Retiming can not achieve the optimal rate in this case, because the cycle period of G_r is bounded by the computation time of node B, which is 3 time units. The static schedule of G_r is shown in Figure 7(a). The nodes in a dashed rectangular box belong to one iteration in the retimed DFG. The shaded nodes belong to the first iteration of the original loop. Note that some nodes are pushed out of the first iteration to become prologue. The iteration bound can be achieved by unfolding the retimed DFG by unfolding factor $f = 2$. The retimed, unfolded DFG G_{rf} is shown in Figure 6(b). The static schedule of G_{rf} is shown in Figure 7(b). The iteration period of the retimed, unfolded graph is $3/2$ with a code size much smaller than the unfolded graph in Figure 4(b). Therefore, retiming (software pipelining) and unfolding can be combined together to achieve superior designs considering both timing and code size requirements.

2.5 Code Size Optimization

Since retiming and unfolding greatly expand the code size, it's necessary to reduce the code size after retiming and unfolding to further optimize the final code. A simple *for* loop and its code after applying software pipelining are shown in Figure 8(a) and Figure 8(b). The loop schedule length is reduced from four control steps to one control step for software-pipelined loop. However, the code size of software-pipelined loop is three times larger than the original code size.

```
for i = 1 to n do
  A[i] = D[i-3] + 9;
  B[i] = A[i-1] * 5;
  C[i] = A[i] + B[i];
  D[i] = C[i] * 0.2;
end
```

(a)

```
A[1] = D[-2] + 9;
B[1] = A[0] * 5;
A[2] = D[-1] + 9;
B[2] = A[1] * 5;
C[1] = A[1] + B[1];
for i = 1 to n-2 do
  A[i+2] = D[i-1] + 9;
  B[i+2] = A[i+1] * 5;
  C[i+1] = A[i+1] + B[i+1];
  D[i] = C[i] * 0.2;
end
C[n] = A[n] + B[n];
D[n-1] = C[n-1] * 0.2;
D[n] = C[n] * 0.2;
```

(b)

Figure 8: (a) The original loop. (b) The loop after applying software pipelining.

By understanding the underlying relationship between retiming and software pipelining, we found that the relationship between code size and software pipeline degree can be formulated by mathematical formula using retiming functions as shown in Theorem 2.1.

Theorem 2.1. *Let* $G_r = \langle V, E, d_r, t\rangle$ *be a retimed DFG with a given retiming function* r. *Let* n *be the number of iterations of the original loop. The prologue and epilogue can be correctly executed by conditionally executing the loop body.*

- *For prologue, executing node u whose* $r(u) = k$ *for* k *times starting from the* $(\max_u r(u) - k + 1)$*-th iteration,* $\forall u \in V$ *and* $k \geq 0$.
- *For epilogue, executing node* $u \in V$ *with retiming value* $r(u) = k$ *for* $(\max_u r(u) - k)$ *times in the last* $\max_u r(u)$ *iterations starting from the* $(n + 1)^{th}$ *iteration,* $\forall u \in V$ *and* $k \geq 0$.

Based on Theorem 2.1, the code size expansion introduced by prologue and epilogue of software-pipelined loops can be removed if the execution order of the retimed nodes can be controlled based on their retiming values. The code size reduction technique [25] uses the retiming function to control the execution order of the computation nodes in a software-pipelined loop. The relative values are stored in a counter to set the "life-time" of the nodes with the same retiming value. For node v with retiming value $r(v)$, its counter is set as the maximum retiming value minus the retiming value of node v, i.e. $p = max_u r(u) - r(v)$. We also specify that the instruction is executed only when $0 \geq p > -n$. In other words, the instruction is disabled when $p > 0$ or $p \leq -n$, where n represents the original loop counter.

Based on this understanding, code size reduction technique can remove the code in prologue and epilogue by conditionally executing the loop body using either conditional branches or predicate registers. For a processor with predicate register, an instruction *guarded* by a predicate register is conditionally executed depending on the value of the predicate register. If it is "true", the instruction is executed. Otherwise, the instruction is disabled. Each register is initialized to a different value depending on its retiming value, and is decreased by one for each iteration. After applying code size reduction, each iteration executes only the static schedule of the loop body after applying code size reduction. Code size reduction can be generally applied to any processors with or without predicate registers, and achieves smaller code size. For the details, please refer to the work published in [25].

