Reducing the code size of retimed software loops under timing and resource constraints

Noureddine Chabini
Department of Electrical and Computer Engineering, Royal Military College of Canada
P.B. 17000, Station Forces, Kingston, ON, Canada, K7K 7B4, Email: chabini@rmc.ca

Wayne Wolf
Department of Electrical Engineering, Princeton University
Eng. Quadrangle, Olden Street, Princeton, NJ, USA, 08544, Email: wolf@princeton.edu

Abstract: Retiming has been originally proposed as an optimization technique for clocked sequential digital circuits. It has been successfully applied for optimizing loops during the compilation of loop-intensive programs. After applying a retiming, the original loop transforms to another loop which is preceded by a segment of code called *prologue* and is followed by a segment of code called *epilogue*. To optimize a loop, there are many possible retimings that allow to achieve the same value of the objective function. Depending on the retiming used, the number of operations in the *prologue* and *epilogue* can increase or decrease. Decreasing the code size for retimed loops is of great importance in particular for memory-constrained system-on-chip and embedded systems. It has also an impact on power dissipation. This paper addresses the problem of reducing the code size for retimed software loops under timing and resource constraints. We mathematically formulate this problem and develop algorithms to optimally solve it. Experimental results are also provided.

Keywords: Code Size; Loops; Retiming; Timing; Resource; Embedded Systems; Power.

1 INTRODUCTION

While it was proposed originally for optimizing synchronous sequential digital circuits, retiming[9] has been successfully used for optimizing loops as well[1,7,8]. Retiming moves registers to achieve a certain objective function.

There is a relationship between a register in the context of hardware implementations and an iteration in the context of software implementations. In the context of software implementations, moving for instance one register from the inputs of an operator to its outputs transforms to: *i*) delaying by one iteration the consummation of the result produced by this operator, and *ii*) advancing by one iteration the consummation of the operands of that operator.

Applying retiming on a single loop leads to another code composed by three consecutive parts in the following order. The first one is a segment of code called *prologue*. The second one is a new loop which runs faster than the former one when the goal from retiming is optimizing timings. The third and last one is a segment of code called *epilogue*. The new loop can start executing once the execution of the *prologue* has terminated. The execution of the *epilogue* can begin once the new loop terminates executing.

There is more than one way to retime a loop for achieving a certain target. Depending on the retiming used, the code size of the *prologue* and *epilogue* can increase or decrease. Reducing the size of that code is very important in the case of embedded systems as well as of system-on-chip[1,3]. Both of these two kinds of systems have constraints on the memory size, and hence the code size must be reduced for them. The size of the code has also an implicit impact on both the execution time as well as the power consumption.The problem of reducing the code size has been widely addressed in the literature[2,4,5], but only few papers have recently addressed this problem when techniques like retiming has been used[1,3].

In the rest of this paper, we mean by "code size" the number of operations in the *prologue* and *epilogue* after retiming a loop. We address the problem of reducing the code size for retimed loops under timing and resource constraints. We formulate this problem mathematically and develop algorithms to optimally solve it. We provide a polynomial run-time algorithm to optimally solve the problem for the case of unlimited resources. In its general form, our target problem becomes NP-hard in the case of limited resources since to solve it, one needs to solve the problem of scheduling under timing and resource constraints which is already known as an NP-hard problem in its general form. We also propose an exact but not polynomial run-time algorithm to solve this latter version of our target problem.

There are many real-life examples of loop-intensive applications where the loops can be modeled as *for-type* loops. Moreover, the body of these loops does not contain conditional statements like *if-then-else*, etc. Such examples include digital filters like the correlator, the finite impulse response, and the infinite impulse response. In this paper, we focus on solving our target problem for these class of applications. This kind of applications is at the heart of many concrete embedded systems.

The rest of this paper is organized as follows. Section 2 shows how we model a loop. In Section 3, we give an introduction to basic retiming and some of its related background required for our proposed approach. In Section 4, we define valid periodic schedules that we target in this paper and propose an algorithm to compute them. We develop in Section 5 an algorithm to generate the transformed code after applying a retiming on a single *for*-loop. In

Section 6, we develop a polynomial-time exact method for computing a retiming leading to a small code size for the case of unlimited resources. In Section 7, we propose an exact but not polynomial-time algorithm to compute a retiming leading to a small code for the case of limited resources. Experimental results and conclusions are given in Sections 8 and 9.

2 CYCLIC GRAPH MODEL

In this paper, we are interested in *for-type* loops as the one in Fig. 1 (a). We assume that the body of the loop is constituted by a set of computational and/or assignment operations only (i.e., no conditional or branch instructions like for instance *if-then-else* is inside the body).

We model a loop by a directed cyclic graph $G = (V, E, d, w)$, where V is the set of operations in the loop's body, and E is the set of edges that represent data dependencies. Each vertex v in V has a non-negative integer execution delay $d(v) \in N$, where N is the set of non-negative integers. Each edge $e_{u,v} \in E$, from vertex $u \in V$ to vertex $v \in V$, has a weight $w(e_{u,v}) \in N$, which means that the result produced by u at any iteration i is consumed by v at iteration $(i + w(e_{u,v}))$.

