An Approach for Integrating Basic Retiming and Software Pipelining

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-n@rmc.ca

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

ABSTRACT

Basic retiming is an algorithm originally developed for hardware optimization. Software pipelining is a technique proposed to increase instruction-level parallelism for parallel processors. In this paper, we show that applying software pipelining alone for minimizing timings under resource constraints can lead to sub-optimal results, compared to the case if an unification of basic retiming and software pipelining is used. We propose an approach to realize this unification. The approach allows to minimize the code size of the optimized loop as well as minimizing the idleness of computational elements. We extend this approach to solve the problem of minimizing peak power consumption for time-constrained and resource-constrained software pipelined loops. Solving these problems is important for portable embedded systems as well as system-on-chip design. The approaches are tested using known benchmarks. On average, relative timing improvement is 60.19%, and relative reduction of peak power consumption is 13.17% without any trade-off in timings.

Categories and Subject Descriptors

D. Software; D.0 General.

General Terms

Algorithms; Design; Performance.

Keywords

Retiming; Software Pipelining; Instruction-Level Parallelism; VLIW; Superscalar Processor; Peak Power; Timings; Code Size; Embedded Systems; System-on-Chip.

1. INTRODUCTION

The processing speed of digital systems continues to increase thanks to the combination of the improvement of the compilers' intelligence, the development of new architectures, and the advance of the semi-conductor technology that continues to allow putting more and more transistors on the same chip. However, even if new digital systems with improved processing speed continue to emerge, new computational-hungry applications continue to emerge as well while other old applications continue to intimidate even the fastest computer at this time. Applications from digital signal processing and image processing and multimedia applications are examples of applications that still require high processing speed. Applications that require high processing speed are in general those that are loop-intensive.

To increase the processing speed for loop intensive applications, many compiler techniques are used to generate code that efficiently exploits the component of the hardware. Software pipelining [4][8] is one of such techniques. It overlaps the execution of consecutive iterations of a given loop, thereby increasing the instruction-level parallelism which is useful for parallel processors like the Very Large Instruction Word (VLIW) and the superscalar processors.

Software pipelining is not a recent technique. It has been devised since many years, and is used in developing the code for many very-well-placed processors in the market. However, how to realize this technique under some constraints is a challenging problem. Indeed, in software pipelining there are two important parameters: Latency (L) which is the time required to execute all the instructions that constitute the body of the loop, and the Initiation Interval (II) which is the interval of time that separates the start execution time of each two consecutive iterations of the loop. As a first challenging problem is the problem of minimizing II under resources constraints. That problem is an NP-hard problem in general and many heuristic approaches are proposed to approximately solve it. As a second challenging problem is the problem of minimizing L for a given II and under resource constraints. A third challenging problem will be presented shortly.

Increasing the processing speed of digital systems is no-longer the only main design objective to achieve in today and at the future. We have been and would continue to be constrained by the need of reducing the power consumption of digital systems. The need of reducing power consumption is mainly motivated by: (1) the need of reducing the cooling cost for high speed digital systems, and (2) the need of prolonging battery lifetime for battery-powered portable systems. The power consumed in high speed systems transforms to heat which requires special cooling devices in order to avoid malfunction and damaging hardware. Designers must then develop sophisticated cooling mechanisms under cooling-cost constraints. For battery-powered portable systems, prolonging battery life is required for some critical portable systems such as wearable medical systems. In addition, battery life became a product differentiator in the market of portable digital systems.

The peak power is the power consumed at the most power-hungry control-step. Peak power must be reduced since high peak power might lead to malfunction of a digital system or to damaging its hardware. Software pipelining allows to increase the instruction-level parallelism. This means that the number of none-idle computational elements (i.e., ALU, Multiplier, etc.) would increase which would increase the peak power. However, by scheduling those instructions in some manner, peak power can be reduced compared to the case of using a peak-power-unaware schedule.

EMSOFT'04, September 27–29, 2004, Pisa, Italy.

As a third challenging problem related to how to realize the software pipelining technique under some constraints, we state the following. How can one assign instructions (that constitute the body of the loop to be optimized by software pipelining) to control steps under resource constraints and for a given *L* and *II* while minimizing the peak power?

