

Design Space Minimization with Timing and Code Size Optimization for Embedded DSPs *

Qingfeng Zhuge, Zili Shao, Bin Xiao, Edwin H.-M. Sha

Department of Computer Science

University of Texas at Dallas

Richardson, Texas 75083, USA

{qfzhuge, zls015000, bxiao, edsha}@utdallas.edu

Abstract

One of the most challenging problems in high-level synthesis is how to quickly explore a wide range of design options to achieve high-quality designs. This paper presents an Integrated Framework for Design Optimization and Space Minimization (IDOM) towards 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 relations among multiple design parameters, such as retiming value, unfolding factor, timing, and code size. Theorems are presented to produce a small set of feasible design choices with provable quality. IDOM algorithm is proposed to generate high-quality design by integrating performance and code size optimization techniques. The experimental results on a set of DSP benchmarks show the efficiency and effectiveness of the IDOM algorithm. It constantly generates the minimal configuration for all the benchmarks. The cost of design space exploration using IDOM is only 3% of that using the standard method.

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

1 Introduction

Embedded systems usually have strict timing requirement and limited memory resources. Architectural-level synthesis with code generation is a critical stage toward generating embedded systems with stringent requirements. 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 [1,4,7]. In this context, the design space 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 performance, optimization techniques such as unfolding and software pipelining are commonly used to improve the execution time of loops. [2, 5, 8]. The proper unfolding factor and software pipelining degree need to be chosen in architectural-level synthesis to satisfy the performance constraint. On the other hand, these optimization techniques introduce large code size expansions [10], which is not desirable due to very limited memory resources. Therefore, a good design exploration 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.

In order to find feasible design points in a large design space concerning multiple design parameters within a short time, design space minimization problem becomes a very important problem in high-level synthesis. A number of research results are published on optimization problems [8, 10] and design space exploration [1, 4, 7, 9]. However, we are unaware of any previous work that addresses the combination of various optimization techniques and their effects on the design space minimization. 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.

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. Then, based on this understanding, 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 [8, 10], 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 greatly reduces the design space and the exploration cost by exploiting the fundamental properties of retimed unfolded data flow graph and the relationship between design parameters. Second, it combines several optimization techniques, namely, unfolding [8], extended retiming [6] and code size reduction [10], to produce a superior solution satisfying schedule length and code size requirements.

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 theorems of the integrated framework for design optimization and space minimization. Section 3 provides the algorithms and computation cost for several design space exploration algorithms. An example for design optimization and space exploration is demonstrated in Section 4 to compare the size of search space, the computation cost and the quality of design solutions among four different design space exploration algorithms. In Section 4.1, we present the experimental results on a set of DSP benchmarks. Finally, concluding remarks are provided in Section 5.

2 Optimization Techniques and Design Space Minimization

In this section, we present the theoretical foundation of the integrated framework for design optimization and space minimization. We provide an overview of the basic principles of retiming and unfolding in Section 2.1. An in-depth analysis for these optimization techniques based on data flow graph model reveals underlying relations between the design parameters. Section 2.2 presents the theorems of design space minimization. It shows that the search space and search cost of design space exploration can be greatly reduced based on the properties and relations between the design parameters considering unfolding factor, retiming function and iteration period. Section 2.3 presents an code size optimization technique that reduces the code size for retimed data flow graph.

2.1 Performance Optimizations

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 without resource constraint. For example, $c(G)$ of the graph in Figure 1(a) is 2.

Given a DFG G which may contain cycles, retiming and unfolding can be used to minimize the execution time of all tasks in one iteration.

Retiming [5] 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*

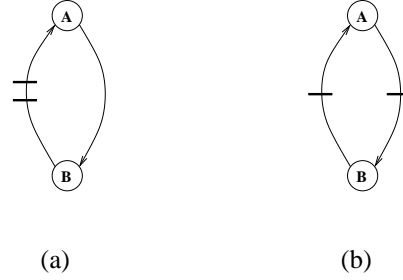


Figure 1: (a) A simple DFG. (b) The retimed DFG with $r(A) = 1$ and $r(B) = 0$.

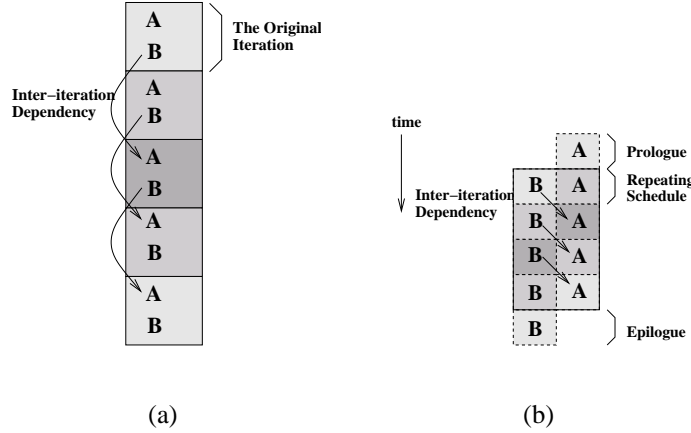


Figure 2: (a) A static schedule of original loop. (b) The pipelined loops.