Fig. 1 presents a simple loop and its directed cyclic graph model. For Fig. 1 (b), the execution delay of each operation v_i is specified as a label on the left of each node of the graph, and the weight $w(e_{u,v})$ of each arc $e_{u,v} \in E$ is in bold. For instance, the execution delay of v_1 is 1 unit of time and the value 2 on the arc e_{v_1,v_2} means that operation v_2 at any iteration i uses the result produced by operation v_1 at iteration $(i - 2)$.

3 INTRODUCTION TO BASIC RETIMING

Let $G = (V, E, d, w)$ be a cyclic graph. Basic retiming[9] (or retiming for short in this paper) r is defined as a function $r : V \rightarrow Z$, which transforms G to a functionally equivalent cyclic graph $G_r = (V, E, d, w_r)$. The set Z represents natural integers.

The weight of each edge $e_{u,v}$ in G_r is defined as follows:

$$w_r(e_{u,v}) = w(e_{u,v}) + r(v) - r(u), \ \forall e_{u,v} \in E. \quad (1)$$

Since in the context of hardware the weight of each edge in G_r represents the number of registers on that edge, then we must have:

$$w_r(e_{u,v}) \geq 0, \ \forall e_{u,v} \in E. \quad (2)$$

Any retiming r that satisfies inequality (2) is called a valid retiming. From expressions (1) and (2) one can deduce the following inequality:

$$r(u) - r(v) \leq w(e_{u,v}), \ \forall e_{u,v} \in E. \quad (3)$$

Let us denote by $P(u, v)$ a path from node *u* to node *v* in *V*. Equation (1) implies that for every two nodes *u* and *v* in *V*, the change in the register count along any path $P(u, v)$ depends only on its two endpoints:

$$w_r(P(u, v)) = w(P(u, v)) + r(v) - r(u),\ \forall u, v \in V, \quad (4)$$

where:

$$w(P(u, v)) = \sum_{e_{x,y} \in P(u, v)} w(e_{x,y}). \quad (5)$$

Let us denote by $d(P(u, v))$ the delay of a path $P(u, v)$ from node *u* to node *v*. $d(P(u, v))$ is the sum of the execution delays of all the computational elements that belong to $P(u, v)$.

A *0-weight path* is a path such that $w(P(u, v)) = 0$. The minimal clock period of a synchronous sequential digital design is the longest *0-weight path*. It is defined by the following equation:

$$\Pi = Max_{\forall u, v \in V}\{d(P(u, v)) \mid (w(P(u, v)) = 0)\}. \quad (6)$$

Two matrices called *W* and *D* are very important to the retiming algorithms. They are defined as follows[9]:

$$W(u, v) = Min\{(w(P(u, v)))\},\ \forall u, v \in V, \quad (7)$$

and

$$D(u, v) = Max\{d(P(u, v)) \mid w(P(u, v)) = W(u, v)\},\ \forall u, v \in V \quad (8)$$

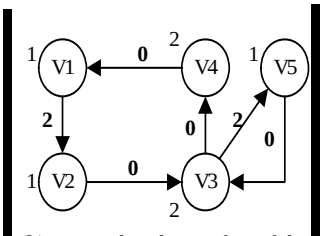
The matrices *W* and *D* can be computed as explained in[9].

Minimizing the clock period of a synchronous sequential digital design is one of the original applications of retiming that are reported in[9]. For instance, for Fig. 1 (b), the clock period of that design is (see its definition above): $\Pi = 6$, which is equal to the sum of execution delays of computational elements $v_{i = 1, ..., 4}$ (i.e., $\Pi = 6 = d(v_2) + d(v_3) + d(v_4) + d(v_1)$). However, we can obtain $\Pi = 3$ if we apply the following retiming vector $\{2, 1, 1, 2, 0\}$ to the vector of nodes $\{1, 2, 3, 4, 5\}$ in G. The retimed graph G_r is presented by Fig. 1 (c). Notice that the weight of each arc in G_r is computed using expression (1).

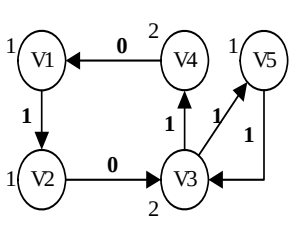
Once a retiming defined on Z is computed for solving a target problem, it can then be transformed to a non-negative retiming without impact on the solution of the problem. In the rest of this paper, we consider non-negative retiming only, unless it is explicitly specified otherwise. For the purpose of this paper, we extract from[9] the following theorem, which is also proved in[9].

```
#define U   1000
main () {
   int  a[U], b[U], c[U], d[U], e[U], i;
   for (i=2; i<= U; i++)
      v1: a[i] = 1 + d[i];
      v2: b[i] = 1 + a[i-2];
      v3: c[i] = b[i] * e[i];
      v4: d[i] = 2 * c[i];
      v5: e[i] = 1 + c[i-2];
}
```

(a) Simple loop.



(b) Directed cyclic graph model.



(c) Retimed graph with $\Pi = 3$.

*Figure 1.*A simple loop, its directed cyclic graph model, and a retimed version of graph (b).