In this paper, we establish a formal relationship between *L* and the code size of loops that are optimized by software pipelining. When we fix *II*, the code size increases when *L* increases. Consequently, to reduce the code size, one needs to reduce *L*. Furthermore, by decreasing *L*, the idleness of computational elements will decrease. We show that applying software pipelining alone to optimize a loop, under resource constraints and a target *II*, will lead to a relative minimal value for *L* (possibly large *L*) compared to the case if an unification of basic retiming [1] and software pipelining is used. We propose an Integer Linear Program (ILP) to realize that unification. This ILP constitutes a flexible mathematical framework. Indeed, we have easily extended that ILP to solve the problem of minimizing peak power as stated above. To the best of our knowledge, this is the first paper in the literature that addresses the latter problem. Although it is not done yet, the proposed mathematical framework can be extended to solve the problem of minimizing register requirement for software pipelined loops.

2. LOOP REPRESENTATION

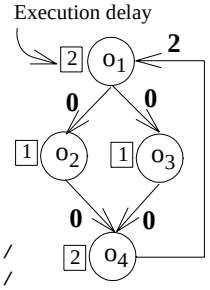
In this paper, we are interested in "for"-type loops as the one in Figure 1 (a). We assume that the body of the loop is constituted by a set of computational and/or assignment instructions only (i.e., no conditional or branch instruction like for instance *if-then-else* is inside the body).

Let N be the set of non-negative integers. We model a loop by a directed cyclic graph $G = (O, E, d, w)$, where O is the set of instructions (or atomic operations like addition and multiplication) in the loop body, and E is the set of arcs that represent data dependencies. Each vertex $o_i \in O$ has a non-negative integer execution delay $d(o_i) \in N$. Each arc $e_{o_i, o_j} \in E$, from vertex $o_i \in O$ to vertex $o_j \in O$, has a weight $w(e_{o_i, o_j}) \in N$, which means that the result produced by o_i at any loop's iteration k is consumed by o_j at iteration $(k + w(e_{o_i, o_j}))$.

Figure 1 presents a simple loop and its directed cyclic graph model. For Figure 1 (b), the execution delay of each operation o_i is specified as a label (i.e., the number within a rectangular) on the left of each vertex of the graph, and the weight $w(e_{o_i, o_j})$ of each arc $e_{o_i, o_j} \in E$ is in bold. For instance, the execution delay of o_1 is 2 time units and the value 0 on the arc e_{o_1, o_2} means that operation o_2 at any loop's iteration k uses the result produced by operation o_1 at loop's iteration $(k - 0)$.

```
#define   U   100
main () {
int A[U], B[U], C[U], D[U];

for (int i=2; i<U; i++) {
  A[i] = 10   * D[i-2]; /*o1*/
  B[i] = 10   + A[i];   /*o2*/
  C[i] = 10   + A[i];   /*o3*/
  D[i] = B[i] * C[i];   /*o4*/
}/*Execution delays of o1 and o4 are 2*/
}/*Execution delays of o2 and o3 are 1*/
```



(a) A simple loop in the programming language C. **(b)** Cyclic graph of (a).

Figure 1. A simple loop and its directed cyclic graph model.

3. INTRODUCTION TO BASIC RETIMING

A synchronous sequential design can be modeled as a directed cyclic graph as we did for loops in Section 2. Instructions become computational elements of the design, arcs become wires, and $w(e_{o_i, o_j})$'s become the number of registers on the wire between computational element o_i and computational element o_j.

Let $G = (O, E, d, w)$ be a synchronous sequential digital design. We denote by Z the set of natural integers. Basic retiming (or retiming for short in this paper) r [1] is defined as a function $r : O \rightarrow Z$, which transforms G to a functionally equivalent synchronous sequential digital design $G_r = (O, E, d, w_r)$ by assigning a label $r_{o_i} = r(o_i)$ to each vertex o_i in G. The physical meaning of the assigned labels can be viewed as follows. If r_{o_i} is positive then we have to move r_{o_i} registers from each output wire of o_i and to put them on each input wire of o_i, assuming that we have at least r_{o_i} registers on each output wire of o_i. If r_{o_i} is negative, the previous process is reversed. When r_{o_i} is equal to zero, no register have to be moved across o_i.