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 1(b) shows the retimed DFG of Figure 1(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 1(b), the actual effect on the schedule of the new DFG is that the i^{th} copy of A is shifted up and is executed with $(i - 1)^{\text{th}}$ 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 2(a) and Figure 2(b).

In fact, every retiming operation corresponds to a software pipelining operation. When one delay is pushed forward through a node u , every copy of this node is moved up by one iteration, and the first copy of the node is shifted out of the first iteration into the prologue. With retiming function r , 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 $\max_u r(u)$, 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 1(a). Then, there is one copy of node A in the pipelined schedule as shown in Figure 2(b).

The extended retiming allows the delays to cut the execution time of a multiple-cycle node to achieve optimal schedule length [6]. Due to the space limitation, we will not illustrate the extended retiming in details.

Unfolding [3, 8] is another effective technique for improving the average cycle period of a static schedule. The original DFG G is unfolded f times, so the unfolded graph G_f consists of f copies of the original node set. Thus, a schedule with unfolding factor f contains f iterations of the original DFG.

For any DFG G , the average computation time of an iteration is called the *iteration period* of the graph. The instruction-level parallelism between the iterations in an unfolded loop helps to improve the iteration period $P = c(G_f)/f$. For a DFG containing a loop, the iteration period is bounded from below by the iteration bound of the graph which is defined 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, respectively, in a cycle ℓ .*

It is clear that unfolding will increase code size by a factor of f . We want to find the minimum f with retiming r so the iteration period of the resultant loop schedule is optimal. But it is very likely that such an optimal f is too large for the program to fit into a small-size on-chip memory. Therefore, we need to explore what will be the good f , r for satisfying both iteration period and memory constraints.

Without retiming, the rate optimal unfolding factor is upper bounded by the least common multiple of the delay counts of all cycles in DFG. That is, if we unfolded the DFG G by $f = \text{lcm}(D(\ell)), \forall \ell \in G$, we can always get a DFG with the optimal execution rate $P(G_f) = T(\ell_{cr})/D(\ell_{cr})$. Consider the data flow graph G in Figure 3(a) with iteration bound $B(G) = 6/4 = 3/2$ and cycle period $c(G) = 7$. After unfolding by 4, the unfolded graph G_f in Figure 3(b) achieves the iteration period $P(G_f) = 6/4 = 3/2$ which is optimal.

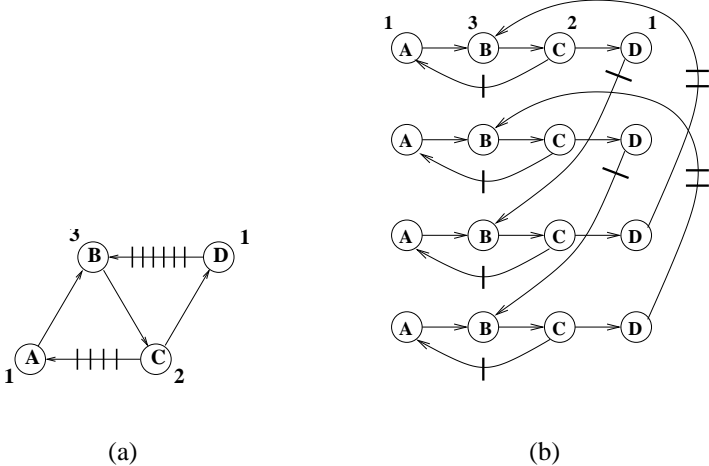


Figure 3: (a) A data flow graph. (b) The unfolded graph.

Since the unfolded graph is f times larger than the original DFG, we would like to find the minimum unfolding factor for achieving the optimal execution rate. By combining retiming and unfolding, the minimum unfolding factor can be reduced. For the example in Figure 3(a), the minimum unfolding factor is 2. In Figure 4(a), we show the retimed graph G_r . Note that traditional 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. After unfolding G_r by 2, the unfolded retimed DFG G_{fr} shown in Figure 4(b) achieves the optimal iteration period $P(G_{fr}) = 3/2$. Therefore, retiming (software pipelining) and unfolding are usually combined in design optimization.