Theorem 1 : *Let* $G = (V, E, d, w)$ *be a synchronous digital design, and let* Π *be a positive real number. Then there is a retiming r of G such that the clock period of the resulting retimed design* G_r *is less than or equal to* Π *if and only if there exists an assignment of integer value r(v) to each node v in V such that the following conditions are satisfied: (1)* $r(u) - r(v) \leq w(e_{u,v}), \forall e_{u,v} \in E$, *and (2)* $r(u) - r(v) \leq W(u, v) - 1, \forall u, v \in V$ *such that* $D(u, v) > \Pi$. □

4 VALID PERIODIC SCHEDULE

Let $G = (V, E, d, w)$ be a directed cyclic graph modeling a loop. A schedule is a function $s : N \times V \rightarrow N$ that, for each iteration $k \in N$ of the loop, determines the start execution time $s_k(v)$ for each operation v of the loop's body. Here, N is the set of non-negative integers.

The schedule s is said to be *periodic* with period Π iff it satisfies (9):

$$s_k(v) = s_0(v) + \Pi \cdot k, \forall k \in N, \forall v \in V, \quad (9)$$

where $s_0(v)$ is the start execution time of the first instance of the operation v. Without loss of generality, we assume through this paper that:

$$s_0(v) \geq 1, \forall v \in V. \quad (10)$$

In this paper, the schedule s is said to be *valid* iff it satisfies both *data dependency constraints* and *resource constraints*.

Data dependency constraints mean that a result computed by operation u can be used by operation v only after u has finished computing that result. In terms of start execution time, this is equivalent to the following inequality:

$$s_{(k + w(e_{u,v}))}(v) \geq s_k(u) + d(u), \forall k \in N, \forall e_{u,v} \in E. \quad (11)$$

Using equation (9), inequality (11) transforms to:

$$s_0(v) - s_0(u) \geq d(u) - \Pi \cdot w(e_{u,v}), \forall e_{u,v} \in E. \quad (12)$$

In this paper, resource constraints mean that at any time, the number of operations that require execution on the processing unit number k must not exceed 1. For resource constraints, we handle in this paper each processing unit individually since this allows us to also realize the binding task (i.e., once the schedule is computed, we will automatically know on which resource each operation will execute).

The following notations and definitions will be used in this paper.

- $|F|$ The number of all the available functional units.
- $F(v)$ The set of labels of processing units that can execute $v \in V$. It is: $F(v) = \{k \in \{1, ..., |F|\} \mid v$ *can execute on the functional unit number* $k\}$.
- $x_{v,t(v),k}$ 0-1 unknown variable associated to each $v \in V$. This variable will be equal to 1 if operation v starts executing at time $t(v)$ on the functional unit number k, otherwise it will be equal to 0.

Notice that, in this paper, we assume that the functional units are not pipelined. Moreover, we are interested in the class of *valid periodic schedules* that satisfy the following constraint while Π must be the smallest possible.

$$1 \le s_0(v) \le (\Pi + 1 - d(v)),\ \forall v \in V. \tag{13}$$

If the operation v is executed using the functional unit number k, then the execution delay of v will be $d_k(v)$ instead of $d(v)$. Using $d_k(v)$ as an execution delay of v, the operation v will execute in the discrete interval of time $\{1,\ ..., (\Pi + 1 - d_k(v))\}$. By using binary variables $x_{v,\,t(v),\,k}$ as well as other notations defined above and the expression (13), we can write each $s_0(v)$ as follows:

$$s_0(v) = (\sum_{k \in F(v)} \sum_{t(v) = 1}^{(\Pi + 1 - d_k(v))} t(v) \cdot x_{v,\,t(v),\,k}),\ \forall v \in V \tag{14}$$

where:

$$\sum_{k \in F(v)} \sum_{t(v) = 1}^{(\Pi + 1 - d_k(v))} x_{v,\,t(v),\,k} = 1,\ \forall v \in V,\ \text{and} \tag{15}$$

$$x_{v,\,t(v),\,k} \in \{0, 1\},\ \forall v \in V,\ k \in F(v),\ t(v) = 1, 2, ..., (\Pi + 1 - d_k(v)) \tag{16}$$

In this paper, we are interested in identical binding for all the iterations of the loop. The operation v and all its future instances will execute on the same functional unit k. Hence, assuming the expressions (15) and (16) are available, the execution delay of v is :

$$d(v) = (\sum_{k \in F(v)} \sum_{t(v) = 1}^{(\Pi + 1 - d_k(v))} d_k(v) \cdot x_{v,\,t(v),\,k}),\ \forall v \in V. \tag{17}$$

Notice that thanks to the binary variables $x_{v,\,t(v),\,k}$, the binding task will be implicitly carried out once the values of these variables become known.