The difference between G and its retimed version G_r is the weight of arcs. The weight $w_r(e_{o_i, o_j})$ of each arc e_{o_i, o_j} in G_r is now defined as follows:

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

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

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

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

$$r(o_j) - r(o_i) \geq -w(e_{o_i, o_j}),\ \forall e_{o_i, o_j} \in E. \quad (3)$$

Let $P(o_i, o_j)$ denotes a path from vertex o_i in O to vertex o_j in O. Equation (1) implies that for every two vertices o_i and o_j, the change in the register count along any path $P(o_i, o_j)$ depends only on its two endpoints:

$$w_r(P(o_i, o_j)) = w(P(o_i, o_j)) + r(o_j) - r(o_i),\ \forall o_i, o_j \in O, \quad (4)$$

where:

$$w(P(o_i, o_j)) = \sum_{e_{o_i, o_j} \in P(o_i, o_j)} w(e_{o_i, o_j}). \quad (5)$$

Let $d(P(o_i, o_j))$ denotes the delay of a path $P(o_i, o_j)$ from vertex o_i to vertex o_j. $d(P(o_i, o_j))$ is the sum of the execution delays of all the vertices that belong to $P(o_i, o_j)$.

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

$$\Pi = Max_{\forall o_i, o_j \in O}\{d(P(o_i, o_j)) \mid (w(P(o_i, o_j)) = 0)\}. \quad (6)$$

One application of retiming is to minimize the clock period of synchronous sequential digital designs. For instance, by thinking of Figure 1 (b) as a synchronous sequential design, the clock period of that design is $\Pi = 5$ time units, which is equal to the sum of execution delays of vertices $o_{i = 1, 2, 4}$ (i.e., $\Pi = 5 = d(o_1) + d(o_2) + d(o_4)$). However, we can obtain $\Pi = 3$ time units if we apply the following retiming vector $\{0, 0, 0, 1\}$ to the vector of vertices $\{o_1, o_2, o_3, o_4\}$ in G, where the value located at the i^{th} position in the retiming vector corresponds to the value assigned by r to the vertex located at the i^{th} position in the vector of vertices. The retimed graph G_r is presented by Figure 2.

The new arcs' weights are computed using equation (1).

$w_r(e_{o_1, o_2}) = 0 = 0 + 0 - 0$

$w_r(e_{o_1, o_3}) = 0 = 0 + 0 - 0$

$w_r(e_{o_2, o_4}) = 1 = 0 + 1 - 0$

$w_r(e_{o_3, o_4}) = 1 = 0 + 1 - 0$

$w_r(e_{o_4, o_1}) = 1 = 2 + 0 - 1$

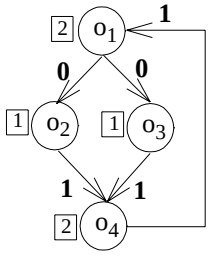


Figure 2. Retiming Figure 1 (b) by assigning 0, 0, 0, and 1 to vertices o_1, o_2, o_3, and o_4 respectively.

4. NOTATIONS AND DEFINITIONS

The following notations and definitions will be used in the rest of the paper. Without loss of generality, we assume in this paper that computational elements are not pipelined.

$t(o_i)$ Denotes the start execution time of $o_i \in O$.

$x_{o_i, t(o_i)}$ 0-1 unknown variable associated to each $o_i \in O$. This variable is equal to 1 if o_i starts executing at time $t(o_i)$, otherwise it is equal to 0.

μ Known variable, which is equal to the number of different classes of computational elements of the same type. For instant, if we have one set of adders and one set of multipliers then $\mu = 2$.

μ_k Denotes the k^{th} class of computational elements of the same type, where $k = 1, 2, ..., \mu$. For instance, we might denote by μ_1 the class of adders, and by μ_2 the class of multipliers.

$|\mu_k|$ Denotes the number of resources in the k^{th} class of computational elements. Suppose that we have 3 adders and 2 multipliers. Using the above notation, we then have $|\mu_1| = 3$ and $|\mu_2| = 2$.