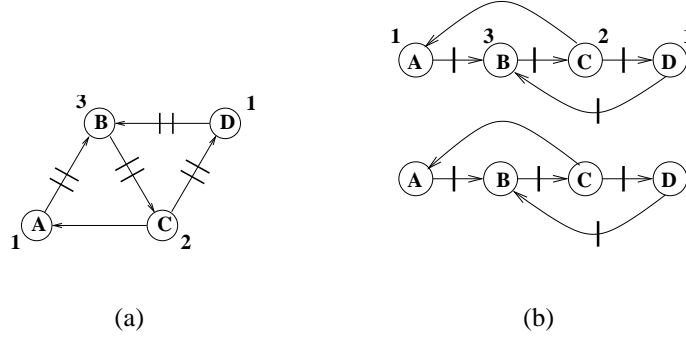


Figure 4: (a) The retimed graph. (b) The unfolded retimed graph.

2.2 Design Space Minimization

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. Let r be the retiming value, f the unfolding factor, S the number of points in design space. The design space considering various retiming value and unfolding factor is $r \times f \times S$. In this section, we present theorems for finding the minimum set of unfolding factors based on the fundamental relationships between the design parameters in a unfolded retimed graph, so that the search space and search cost can be greatly reduced.

Theorem 2.1 states the relationship between unfolding factor and corresponding minimum feasible cycle period. It shows the necessary and sufficient condition of finding a legal static schedule [2].

Theorem 2.1. *Let $G = \langle V, E, d, t \rangle$ be a given data flow graph, $f \in \mathbb{Z}^+$ an unfolding factor and $c \in \mathbb{R}$ a cycle period, 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, given unfolding factor f , the minimum cycle period $c_{\min}(G_f) = \max(\max_v t(v), \lceil f \cdot B(G) \rceil)$.*

Consider a simple DFG shown in Figure 5, 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 $f = 2$, we can directly find the minimum feasible cycle period as $c_{\min}(G_f) = \max(5, 4) = 5$. Thus, it proves that it is impossible to find a schedule with cycle period < 5 with $f = 2$.

On the other hand, given an iteration period constraint, we can quickly know the feasible

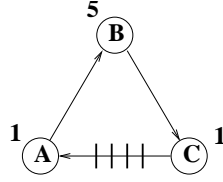


Figure 5: An exemplary DFG G with $B(G) = 7/4$.

unfolding factors that are possible to produce a schedule that satisfies the constraint as stated in the following theorem.

Theorem 2.2. *Let $G = \langle V, E, d, t \rangle$ be a data flow graph, f an unfolding factor, 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. $B(G) \cdot f \leq c_{\min}(G, f) \leq P \cdot f$.

As to the previous example, assume that we want to achieve an average iteration period $P = 7/3$, and we would like to know what unfolding factor is possible for achieving this requirement. For instance, how about unfolding factor $f = 2$? Since the cycle period of the unfolded graph is $c_{\min}(G_f) = 5$ from Theorem 2.1. We find that $c_{\min}(G_f) > P \cdot f = 14/3$. Therefore, the iteration period constraint cannot be achieved with unfolding factor $f = 2$. By using Theorem 2.1 and Theorem 2.2, we can immediately eliminate many infeasible ones *without performing real scheduling*, and thus significantly reduce the search space and search cost.

Since unfolding multiplies the code size, we would like to find the minimum possible unfolding factor that can achieve the iteration period constraint via retiming. We can use these two theorems to easily check (without real scheduling) if an unfolding factor is possible to satisfy timing requirement or not.

2.3 Code Size Optimization

Since retiming and unfolding greatly expand the code size, it's possible that the generated code is too large to be fit into the on-chip memory of embedded DSP processors. A simple *for* loop and its code after applying software pipelining are shown in Figure 6(a) and Figure 6(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.

<pre> 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 </pre>	<pre> 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; </pre>
(a)	(b)

Figure 6: (a) The original loop. (b) The loop after applying software pipelining.

In our research work, we study the underlying relationship between retiming and software pipelining, and found that the size of code expansion is closely related to the retiming function. As we found in Theorem 2.3, the relationship between code size and software pipeline degree can be formulated by mathematical formula using retiming functions.

Theorem 2.3. *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 $(n + 1)^{\text{th}}$ iteration, $\forall u \in V$ and $k \geq 0$.

Based on Theorem 2.3, 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 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 CRED. 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 [10].

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

Four algorithms are: STDu, STDur, IDOMe and IDOMeCR. Algorithm 3.1 (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 3.2 (IDOMeCR) shows the algorithm of IDOM. The algorithm computes the minimum feasible cycle period c_{min} using Theorem 2.1. Then, it eliminates the infeasible unfolding factors from a set of unfolding factors $1 \leq f \leq f_{max}$ using Theorem 2.2, 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 3.1 Design Space Exploration Algorithm of the Standard Approach Using Only Unfolding (STDu)

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

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

Step 1. Compute the upper bound on unfolding factor $f_{max} = \lfloor M/|V| \rfloor$.