3 Design Space Minimization

In this section, we present the the theoretical foundation of the integrated framework for design optimization and space minimization. With the understanding of the relationships among retiming function, unfolding factor and iteration period, we show that the theorem presented in this section can be applied effectively to reduce the space and computation cost of design exploration, and also the design quality can be improved considering timing and code size.

Design space is a n-dimensional space where each point represents a design solution and each dimension represents a design parameter. The design parameters we considered here are unfolding factor, retiming function, iteration period and code size. We would like to find the

minimal configuration of functional units or processors to execute an application satisfying the timing and code size requirement. Let r be the retiming value, f the unfolding factor, and S the number of points in design space. When we combine retiming and unfolding technique together to optimize a design, the design space is drastically expanded to $S \times r \times f$. Noticing that many of the unfolding factors cannot achieve the timing constraint, and thus most of the design points in the design space are infeasible, we show that the number of feasible unfolding factors given iteration period constraint can be a very small set. Therefore, the design cost of searching all the design options can be significantly reduced.

Theorem 3.1 shows the computation of the minimum cycle period of a data flow graph with given unfolding factor [4]. Note that the minimum cycle period can be achieved via retiming and unfolding with given unfolding factor.

Theorem 3.1. *Let $G = \langle V, E, d, t \rangle$ be a data flow graph, f a given unfolding factor. Let c be the cycle period of G. There exists a legal static schedule without resource constraints iff $c/f \geq B(G)$ and $c \geq max_v t(v)$, $\forall v \in V$. Thus, the minimum cycle period given unfolding factor f is $c_{min}(G, f) = max(max_v t(v), \lceil f \cdot B(G) \rceil)$.*

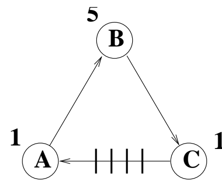

Figure 9: An exemplary DFG G with $B(G) = 7/4$.

Consider a simple DFG shown in Figure 9, the computation time of the circuit is $1+5+1 = 7$ time units. The delay count is 4. Thus, the iteration bound $B(G) = 7/4$. If the unfolding factor is $f = 2$, we can directly find the minimum feasible cycle period as $c_{min}(G, f) = max(max_v t(v), \lceil f \cdot B(G) \rceil) = max(5, \lceil 2 \cdot 7/4 \rceil) = max(5, 4) = 5$. That is, the minimum cycle period can be achieved via retiming and unfolding with unfolding factor 2 is five.

Given a timing constraint, one way to find the minimum feasible unfolding factor is to try out all the candidate unfolding factors by performing real scheduling on all the design points. We

propose to further verify the feasibility of an unfolding factor *before* performing real scheduling. In the following theorem, we show that the verifi cation can be done very quickly, so that many infeasible unfolding factors can be eliminated immediately without performing real scheduling.

Theorem 3.2. *Let* $G = \langle V, E, d, t \rangle$ *be a data flow graph,* f *a given unfolding factor, and* P *a given iteration period constraint. The following statements are equivalent:*

1. *There exists a legal static schedule of unfolded graph* G_f *with iteration period less than equal to* P.
2. $B(G) \cdot f \leq c_{min}(G, f) \leq P \cdot f$.

Proof. The left hand side of the equation follows directly from Theorem 3.1. For a data flow graph G with a given unfolding factor f, the minimum iteration period of G via retiming and unfolding can be computed from the minimum cycle period as $c_m in(G, f)/f$. To obtain a legal static schedule with iteration period less than equal to P, we must have $c_m in(G, f)/f \leq P$. Thus, the right hand side of the equation is proved. □

Consider the example in Figure 9 again. Assume that we would like to achieve an iteration period $P = 7/3$, and we would like to know what unfolding factors are possible for achieving this requirement. For instance, how about unfolding factor $f = 2$? The minimum feasible cycle period of the unfolded graph can be computed as $c_{min}(G, f) = 5$ based on Theorem 3.1, while the iteration period requirement for the unfolded graph is $P \cdot f = 14/3$. The left hand side of the inequality in Theorem 3.2 is not satisfi ed. That is to say, the iteration period constraint cannot be achieved with unfolding factor $f = 2$. Thus, we don't even need to try scheduling with unfolding factor 2. Considering the retiming and scheduling efforts we saved by just verifying an unfolding factor, the design cost can be reduced evidently for every unfolding factor we eliminated by using Theorem 3.1 and Theorem 3.2.