Now, we will show how to derive a formal expression for the resource constraints. Any operation $v \in V$ which is executing at time t implies that v has started to execute somewhere in the discrete interval $\{(Max(\ 1\ , (t - d_k(v) + 1))), \ldots, t\}$, which transforms to (18):

$$\sum_{k \in F(v)} \sum_{t(v) = Max(1,\,(t - d_k(v) + 1))}^{t} x_{v,\,t(v),\,k} = 1,\ \forall v \in V,\ \ t = 1, 2,\ ..., (\Pi + 1) \tag{18}$$

Using expression (13), any operation $v \in V$ must start executing no later than $(\Pi + 1 - d_k(v))$. Thus, equation (18) transforms to:

$$\sum_{k \in F(v)} \sum_{t(v) = Max(1,\,(t - d_k(v) + 1))}^{Min((\Pi + 1 - d_k(v)),\,t)} x_{v,\,t(v),\,k} = 1,\ \forall v \in V,\ \ t = 1, 2,\ ..., (\Pi + 1) \tag{19}$$

The schedule s must be computed in such a way that at any time $t = 1, 2,\ ..., (\Pi + 1)$, the number of operations which are executing on any functional unit number k, $k = 1, 2, ..., |F|$, must not exceed 1 (recall that each functional unit is handled individually in this paper). This transforms to (20):

$$\sum_{\{\forall v \in V \text{ and } k \in F(v)\}} \left(\sum_{t(v) = Max(1,\,(t - d_k(v) + 1))}^{Min((\Pi + 1 - d_k(v)),\,t)} x_{v,\,t(v),\,k} \right) \le 1$$

$$k = 1, 2, ..., |F|, \qquad t = 1, 2, ..., (\Pi + 1) \tag{20}$$

Putting all these developments together, our target *valid periodic schedule* in this paper must satisfy the following set of constraints expressed by (21)-to-(26). Notice that the constraint expressed by (13) is implicitly incorporated into (21)-to-(26) and hence it is implicitly satisfied.

- *Constraint#1: Each operation must start to execute at one and only one point of time.*

$$\sum_{k \in F(v)} \sum_{t(v)=1}^{(\Pi + 1 - d_k(v))} x_{v, t(v), k} = 1, \ \forall v \in V. \quad (21)$$

- *Constraint#2. Data dependencies must be satisfied.*

$$\left(\sum_{k \in F(v)} \sum_{t(v)=1}^{(\Pi + 1 - d_k(v))} t(v) \cdot x_{v, t(v), k} \right) - \left(\sum_{k \in F(u)} \sum_{t(u)=1}^{(\Pi + 1 - d_k(u))} t(u) \cdot x_{u, t(u), k} \right)$$

$$\geq \left(\sum_{k \in F(u)} \sum_{t(u)=1}^{(\Pi + 1 - d_k(u))} d_k(u) \cdot x_{u, t(u), k} \right) - \Pi \cdot w(e_{u, v}), \ \forall e_{u, v} \in E \quad (22)$$

- *Constraint#3. Only one operation can execute on a functional unit at any point of time.*

$$\sum_{\{ \forall v \in V \text{ and } k \in F(v) \}} \left(\sum_{t(v) = Max(1, (t - d_k(v) + 1))}^{Min((\Pi + 1 - d_k(v)), t)} x_{v, t(v), k} \right) \leq 1$$

$$k = 1, 2, ..., |F|, \qquad t = 1, 2, ..., (\Pi + 1) \quad (23)$$

- *Constraint#4. It is just a re-writing of the expression (16). So we have:*

$$x_{v, t(v), k} \leq 1, \ \forall v \in V, \ k \in F(v), \ t(v) = 1, 2, ..., (\Pi + 1 - d_k(v)) \quad (24)$$

$$x_{v, t(v), k} \geq 0, \ \forall v \in V, \ k \in F(v), \ t(v) = 1, 2, ..., (\Pi + 1 - d_k(v)) \quad (25)$$

$$x_{v, t(v), k} \in N, \ \forall v \in V, \ k \in F(v), \ t(v) = 1, 2, ..., (\Pi + 1 - d_k(v)) \quad (26)$$

MinPeriod Algorithm: To compute a valid periodic schedule that satisfies the constraints expressed by (21)-to-(26) (which implicitly include (13) as well) while minimizing Π, then one can initialize Π to a lower bound (0 by default) and iteratively do $\Pi = \Pi + 1$ (or do a binary search) until the system expressed by (21)-to-(26) can be solved. Π must be fixed before solving this system to allow for the linearity of (21)-to-(26). The tool [10] can be used to solve this system at each iteration, as we did for the algorithm in Section 7.

5 CODE GENERATION AFTER RETIMING

In the rest of this paper, to avoid repetition, we denote by Π the length of the longest *0-weight path* in a given loop (including the retimed loop). Assuming no resource constraints at this time, Π can be minimized by using one of the retiming algorithms for clock period minimization proposed in[9].

Our objective in this section is to show how to generate the transformed code after applying a retiming on a given loop to minimize Π. The transformed code is constituted by three parts: *prologue*, *new loop*, and *epilogue*.

Recall that the *prologue* is the segment of the original code that must be executed before a repetitive processing appears that corresponds to the *new loop*. The *epilogue* is the segment of the original code that cannot be executed by the *new loop* and the *prologue*. Let us use the example of a loop in Fig. 1(a). As provided in Section 3, the value of Π for this loop is 6 units of time. By applying the retiming vector $\{2, 1, 1, 2, 0\}$ on Fig. 1 (b), we can get the retimed graph given by Fig. 1 (c), where the value of Π is now 3 units of time. The transformed code after applying this retiming is given in Fig. 2(a).

Let $M = Max\{r(v), \forall v \in V\}$. With the help of Fig. 2, one can easily double-check that the **P**rologue, the new **L**oop and the **E**pilogue for the retimed loop can be obtained by the algorithm **PLE** below.