β Function that binds each $o_i \in O$ to one of the classes of computational elements μ_k, $k = 1, 2, ..., \mu$. If the execution of o_i requires a resource from the class μ_m, then we have $\beta(o_i) = \mu_m$.

l Known variable which is an upper bound on the latency L to be minimized. A trivial value for l is the sum of the delays of all the instructions in the loop's body, which is equal to the run-time when the loop is executed sequentially. For instance for the loop in Figure 1, we have $l = 6$.

5. VALID PERIODIC SCHEDULE

Let $G = (O, E, d, w)$ be a directed cyclic graph modeling a loop. A schedule is a function $s : N \times O \rightarrow N$ that, for each iteration $k \in N$ of the loop, determines the start execution time $s_k(o_i)$ for each instruction o_i of the loop's body.

The schedule s is said to be periodic with period P iff it satisfies the following equation:

$$s_k(o_i) = s_0(o_i) + P \cdot k, \ \forall k \in N, \ \forall o_i \in O, \qquad (7)$$

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

$$s_0(o_i) \geq 1, \ \forall o_i \in O. \qquad (8)$$

In this paper, the schedule s is said to be valid iff it satisfies both data dependency constraints and resource constraints (in case of limited number of resources).

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

$$s_{(k + w(e_{o_i, o_j}))}(o_j) \geq s_k(o_i) + d(o_i), \ \forall k \in N, \ \forall e_{o_i, o_j} \in E. \qquad (9)$$

Using equation (7), inequality (9) transforms to:

$$s_0(o_j) - s_0(o_i) \geq d(o_i) - P \cdot w(e_{o_i, o_j}), \ \forall e_{o_i, o_j} \in E. \qquad (10)$$

Resource constraints mean that at any time, the number of instructions that require execution on the class of computational elements, say μ_m, must not exceed the number, $|\mu_m|$, of available resources in μ_m.

When there are no resources constraints (unlimited number of resources), then the schedule s is valid and periodic with period P, iff the system of inequalities described by (10) has a solution for the unknown $s_0(o_i)$. By making a sum of all the inequalities of this system for any cycle, the left hand side will lead to 0. After doing this sum, then by first passing the term that contains P to the other side of the resulting inequality, and secondly doing this for any cycle in the graph, one can prove that the system of inequalities described by (10) has a solution iff P is such that:

$$P \geq Max_{c \in \zeta}\left(\left(\sum_{\forall o_i \in O \text{ and } e_{o_i, o_j} \in c} d(o_i)\right) / \left(\sum_{e_{o_i, o_j} \in c} w(e_{o_i, o_j})\right)\right) \qquad (11)$$

where ζ denotes the set of directed cycles in the directed cyclic graph modeling the loop.

The right hand side of inequality (11) is a Minimum Cost-to-Time Ratio Cycle Problem [5], and can be optimally solved in polynomial run-time using, for instance, one of the algorithms described in [5].

Inequality (11) allows to compute a lower bound on P that is due to data dependency constraints only. Another lower bound on P that is due to resource constraints only can be derived as follows. For instance, if we have only 3 instructions of type addition, and only 2 identical adders with execution delay equal to 1ns (the same as the execution delay of any of those instructions), then the time required to execute those 3 instructions cannot be less than 1.5 ns, where $1.5 = (1 + 1 + 1)/2$. Suppose that there are $|\mu_m|$ resources in the class of computational elements μ_m. The time required to execute all the instructions of the same iteration that execute on resources of class μ_m is at least

$$\left(\sum_{(\forall o_i \in O \text{ and } \beta(o_i) = \mu_m)} d(o_i)\right) / |\mu_m|. \qquad (12)$$

Hence, we have to wait at least the time expressed by (12) before starting to execute the next instance of any one of those instructions. The schedule is periodic. Thus, we have that:

$$P \geq \left(\sum_{(\forall o_i \in O \text{ and } \beta(o_i) = \mu_m)} d(o_i)\right) / |\mu_m|. \qquad (13)$$

Since inequality (13) must be met for all the μ available classes of resources, we then have:

$$P \geq Max_{m = 1, 2, ..., \mu} ((\sum_{(\forall o_i \in O \text{ and } \beta(o_i) = \mu_m)} d(o_i)) / |\mu_m|) . \quad (14)$$