4 Algorithms

In this section, we describe four different design optimization and space exploration algorithms assuming there are two types of functional units (or processors). Each algorithm employs dif-

ferent techniques to approach the design space exploration problem. We will also compare their computation costs.

Four algorithms are: STDu, STDur, IDOMe and IDOMeCR. Algorithm 4.1 (STDu) is a standard method which uses unfolding to optimize the schedule length and search the design space. Given a data flow graph G, the iteration period constraint P, and the memory constraint M. Algorithm STDu generates the unfolded graph G_f for each unfolding factor $1 \leq f \leq f_{max}$, where f_{max} is derived from the memory constraint. For each unfolded graph, the algorithm computes the upper bound for each type of functional units using As Soon As Possible scheduling. Then, the algorithm schedule the DFG with each possible configuration using list scheduling. Algorithm STDu exhaustively searches the space for f_{max} unfolded graphs. Note that using unfolding only may not be able to find a solution satisfying iteration period constraint even with unlimited functional units, because the unfolding factor is upper bounded by memory constraint.

Algorithm STDur is another standard method that applies retiming to optimize schedule length of the unfolded graphs. By using retiming, the cycle period of an unfolded graph is optimized before scheduling. Therefore, algorithm STDur is able to find more feasible solution than algorithm STDu. However, retiming extends the search space and increase the computation cost.

Algorithm 4.2 (IDOMeCR) shows the algorithm of IDOM. The algorithm computes the minimum feasible cycle period c_{min} using Theorem 3.1. Then, it eliminates the infeasible unfolding factors from a set of unfolding factors $1 \leq f \leq f_{max}$ using Theorem 3.2, which selects a minimum set of candidate unfolding factors $F = \{f : c_{min}/f \leq P\}$. It means that we do not need to generate and schedule all the unfolded graphs. Therefore, the computation cost of design space exploration is significantly reduced. In step 5, it performs extended retiming instead of the traditional one to find optimal cycle period for an unfolded graph. In addition to the computation of the upper bound of the functional units, the lower bound of each type of functional units is also computed using latency bound $\lfloor P \cdot f \rfloor$ in step 6. It further reduces the search space. Finally, the algorithm schedules the retimed unfolded graphs and performs code size reduction. A less powerful IDOM algorithm, called IDOMe, will not apply code size reduction as in IDOMeCR.

Algorithm 4.1 Design Space Exploration Algorithm of the Standard Approach Using Only Un-

folding (STDu)

Input: DFG $G = \langle V, E, d, t \rangle$, iteration period constraint P and code size constraint M.

Output: The minimum configuration $(fu1, fu2)$.

1. Compute the upper bound on unfolding factor $f_{max} = \lfloor M/|V| \rfloor$.
2. For each $f \in \{F : 1 \leq f \leq f_{max}\}$ in increasing order, compute the unfolded graph G_{f_i}.
3. Compute the upper bound of functional units $(fu1_{max}, fu2_{max})$.
4. Schedule G_{f_i} for each configuration in $S_i = \{(fu1, fu2) : 1 \leq fu1 \leq fu1_{max}, 1 \leq fu2 \leq fu2_{max}\}$ in increasing order, until a schedule satisfying P and M is found.

The computation cost of the design space exploration algorithms can be compared using the complexity of list scheduling as a unit of the computation cost. The complexity of list scheduling is $O(|V|+|E|)$. The complexity of unfolding is $O(f|V|+f|E|)$. Let F be a set of feasible unfolding factors, and S_T be the set of points in the design space to be searched by an algorithm. Here we compute the search cost of an algorithm in terms of the number of times list scheduling is applied to the original graph. The computation cost of STDu algorithm can be estimated as the summation of unfolding cost (C_{uf}) and scheduling cost (C_{sche}) for the unfolded graphs, i.e., $C_{uf} + C_{sche} = \sum_{f_i \in F} f_i + \sum_{f_i \in F} f_i |S_T|$.