Algorithm: PLE

/* The index i of the original loop is in $[L, U]$. $M = Max\{r(v), \forall v \in V\}$ */

1- Prologue Generation

1.1 $i = L$.

1.2 While $(i < (L + M))$ do

$\forall v \in V$, if $((i + r(v)) < (L + M))$ then generate a copy of v at iteration i as it was in the original loop's body. And, do: $i = i + 1$.

2- New Loop Generation

2.1 The index of the new loop is in $[(L + M), U]$.

2.2 Determine the body of the new loop as follows. $\forall v \in V$, generate a copy of v where the index of each array in v of the original loop's body has now to be decreased by $r(v)$.

3- Epilogue Generation

3.1 $i = U + 1$.

3.2 While $(i \leq (U + M))$

$\forall v \in V$, if $((i - r(v)) \leq U)$ then generate a copy of v by evaluating its expression derived from Step 2.2. And, do: $i = i + 1$.

End of the algorithm PLE.

6 COMPUTING A RETIMING WITH REDUCED CODE SIZE UNDER TIMING CONSTRAINTS FOR THE CASE OF UNLIMITED RESOURCES

Recall that $\forall v \in V$, $r(v)$ is the value assigned by the retiming r to the vertex v. Also, notice that $M = Max\{r(v), \forall v \in V\}$. With the help of the *PLE* algorithm above and Fig. 2(a) and 2(b) below, one can deduce that for the code after retiming a loop, we have:

- The *prologue* contains $(M - r(v))$ copies of the operation modeled by $v \in V$.
- The *epilogue* contains $r(v)$ copies of the operation modeled by $v \in V$.
- For the code composed by the *prologue* and the *epilogue*, the number of instances of each operation is exactly M times.
- If the original loop execute K times, the new loop execute $(K - M)$ times.

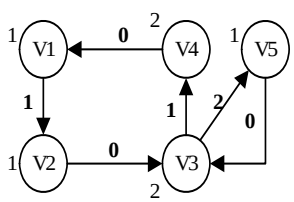
Consequently, to reduce the code size for the retimed loop, one needs to reduce the value of M. The value of M depends on the retiming used and is not

```
#define U 1000
main () {
int  a[U], b[U], c[U],  d[U], e[U], i;
    /* Prologue */
    v2: b[2] = 1 + a[0];
    v3: c[2] = b[2] * e[2];
    v5: e[2] = 1 + c[0];
    v5: e[3] = 1 + c[1];
    /* New Loop */
    for (i=4; i<= U; i++) {
      v1: a[i-2] = 1 + d[i-2];
      v2: b[i-1] = 1 + a[i-3];
      v3: c[i-1] = b[i-1] * e[i-1];
      v4: d[i-2] = 2 * c[i-2];
      v5: e[i] = 1 + c[i-2];
    }
    /* Epilogue */
    v1: a[U-1] = 1 + d[U-1];
    v4: d[U-1] = 2 * c[U-1];
    v1: a[U] = 1 + d[U];
    v2: b[U] = 1 + a[U-2];
    v3: c[U] = b[U] * e[U];
    v4: d[U] = 2 * c[U];
}
```

(a) Code after applying the retiming vector {2, 1, 1, 2, 0} on Fig. 1b **without reducing** the code size.

```
#define U 1000
main () {
int a[U], b[U], c[U], d[U], e[U], i;
    /* Prologue */
    v2: b[2] = 1 + a[0];
    v3: c[2] = b[2] * e[2];
    v5: e[2] = 1 + c[0];
    /* New Loop */
    for (i=3; i<= U; i++) {
      v1: a[i-1] = 1 + d[i-1];
      v2: b[i] = 1 + a[i-2];
      v3: c[i] = b[i] * e[i];
      v4: d[i-1] = 2 * c[i-1];
      v5: e[i] = 1 + c[i-2];
    }
    /* Epilogue */
    v1: a[U] = 1 + d[U];
    v4: d[U] = 2 * c[U];
}
```

(b) Code after applying the retiming vector {1, 0, 0, 1, 0} on Fig. 1b **while reducing** the code size.



(c) Directed cyclic graph model for (b) derived from Fig. 1(b) after applying the retiming vector {1, 0, 0, 1, 0}.

*Figure 2.*Transformed code after retiming a loop.

unique. Indeed, we showed in the previous sections that the value of Π for the loop in Fig. 1 can be reduced from 6 units of time to 3 units of time by applying the retiming vector $\{2, 1, 1, 2, 0\}$ on Fig. 1 (b). But, the retiming vector $\{2 + x, 1 + x, 1 + x, 2 + x, 0 + x\}$ can also be applied to obtain $\Pi = 3$ units of time, where x is a non-negative integer. For the first retiming vector, we have $M = 2$, and for the second one we have $M = 2 + x$. The last retiming vector is then not a good choice and can lead to completely unroll the loop if $x = U - 2$. While the first retiming vector (i.e., $\{2, 1, 1, 2, 0\}$) is better than the second one (i.e., $\{2 + x, 1 + x, 1 + x, 2 + x, 0 + x\}$) for this example, it is not a good one as well. Indeed, one can achieve $\Pi = 3$ units of time by applying the retiming vector $\{1, 0, 0, 1, 0\}$ on Fig. 1 (b). In this case, the resulting retimed graph is given by Fig. 2(c), and the transformed code is provided on Fig. 2(b). For this vector, we have $M = 1$. Also, by comparing Fig. 2(a) and Fig. 2(b), one can easily notice that the size of the code has been reduced (which is consistent with the statements in the previous paragraph).