Using (11) and (14), a lower bound on the period P of any valid periodic schedule s is then:

$$P \geq Max \left(\begin{array}{c} Max_{c \in \zeta} \left(\dfrac{\left(\sum_{\forall o_i \in O \text{ and } e_{o_i, o_j} \in c} d(o_i) \right)}{\left(\sum_{e_{o_i, o_j} \in c} w(e_{o_i, o_j}) \right)} \right) \\ , \\ Max_{m = 1, 2, ..., \mu} \left(\left(\sum_{(\forall o_i \in O \text{ and } \beta(o_i) = \mu_m)} d(o_i) \right) / |\mu_m| \right) \end{array} \right) . \quad (15)$$

6. MINIMIZING LATENCY UNDER RESOURCE CONSTRAINTS AND FOR A TARGET INITIATION INTERVAL

As we have stated in Section 1, there are two parameters related to the technique of software pipelining: Latency (L) which is the time required to execute all the instructions that constitute the body of the loop, and the Initiation Interval (II) which is the interval of time that separates the start execution time of each two consecutive iterations of the loop. Figure 3 illustrates the software pipelining technique as well as L and II. For loops that execute for a large number of times, applying that technique to a loop leads to a new loop called New-Loop on Figure 3. While each iteration of the original loop requires L units of time to execute, the execution of each iteration of New-Loop requires only II units of time. We have $L \geq II$, which justifies why the software pipelining technique allows to reduce the total execution time of an original loop. Before New-Loop appears, instructions from some first-iterations of the original loop must first be executed. Those instructions form the part called Prologue on Figure 3. Once New-Loop terminates executing, non-executed-yet instructions from some last-iterations of the original loop must be executed. Those instructions constitute the part called Epilogue on Figure 3.

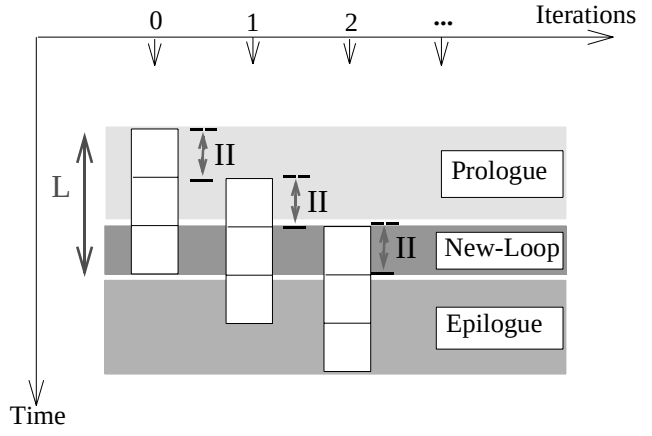


Figure 3. Illustration of software pipelining and its important parameters.

With the help of Figure 3, one can deduce that the New-Loop appears at the iteration number

$$(\lceil L / II \rceil - 1) \quad (16)$$

of the original loop, where $\lceil x \rceil$ denotes the ceiling of x. From expression (16), it is then clear that the values of L and II have an impact on the size (in terms of number of instructions) of both the Prologue and the Epilogue if one wants to use software pipelining to minimize the run-time of a loop. Consequently, those values have an impact on the size of the software-pipelined code. Reducing the size of that code is very important in the case of embedded systems as well as of system-on-chip. 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. For a fixed II, it is then clear that to minimize the code size, one needs to minimize L. One of our objectives in this paper is then to minimize the code size by minimizing L for a certain target value of II.

Many approaches are proposed to realize the software pipelining technique. As it can be observed from Figure 3, realizing that technique transforms to finding a valid periodic schedule with period equal to II. Regarding the relationship between that schedule and L, with the help of Figure 3 and the definition of $s_0(o_i)$ in Section 5, we have that:

$$s_0(o_i) + d(o_i) \leq (L + 1), \forall o_i \in O . \quad (17)$$

The problem of realizing the software pipelining technique while minimizing the size of the code (by minimizing L for a given II as explained above) transforms to the problem of determining a valid periodic schedule with a period II and a latency L (as defined by inequality (17)) such that L is minimal for a given value of II. The value of II is given by the user or computed by automatically trying various values starting from a lower bound, such as the one given by expression (15), until a minimal value for L is found. The latter problem constitutes our target in the rest of this section. We stated it in another manner as follows:

Problem 1: *Given a directed cyclic graph $G = (O, E, d, w)$ modeling a loop, our objective is to find a valid periodic schedule s with a target period II and a minimal latency L, under resources constrains.*

Problem 1 can be solved by solving the following optimization problem:

$$Minimize\ (L) \quad (18)$$

Subject to:

Constraint #1:

Each vertex in $G = (O, E, d, w)$ must have an unique start execution time.