The search cost of the other algorithms can be computed in the similar way. The computation cost of retiming an unfolded graph is $C_{tr} = \sum_{f_i \in F} {f_i}^2 |V| + \sum_{f_i \in F} {f_i}^2 |V| \log(f_i |V|)$. The search cost of STDur algorithm can be computed as $C_{uf} + C_{tr} + C_{sche}$.

The search cost of IDOM is the summation of the cost of unfolding, scheduling, and extended retiming (C_{er}), i.e., $C_{uf} + C_{er} + C_{sche}$. Since the size of unfolding factor set $|F|$ and the search space $|S_T|$ are signifi cantly reduced in IDOM approach, the computation costs of unfolding and scheduling are also greatly reduced. The computation cost of extended retiming on an unfolded graph can be computed as $C_{er} = {f_i}^2 |V|$, which is smaller than traditional retiming cost. The search cost of each algorithm will be computed and compared using a design space exploration example in the next section.

Algorithm 4.2: Design Exploration Algorithm of IDOM Approach Using Unfolding, Retiming, and Code Size Optimizations (IDOMeCR)

Input: DFG $G = \langle V, E, d, t \rangle$, a set of functional unit types $U = \{u_1, u_2, \ldots, u_n\}$, iteration period constraint P, and code size constraint M.

Output: The minimal configuration and the corresponding design parameter values: unfolding factor, retiming function, iteration period, and code size.

1. Compute the iteration bound $B(G)$. If $P > B(G)$, go to **Step 13.**
2. Compute the upper bound on unfolding factor $f_{max} = min(\lfloor M/|V| \rfloor, lcm(D(\ell)))$, for all $\ell \in G$.
3. For each $1 \leq f_i \leq f_{max}$, compute the minimum feasible cycle period c_{min}. (Theorem 3.1)
4. Produce the minimum set of feasible unfolding factors: $F = \{f_i : c_{min}/f_i \leq P\}$. (Theorem 3.2)
5. Generate the unfolded graph G_{f_i} for $f_i \in F$.
6. Apply the extended retiming on G_{f_i} with retiming function r_i to obtain the unfolded, retimed graph G_{f_i,r_i}.
7. Compute the lower bound of functional units $(|u_1|_{min}, |u_2|_{min}, \ldots, |u_n|_{min})$ with latency bound $\lfloor P \cdot f_i \rfloor$.
8. Compute the upper bound of functional units $(|u_1|_{max}, |u_2|_{max}, \ldots, |u_n|_{max})$ using As Soon As Possible Scheduling.
9. Schedule the unfolded, retimed data flow graph G_{f_i,r_i} on a set of configurations S, where $S = \{(|u_1|, |u_2|, \ldots, |u_n|) : |u_1|_{min} \leq |u_1| \leq |u_1|_{max}, |u_2|_{min} \leq |u_2| \leq |u_2|_{max}\}$ with increasing order until a schedule satisfying iteration period constraint P is found.
10. Apply the code size reduction. (Theorem 2.1)
11. Return the configuration and the values of design parameters if both iteration period and code size constraints are satisfied. Exit.
12. Go back to **Step 5.**
13. Report "Infeasible" if no feasible configuration is found. Exit.

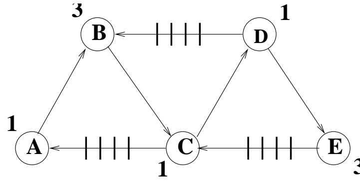

Figure 10: Data flow graph of an example.

5 Example

In this section, we use a simple example to illustrate the design exploration process using the Integrated Framework of Design Optimization and Space Minimization, and compare the effi ciency and quality of different algorithms. Figure 10 shows an exemplary DFG. The iteration bound of this DFG G is $B(G) = 5/4$. Given the iteration period constraint $P = 4/3$, and the code size constraint $M = 25$ instructions, the design task is to fi nd the minimum confi guration satisfying the requirements for an architecture with 2 different types of functional units (or processors).