Recall that we are assuming non-negative retiming as explained in the end of Section 3. To determine a retiming that leads to a small value of *M*, one can add a variable upper bound β on the retiming value for each $v \in V$ as in (27).

$$r(v) \leq \beta, \ \forall v \in V. \tag{27}$$

Using this upper bound β, we propose the following method to compute a retiming that leads to a small value of *M* (and thus to a code with a small size).

We can extend the system of inequalities in Theorem 1 by first adding an upper bound on the value of the retiming function as we did in inequality (27),

but assuming without loss of generality that retiming will take non-negative values. And, second minimizing the value of the unknown variable β. This leads to the Integer Linear Program (ILP) composed by (28)-to-(32) below.

In the ILP (28)-to-(32), recall that Π denotes the length of the longest *0-weight path* in a given loop (including the retimed loop). Also, notice that in the context of clocked sequential circuits, Π is the clock period. Hence, to solve this ILP, we must first compute the value of Π if it is not provided by the user. To do so, a retiming using one of the algorithms in[9] (for clock period minimization) can be first carried out.

$$Minimize(\beta) \tag{28}$$

Subject to:

$$r(u) - r(v) \leq w(e_{u,v}),\ \forall e_{u,v} \in E \tag{29}$$

$$r(u) - r(v) \leq W(u, v) - 1,\ \forall u, v \in V\ such\ that\ D(u, v) > \Pi \tag{30}$$

$$r(v) \leq \beta,\ \forall v \in V \tag{31}$$

$$\beta, r(v) \in N,\ \forall v \in V \tag{32}$$

The ILP (28)-(32) can be solved using the linear programming solver in[10].

Lemma 1 : *The ILP (28)-(32) can be optimally solved in polynomial run-time*.

Proof: Omitted due to the space limitation. □

7 COMPUTING A RETIMING WITH REDUCED CODE SIZE UNDER TIMING AND RESOURCES CONSTRAINTS

In this case, one needs to also perform a scheduling under timing and resource constraints. Since we are dealing with loops, this schedule is periodic. Hence, one needs to compute a valid periodic schedule while minimizing timings as we have presented in Section 4. For instance, to compute this schedule while minimizing its period, one can use the *MinPeriod* algorithm outlined at the end of Section 4. However, notice that this algorithm will produce a *minimal value* for the period Π *relative to the input graph* G. This algorithm might miss the *absolute minimal value* of Π since it does not alter the weight of arcs in the graph (in fact, any algorithm that does not alter the weight of arcs in the graph might miss the *absolute minimal value* of Π). Altering the weight of arcs in the graph is the task of retiming. For instance, assume we have three resources: two identical adders and one multiplier. Suppose that the execution delays of the adder and multiplier are 1 and 2 units of times, respectively. So for these resources, applying the *MinPeriod* algorithm on the graph in Fig. 1(b) will lead to a minimal value of $\Pi = 6$ units of time. However, we can get $\Pi = 3$ units of time if we apply the *MinPeriod* algorithm on either the retimed graph in Fig. 1(c) or the retimed graph in Fig. 2(c) (b). Recall that the retiming used for producing the retimed graph in Fig. 2(c)

is better than the retiming used to produce the retimed graph in Fig. 1(c), since it allows to reduce the code size as it was explained in Section 6.

To get an *absolute minimal value* of Π, one needs to *optimally unify scheduling and retiming*. To achieve this optimal unification, we propose to extend the *MinPeriod* algorithm. The extended version (see the *MinPeriod-WithReducedCodeSize* below) of this algorithm should operate on a *dynamically retimed graph* $G_r = (V, E, d, w_r)$. The graph G_r is as the one defined in Section 3 except that in equation (1), the values $r(v)$ and $r(u)$ will be calculated on the fly during the schedule determination; this is why we said: *dynamically retimed graph*. Of course the dynamically retimed graph G_r must be *functionally equivalent* to the original graph G, which means that the inequality (3) must hold. Moreover, the inequality (27) must also be respected since it will allow to produce a retiming with a reduced code size. Since we are interested in non-negative retiming, then the following expression (33) must be considered as well.

$$r(v) \in N, \forall v \in V \tag{33}$$

And, for the *MinPeriodWithReducedCodeSize* algorithm, data dependencies expressed by (22) now transform to the expression (34) below, since the weight of each arc $e_{u, v} \in E$ is no longer $w(e_{u, v})$ but it is now equal to $(w(e_{u, v}) + r(v) - r(u))$. Moreover, we have two parameters to minimize: Π and β. So, we need to first minimize Π for a large value of β in order to get an absolute minimal value of Π. Next, we need to minimize β to achieve this latter value of Π.

$$\left(\sum_{k \in F(v)} \sum_{t(v) = 1}^{(\Pi + 1 - d_k(v))} t(v) \cdot x_{v, t(v), k} \right) - \left(\sum_{k \in F(u)} \sum_{t(u) = 1}^{(\Pi + 1 - d_k(u))} t(u) \cdot x_{u, t(u), k} \right)$$
$$\geq \left(\sum_{k \in F(u)} \sum_{t(u) = 1}^{(\Pi + 1 - d_k(u))} d_k(u) \cdot x_{u, t(u), k} \right) - \Pi \cdot \left(w(e_{u, v}) + r(v) - r(u) \right), \forall e_{u, v} \in E \tag{34}$$

Algorithm: *MinPeriodWithReducedCodeSize*

Inputs: Cyclic graph $G = (V, E, d, w)$, the set of available functional units, and the set of processing units that can execute each operation $v \in V$.