Constraint #2:

The schedule s must satisfy data dependency constraints.

Constraint #3:

The schedule s must satisfy resource constraints.

Constraint #4:

Each vertex in $G = (O, E, d, w)$ must finish executing no later than $(L + 1)$.

Figure 4. Solving *Problem 1* as an optimization problem.

We will start by first transforming the informal definition of the above optimization problem to a formal one. The resulting formal definition is an Integer Linear Program (ILP). Normally that ILP must produce a solution to *Problem 1* with an absolute minimal value for *L*. We will show that this is not the case. In fact, that ILP will produce a relative minimal value for *L*, which is an absolute minimal value for *L* relative to the directed cyclic graph that was used. To avoid that situation, we will unify that ILP and basic retiming to produce another ILP that will always produce an optimal solution to *Problem 1* (the minimal value for *L* will not be sensitive to the graph used). The main idea is that instead of solving *Problem 1* using the given directed cyclic graph, we will solve it using a retimed version of that graph, where the retiming function to be used is computed during the schedule determination.

We now focus on deriving an ILP, a formal version of Figure 4. Let us start by translating constraint #1 to a formal constraint. We are looking for a schedule s as the one defined by equation (7). Since the period is given, then what we still need to compute is $s_0(o_i)$ for each $o_i \in O$. Since l is an upper bound on the latency L to be minimized, we have from equation (17) the following:

$$s_0(o_i) + d(o_i) \le (l + 1),\ \forall o_i \in O . \tag{19}$$

From equations (8) and (19), we deduce that:

$$1 \le s_0(o_i) \le (l + 1 - d(o_i)),\ \forall o_i \in O . \tag{20}$$

By definition of the schedule s, we have $s_0(o_i) \in N$ for each $o_i \in O$. Hence, using binary variables $x_{o_i, t(o_i)}$ and expression (20), we can then write each $s_0(o_i)$ as follows:

$$s_0(o_i) = \sum_{t(o_i) = 1}^{(l + 1 - d(o_i))} t(o_i) \cdot x_{o_i, t(o_i)}, \qquad \forall o_i \in O , \tag{21}$$

$$\sum_{t(o_i) = 1}^{(l + 1 - d(o_i))} x_{o_i, t(o_i)} = 1,\ \forall o_i \in O , \tag{22}$$

and

$$x_{o_i, t(o_i)} \in \{0, 1\},\ \forall o_i \in O, t(o_i) = 1, 2, ..., (l + 1 - d(o_i)) . \tag{23}$$

Constrain #1 in Figure 4 is now formally defined by expressions (21), (22) and (23).

We focus now on transforming Constraint #2 of Figure 4 to a formal one. The schedule s must satisfy data dependency constrains. Hence, inequality (10) must be met. By combining expressions (10) and (21), the data dependency constraints are:

$$\left(\sum_{t(o_j) = 1}^{(l + 1 - d(o_j))} t(o_j) \cdot x_{o_j, t(o_j)}\right) - \left(\sum_{t(o_i) = 1}^{(l + 1 - d(o_i))} t(o_i) \cdot x_{o_i, t(o_i)}\right) \ge d(o_i) - II \cdot w(e_{o_i, o_j}),\ \forall e_{o_i, o_j} \in E \tag{24}$$

Recall that II is the period of the schedule.