Table 1 compares four algorithms in terms of the size of search space and the quality of outcomes. Column "Search Points" shows the number of points to be explored by the algorithms. Column "Search Cost" lists the computation costs of searching the design space and performing the optimization using the computation described in Section 4. Column "Solutions" shows the resulting design solutions. The parameters displayed in this column are: the iteration period ("Iter. Period"), unfolding factor ("uf"), number of various types of processors ("#fu2" and "#fu2"), and the resulting code size. Assuming that the code size reduction is performed on class 3 processors in the IDOMeCR algorithm. If an entry under column "Solutions" is marked by an "F", it indicates that the corresponding algorithm cannot fi nd a feasible solution. In the following, we explain the various design approaches with the example.

For the STDu algorithm, the upper bound of unfolding factor imposed by code size constraint is $f_{max} = \lfloor M/|V| \rfloor = 5$. The upper bound on the number of functional units is obtained from ASAP schedule of an unfolded graph. It exhaustively searches all the possible confi gurations within the upper bound for all the unfolding graphs. The total number of design points is 55 in this case. Even after computing with all the design choices in the design space, this method cannot fi nd a feasible solution satisfying the iteration period constraint. It shows that using unfolding

Algorithms	**Search Points**	**Search Cost**	**Solutions**				
			uf	**#fu1**	**#fu2**	**iter. per.**	**code size**
STDu	55	230	F	F	F	F	F
STDur	117	660	4	8	8	5/4	F
IDOMe	1	51	3	3	5	4/3	F(90)
IDOMeCR	**1**	**51**	**3**	**3**	**5**	**4/3**	**21**

Table 1: The search costs and outcomes of four different design exploration methods.

hardly finds any solution for tight iteration period and code size constraints.

For the STDur algorithm, various retiming functions increases the number of search points to be 117. The algorithm find a feasible configuration of 8 fu1 and 8 fu2 with unfolding factor f=4. Although the schedule length is satisfied, the code size constraints is violated. Thus, no feasible solution can be produced.

The fourth and fifth rows of Table 1 show the features of IDOMe and IDOMeCR algorithms. The minimum cycle period, and thus iteration period, for each unfolding factor is computed using Theorem 3.1. Then, the iteration period is 2 for f=1. It is 3/2 for f=2, 4/3 for f=3 5/4 for f=4 and 7/5 for f=5. Since the iteration period constraint ($\leq 4/3$) can be satisfied only for unfolding factors f=3 or f=4, we can eliminate the unfolding factors 1, 2, and 5 (Theorem 3.2). Furthermore, since the low bound and upper bound on the number of processors appear to be the same, the design space is further reduced to just 1 point. The minimum configuration found by IDOM is 3 fu1 and 5 fu2 with unfolding factor 3, which is better than that found by the STDur algorithm.

The code size generated by the IDOMe algorithm is 90 instructions. It exceeds the code size constraints. We list the code size in the parenthesis to show the effectiveness of code size optimization used in the IDOMeCR algorithm. The code size is reduced from 90 instructions to 21 instructions using IDOMeCR algorithm. Then, the memory constraint can be satisfied.

6 Experimental Results

To demonstrate the performance and quality of IDOM algorithm, we conduct a series of experiments on a set of DSP benchmarks. Table 2 shows the experimental results. The experiments are conducted using two different design exploration algorithms, one is STDur, the other is IDOMeCR. To make the cases more complicated for design space exploration, we apply different slow down factors to the original circuits [13,15], and choose the computation times for additions and multiplications arbitrarily. We also assume the code size constraint is $M = 5(|V| + 1)$ for all the cases. The measurement of code size is based on the number of instructions in the compiled code for a simulated processor with predicate register similar to TI's TMS320C6x.