Outputs: Schedule, binding, and minimal values of Π and β which is the maximal value of retiming.

Begin

1. The optimal value of Π is in the interval [*L*, *U*]. *L* is the delay of the fastest functional unit. *U* is *the number of nodes* in the graph *G times the execution delay* of the slowest functional unit. If a lower bound on the minimal value of Π is not provided by the user, then let: $\Pi = L$.
2. Using a linear programming solver like the one in[10], solve the system expressed by: (3), (21), (23)-to-(26) and (33)-(34).
3. If the system expressed by (3), (21), (23)-to-(26) and (33)-(34) has a solution, then go to Step 4.
   Else $\Pi = \Pi + 1$ (or do a binary search in [*L*, *U*] to determine Π). And go to Step 2.
4. Let *L* = 0, and *U* = *K*. Let Π be fixed to the value found in Step 3.
5. Let $\beta = \lceil (L + U)/2 \rceil$. If β cannot be reduced further then go to Step 8.

6. Using a linear programming solver like the one in[10], solve the system expressed by: (3), (21), (27), (23)-to-(26) and (33)-(34).
7. If the system expressed by (3), (21), (27), (23)-to-(26) and (33)-(34) has a feasible solution, then
   7.1. From the solution found in Step 6, extract the values for the schedule and the binding as well as the values of Π and β, and save them.
   7.2. Let $U = \beta$. And go to Step 5.
   else Let $L = \beta$. And go to Step 5.
8. Report the result extracted in Step 7.1 and Stop.

End of the algorithm *MinPeriodWithReducedCodeSize*.

8 EXPERIMENTAL RESULTS

Although the proposed approaches in this paper solve optimally the target problem, we however found it logic to experimentally test theirs effectiveness in terms of: *i*) the impact of retiming on reducing the length of the critical paths (*i.e., the length of the longest 0-weight path denoted as* Π *in this paper*) in the case of unlimited resources, *ii*) the impact of retiming on reducing the period's length of the schedule for the case of limited resources, and *iii*) the impact of controlling the values of retiming on reducing the code size of the retimed loops. To this end, some real-life digital filters (from the domain of digital signal processing) are used as input loops to be optimized. The body of these loops (filters) are composed by additions and multiplications operations only. The names of these filters and other designs are labeled on Fig. 3 and 4 as follows. L1: The exemple given on Fig. 1b. L2 : Correlator. L3: FIR Filter. L4: Biquadratic Filter. L5: Polynomial Divider. L6: Three Tape Non-Recursive Digital Filter. L7: Lowpass Reverberator First Order with Feedback. L8: Allpass Reverberator Canonical Form. L9: Coupled form Sine-Cosine Generator. L10: Digital Cosinusoidal Generator. L11: Resonator Filter.

For the results on Fig. 3 and 4, we developed a C++ tool which, from an input graph modeling the target loop, automatically generates the expressions of the ILP (28)-to-(32). In this tool, we also implemented the proposed algorithm *MinPeriodWithReducedCodeSize*. This tool also automatically generates the expressions of the system to be solved in the steps 3 and 7 of the *MinPeriodWithReducedCodeSize* algorithm, and needs to be re-run until the *MinPeriodWithReducedCodeSize* finds the optimal values to be computed.

For results in Fig. 3, we assume unlimited resources. We first compute the value of Π without any retiming. Next, we apply a retiming for minimizing Π, by implementing and running an algorithm for clock period minimization from[9]. This latter step is done twice: *i*) we apply a retiming for minimizing Π without controlling the largest value of the retiming, and *ii*) we apply a retiming to achieve the same minimal value of Π but we minimize the largest value of the retiming (so, in this case we solve the ILP (28)-to-(32)). As it can be noticed from Fig. 3, it was possible to reduce the value of Π for some

input designs. For the set of input designs in this experimentation, the maximal relative reduction of Π is 57%. When it was possible to reduce Π by applying a retiming, then by controlling the values of retiming, we were able to reduce the code size of the retimed loops by as high as 66.67%.

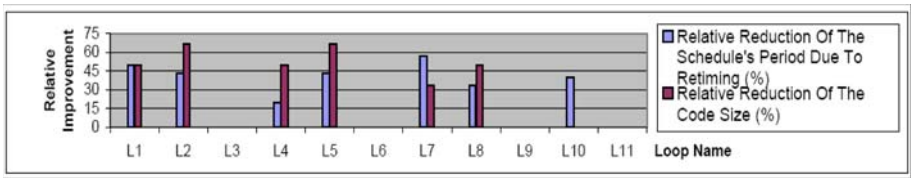


*Figure 3.*Case of unlimited resources.