Step 2. For each $f \in \{F : 1 \leq f \leq f_{max}\}$ in increasing order, compute the unfolded graph G_{f_i} .

Step 3. Compute the upper bound of functional units $(fu1_{max}, fu2_{max})$.

Step 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

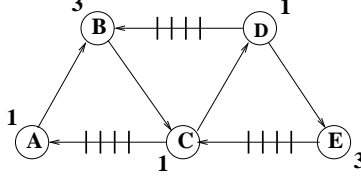


Figure 7: Data flow graph of an example.

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 significantly 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.

4 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 efficiency and quality of different algorithms. Figure 7 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 find the minimum configuration 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

Algorithm 3.2 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 $U = \{u_1, u_2, \dots, u_n\}$, iteration period constraint P , and code size constraint M .

Output:

1. A set of feasible configurations S ;
2. The minimum configuration S_{min} ;
3. The schedule $Schedule$, the minimum unfolding factor f_{min} , the iteration period $IterPer$, and the code size $CodeSize$, when the minimum configuration is used;
4. Or report “Infeasible”.

Step 1. $S_{min} \leftarrow \emptyset$, $f_{min} \leftarrow 1$, $IterPer \leftarrow 0$, $CodeSize \leftarrow 0$.

Step 2. Compute the upper bound on unfolding factor $f_{max} = \lfloor M/|V| \rfloor$.

Step 3. For each $1 \leq f_i \leq f_{max}$, compute the minimum feasible cycle period c_{min} . (Theorem 2.1)

Step 4. Select the feasible unfolding factor: $F = \{f_i : c_{min}/f_i \leq P\}$. (Theorem 2.2)

Step 5. Generate unfolded graph G_{f_i} for each $f_i \in F$.

Step 6. Apply extended retiming on G_{f_i} with retiming function r_i . Get the unfolded retimed graph G_{f_i, r_i} .

Step 7. Compute the lower bound of functional units $(|u_1|_{min}, |u_2|_{min}, \dots, |u_n|_{min})$ with latency bound $\lfloor P \cdot f_i \rfloor$.

Step 8. Compute the upper bound of functional units $(|u_1|_{max}, |u_2|_{max}, \dots, |u_n|_{max})$ using As Soon As Possible Scheduling.

for all $f_i \in F$ **do**

for all Configuration S_i in $S = \{(|u_1|, |u_2|, \dots, |u_n|) : |u_1|_{min} \leq |u_1| \leq |u_1|_{max}, |u_2|_{min} \leq |u_2| \leq |u_2|_{max}\}$ **do**

Step 9. Schedule G_{f_i, r_i} on S in an increasing order, until a $Schedule$ satisfying iteration period constraint P is found with configuration S_i .

Step 10. Apply code size reduction. (Theorem 2.3)

Step 11. If $CodeSize \leq M$, $S_{min} \leftarrow S_i$, $f_{min} \leftarrow f_i$, $IterPer \leftarrow |Schedule|/f_{min}$. Report design solution and exit.

end for

end for

Step 12. If $S_{min} = \emptyset$, report “Infeasible”.

the optimization using the computation described in Section 3. 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 (“#fu1” 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 find a feasible solution. In the following, we explain the various design approaches with the example.

Algorithms	Search	Search	Solutions				
	Points	Cost	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.

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 configurations 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 find a feasible solution satisfying the iteration period constraint. It shows that using unfolding 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 2.1. 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 2.2).

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.

4.1 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 [5, 6], 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 with TI's TMS320C6x. The experimental results show that IDOM can generate a better design solution with much less computation cost.

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 $P = 5/3$. a configuration 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 find a smaller configuration of 3 adders and 10 multipliers with unfolding factor $f=3$. The resulting schedule satisfies 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.

5 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 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. We believe that this fundamental study is essential to make the design space exploration to be more efficient.

References

- [1] C. Chantrapornchai, E. H.-M. Sha, and X. S. Hu. Efficient 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.
- [2] 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.
- [3] 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.
- [4] 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.
- [5] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, Aug. 1991.
- [6] T. O’Neil, S. Tongsimma, 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.
- [7] 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.

- [8] K. K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley & Sons, 1999.
- [9] 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.
- [10] Q. Zhuge, E. H.-M. Sha, and C. Chantrapornchai. CRED: Code size reduction technique and implementation for software-pipelined applications. In *Proc. IEEE Workshop of Embedded System Codesign (ESCODES)*, pages 50–56, Sept. 2002.