For Biquad Filter ("Biquad"), we assume that addition takes 1 time unit and multiplication 4 time units. The resulting circuit has an iteration bound $B(G) = 3/2$. Given an iteration period constraint $P = 5/3$. an confi guration satisfying the iteration period constraint is found by STDur. It has 3 adders and 16 multipliers with unfolding factor f=4. The resulting iteration period is 3/2. However, the code size exceeds the memory constraint. For the illustration purpose, we still show the resulting code size generated by the STDur algorithm in the parenthesis. For the same case, IDOMeCR fi nd a smaller confi guration of 3 adders and 10 multipliers with unfolding factor f=3. The resulting schedule satisfi es both iteration period and memory constraints. Furthermore, the search cost of IDOMeCR is only 4% of that using STDur, as shown in column "Ratio".

The experimental settings for the other cases are described in the following. For Partial Differential Equation ("DEQ"), assume that an addition operation takes 1 time unit, and a multiplication operation takes 3 time units. The resulting iteration bound is $B(G) = 8/5$. The iteration period constraint is $P = 7/4$. For Allpole Filter ("Allpole"), an addition operation takes 2 time units, and a multiplication operation takes 5 time units.the iteration bound is $B(G) = 18/5$, and the iteration period constraint is given as $P = 15/4$. For 5th Order Elliptic Filter ("Elliptic"), an addition takes 1 time unit, and a multiplication takes 5 time units, the iteration bound is $B(G) = 18/5$, and the iteration period constraint is $P = 15/4$. For 4-Stage Lattice Filter ("4-Stage"), an addition takes 1 time unit, and a multiplication 6 time units, the iteration bound is $B(G) = 7/4$, and the iteration period constraint is $P = 9/5$.

Applying the algorithms on all these experimental cases clearly yields the conclusion that

IDOMeCR significantly reduces the search cost of design exploration process compared to the standard method. Its search cost is only 3% of that using STDur on average. Furthermore, IDOMeCR always find the minimal configurations for all the benchmarks.

Benchmarks	**Search Points**	**Search Cost**	**Ratio**	**Solutions**				
				uf	**#add.**	**#mult.**	**iter. per.**	**code size**
Biquad(std.)	228	2165		4	4	16	3/2	F(80)
Biquad(ours)	**4**	**87**	**4%**	**3**	**3**	**10**	**5/3**	**28**
DEQ(std.)	486	6017		5	10	18	8/5	F(110)
DEQ(ours)	**6**	**120**	**2%**	**3**	**5**	**15**	**5/3**	**37**
Allpole(std.)	510	7957		5	10	10	18/5	F(150)
Allpole(ours)	**6**	**156**	**2%**	**3**	**10**	**3**	**11/3**	**51**
Elliptic(std.)	694	19062		F	F	F	F	F
Elliptic(ours)	**2**	**142**	**0.7%**	**2**	**6**	**9**	**5**	**76**
4-Stage(std.)	909	15329		F	F	F	F	F
4-Stage(ours)	**55**	**640**	**4%**	**4**	**7**	**33**	**7/4**	**112**

Table 2: The design solutions generated by STDur and IDOMeCR algorithms.

7 Conclusion

In this paper, we presented an Integrated Framework of Design Optimization and Space Minimization (IDOM) to find the minimum configuration satisfying the schedule length and memory constraints. The properties of the unfolded retimed data flow graph are studied to produce the minimum set of feasible unfolding factors without performing real scheduling. The relationships among unfolding factor, retiming function, iteration period and code size are stated in our theorems. We found that after we clearly understood the intrinsic relationships between the design parameters, a huge number of design points are immediately proved to be infeasible. Therefore, the design space to be explored is greatly reduced. We integrated several optimization techniques, i.e., retiming, unfolding and code size reduction to produce superior designs. The experimental results show that the search cost of IDOM is only 3% of the standard method on average.

References

[1] E. Altman, R. Govindarajan, and G. Gao. Scheduling and mapping: Software pipelining in the presence of structure hazards. In *Proc. ACM SIGPLAN 1995 Conf. on Programming Language Design and Implementation*, pages 139–150, Jun. 1995.

[2] C. Chantrapornchai, E. H.-M. Sha, and X. S. Hu. Effi cient acceptable design exploration based on module utility selection. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 19(1):19–29, Jan. 2000.

[3] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha. Rotation scheduling: A loop pipelining algorithm. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 16(3):229–239, Mar. 1997.