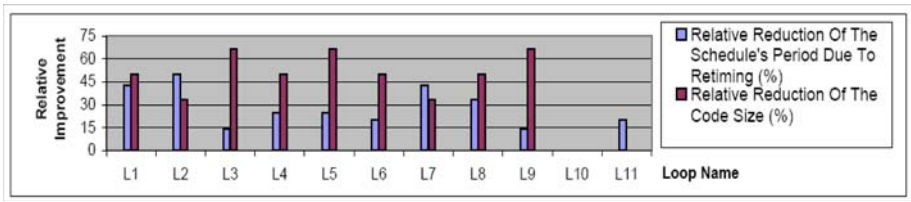


*Figure 4.*Case of limited resources.

For results in Fig. 4, we assume limited resources. In this case, we suppose that we have an hypothetical processor with 2 non-identical adders and 2 non-identical multipliers. For our adders, one of them has execution delay equal to 1 unit of time and the second one has execution delay equal to 2 units of time. For our multipliers, one of them has execution delay equal to 2 units of time and the second one has execution delay equal to 3 units of time. We think it makes sense to take a fast functional unit and a slow functional unit for each type of the resources, since some operations of the loop are not critical and should be executed using slow units in order to reduce energy/power dissipation; normally, a fast functional unit consumes more power than its slow counterpart. Next, for this hypothetical processor, we first compute a valid periodic schedule with a minimal period Π but without any retiming. For doing this, the C++ tool executes the algorithm *MinPeriod*. Then, we compute a valid periodic schedule with a minimal period Π but in this case we perform retiming as well. For this latter case, the C++ tool executes the algorithm *MinPeriodWithReducedCodeSize* and we proceed in two steps: *i*) we apply a retiming without controlling its largest value (in this case, β has no effect in the algorithm *MinPeriodWithReducedCodeSize*), and *ii*) we apply a retiming while minimizing its largest value (by minimizing β in this case). As it can be noticed from Fig. 4, except for the Digital Cosinusoidal Generator design, it was possible to reduce the value of Π for all the other input designs. For the set of input designs in this experimentation, the relative reduction of Π ranges from 14% to 50%. When it was possible to reduce Π by applying a retiming, then by controlling the values of retiming, we were

able to reduce the code size of the retimed loops by as high as 66.67%. The run-time of the algorithms *MinPeriod* and *MinPeriodWithReducedCodeSize* is dominated by the run-time of solving the systems of constraints they contain. The run-time for solving these systems of constraints was less than 4 seconds for these experimental results. The bounds we have introduced in the expressions of these constraints (i.e., see the *Min* and *Max* in some expressions) are helped in achieving such small run-times.

9 CONCLUSIONS

Retiming has been originally proposed as an optimization technique for clocked digital circuits. It has been successfully applied for optimizing loop-intensive programs. Decreasing the code size for retimed loops is of great importance in particular for memory-constrained system-on-chip and embedded systems. The size of the code has also an implicit impact on both the run-time and the power consumption. This paper addressed the problem of reducing the code size for retimed software loops under timing and resource constraints. We mathematically formulated this problem and devised exact algorithms to optimally solve it. Experimental results has re-confirmed the importance of solving this problem. For unlimited resources, the exact polynomial-time method can be used for source-to-source code optimization. For limited resources, the approach can be used to design optimized libraries, to develop heuristics and to test their effectiveness.

References

1. Zhuge Q., Xiao B., and Sha, E.H.M.: Code size reduction technique and implementation for software-pipelined DSP applications. ACM Trans. on Embedded Comput. Syst., V.2, N.4, 2003, pp.590-613.
2. Benini L., Macii A., Macii E., and Poncino M.: Minimizing memory access energy in embedded systems by selective instruction compression. IEEE Transactions on Very Large Scale Integration Systems, V.10, N.5, Oct. 2002, pp:521-531.
3. Granston E., Scales R., Stotzer E., Ward A., and Zbiciak J.: Controlling code size of software-pipelined loops on the TMS320C6000 VLIW DSP architecture. Proceedings 3rd IEEE/ACM Workshop on Media and Streaming Processors, 2001, pp. 29-38.
4. Sung W., and Ha S.: Memory efficient software synthesis with mixed coding style from dataflow graphs. IEEE Trans. on Very Large Scale Integration Systems, V. 8, N. 5, Oct. 2000, pp: 522-526.
5. Lekatsas H, Henkel J., and Wolf W.: Code compression for low power embedded systems design. Proc. of Design Automation Conference, 2000, pp. 294–299.
6. Nemhauser G.L., and Wolsey L.A.: Integer and Combinatorial Optimization, John Wiley and Sons Inc., ISBN 0-471-35943-2, 1999.
7. Fraboulet A., Mignotte A., and Huard G.: Loop alignment for memory accesses optimization. Proc. of 12th International Symp. on System Synthesis, Nov.1999, pp.71-77.
8. Chao L.F., LaPaugh A.S., and Sha E.H.M.: Rotation scheduling, A loop pipelining algorithm. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, V.16, N.3, 1997, pp. 229-239.
9. Leiserson C.E., and Saxe J.B.: Retiming Synchronous Circuitry. Algorithmica, Jan. 1991, pp. 5-35.
10. The LP_Solve Tool: ftp://ftp.ics.ele.tue.nl/pub/lp_solve/