[4] L.-F. Chao and E. H.-M. Sha. Static scheduling for synthesis of DSP algorithms on various models. *Journal of VLSI Signal Processing*, 10:207–223, 1995.

[5] L.-F. Chao and E. H.-M. Sha. Scheduling data-flow graphs via retiming and unfolding. *IEEE Trans. on Parallel and Distributed Systems*, 8(12):1259–1267, Dec. 1997.

[6] S. Chaudhuri, S. A. Blythe, and R. A. Walker. A solution methodology for exact design space exploration in a three dimensional design space. *IEEE Trans. on VLSI Systems*, 5(1):69–81, Mar. 1997.

[7] A. Darte and G. Huard. Loop shifting for loop compaction. *International Journal of Parallel Programming*, 28(5):499–534, Oct. 2000.

[8] E. Granston, R. Scales, E. Stotzer, A. Ward, and J. Zbiciak. Controlling code size of software-pipelined loops on the TMS320C6000 VLIW DSP architecture. In *Proc. 3rd IEEE/ACM Workshop on Media and Streaming Processors*, pages 29–38, Dec. 2001.

[9] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitive Approach*. Morgan Kaufmann Publishers, Inc., 2nd edition, 1995.

[10] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. ACM SIGPLAN'93 Conf. on Programming Language Design and Implementation*, pages 258–267, Jun. 1993.

[11] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 207–218, Jan. 1981.

[12] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. ACM SIGPLAN'88 Conf. on Programming Language Design and Implementation*, pages 318–328, Jun. 1988.

[13] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, Aug. 1991.

[14] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.

[15] T. O'Neil, S. Tongsima, and E. H.-M. Sha. Extended retiming: Optimal scheduling via a graph-theoretical approach. In *Proc. IEEE Int'l Conf. on Acoustics, Speech, and Signal Processing*, volume 4, pages 2001–2004, Mar. 1999.

[16] K. V. Palem, R. M. Rabbah, V. J. Mooney, P. Korkmaz, and K. Puttaswamy. Design space optimization of embedded memory systems via data remapping. In *Proc. the Joint Conf. on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems (LCTES'02-SCOPES'02)*, pages 28–37, Jun. 2002.

[17] K. K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley & Sons, 1999.

[18] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. 27th IEEE/ACM Annual Int'l Symp. on Microarchitecture (MICRO)*, pages 63–74, Nov. 1994.

[19] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview and perspective. *Journal of Supercomputing*, 7(1/2):9–50, Jul. 1993.

[20] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientifi c computing. In *Proc. 14th ACM/IEEE Annual Workshop on Microprogramming*, pages 183–198, Oct. 1981.

[21] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code generation schema for modulo scheduled loops. In *Proc. 25th IEEE/ACM Annual Int'l Symp. on Microarchitecture (MICRO)*, pages 158–169, Dec. 1992.

[22] M. Renfors and Y. Neuvo. The maximum sampling rate of digital filters under hardware speed constraints. *IEEE Trans. on Circuits and Systems*, CAS-28:196–202, Mar. 1981.

[23] R. S. Sambandam and X. Hu. Predicting timing behavior in architectural design exploration of real-time embedded systems. In *Proc. 34th ACM/IEEE Design Automation Conf. (DAC)*, pages 157–160, Jun. 1997.

[24] F. Thoen, M. Corncro, G. Goossens, and H. D. Man. Software synthesis for real-time information processing systems. In *Proc. ACM SIGPLAN 1995 Conf. on Programming Language Design and Implementation*, pages 60–69, Jun. 1995.

[25] Q. Zhuge, Z. Shao, and E. H.-M. Sha. Optimal code size reduction for software-pipelined loops on dsp applications. *ACM Trans. on Embedded Computing Systems (TECS)*, 11(1):590–613, Mar. 2004.

[26] Q. Zhuge, B. Xiao, Z. Shao, E. H.-M. Sha, and C. Chantrapornchai. Optimal code size reduction for software-pipelined and unfolded loops. In *Proc. 15th IEEE/ACM Int'l Symp. on System Synthesis*, pages 144–149, Oct. 2002.