We focus now on devising a formal version of the resource constraints expressed by Constraint#3 in Figure 4. The schedule s must be computed in a such way that at any time $t = 1, 2, ..., (l + 1)$, the number of instructions that are executing on the class of computational elements, μ_k, must not exceed $|\mu_k|$ (the number of computational elements of that class). We derive a mathematical formula for resource constraints as follows. Any instruction $o_i \in O$ that is executing at time t implies that o_i has started to execute somewhere in the discrete interval $\{Max(1, (t - d(o_i) + 1)), ..., t\}$, which transforms to:

$$\sum_{t(o_i) = Max(1, (t - d(o_i) + 1))}^{t} x_{o_i, t(o_i)} = 1, \quad \forall o_i \in O, \quad t = 1, 2, ..., (l + 1) . \tag{25}$$

From expression (20), any instruction $o_i \in O$ must start executing no later than $(l + 1 - d(o_i))$. Thus, equation (25) transforms to:

$$\sum_{t(o_i) = Max(1, (t - d(o_i) + 1))}^{Min((l + 1 - d(o_i)), t)} x_{o_i, t(o_i)} = 1, \quad \forall o_i \in O, \quad t = 1, 2, ..., (l + 1) . \tag{26}$$

Software pipelining allows to start executing an iteration of the original loop before the previous iteration has finished its execution. Consequently, instructions that are executing at any time t can be classified into two classes: $C_{t,1}$ and $C_{t,2}$. The class $C_{t,1}$ contains instructions belonging to the set of instructions of the first iteration of the original loop (i.e., no instance of anyone of those instructions is executed before). The class $C_{t,2}$ contains instructions from iterations of the original loop that are not from its first iteration (i.e., the j^{th} instance of an instruction is executing, where $j \ge 2$). The number of instructions that are executing at any time t using the class of computational elements μ_k is the sum of some instructions from $C_{t,1}$ and some instructions from $C_{t,2}$. Expression (26) holds for the case of instructions belonging to class $C_{t,1}$. Hence, the number of instructions belonging to class $C_{t,1}$ that are executing (at any time t) using the class of computational elements μ_k is given by the following formula:

$$\sum_{\{\forall o_i \in O \text{ and } \beta(o_i) = \mu_k\}} \left(\sum_{t(o_i) = Max(1, (t - d(o_i) + 1))}^{Min((l + 1 - d(o_i)), t)} x_{o_i, t(o_i)} \right)$$
$$k = 1, 2, ..., \mu, \qquad t = 1, 2, ..., (l + 1) \tag{27}$$

We focus now on deriving the number of instructions belonging to class $C_{t,2}$ that are executing (at any time t) using the class of computational elements μ_k. The schedule s is periodic with period II. Hence, the class $C_{t,2}$ is empty in the time interval $[1, II]$. $C_{t,2}$ is not empty starting at time $t > II$. As stated above, any instruction $o_i \in O$ that is executing at time t implies that o_i has started to execute somewhere in the discrete interval $\{Max(1, (t - d(o_i) + 1)), ..., t\}$. Since $o_i \in C_{t,2}$, this means that some instances of o_i are executing and have been executed in the discrete interval $\{Max(1, (t - d(o_i) + 1 - n \cdot II)), ..., (t - n \cdot II)\}$, where $1 \le n \le (\lceil t/II \rceil - 1)$ (i.e., derived using Figure 3 and expression (16)). This implies that:

$$\sum_{n = 1}^{\lceil t/II \rceil - 1} \left(\sum_{t(o_i) = Max(1, (t - d(o_i) + 1 - n \cdot II))}^{t - n \cdot II} x_{o_i, t(o_i)} \right) = 1,$$
$$\forall o_i \in O, \quad t = (II + 1), (II + 2), ..., (l + 1) \tag{28}$$

Let δ_t be a 0-1 known variable defined as follows:

$$\delta_t = Min(1, \lfloor (t-1)/II \rfloor), \qquad t = 1, 2, ..., (l+1), \quad (29)$$

where $\lfloor x \rfloor$ denotes the floor of x. Note that $\delta_{o_i, t}$ is 0 when $t \le II$, and 1 otherwise. Hence, equation (28) can be re-written as follows:

$$\sum_{n=1}^{\lceil t/II \rceil - 1} \left(\sum_{t(o_i) = Max(1, (t - d(o_i) + 1 - n \cdot II))}^{t - n \cdot II} \delta_t \cdot x_{o_i, t(o_i)} \right) = 1, \quad \forall o_i \in O, \quad t = 1, 2, ..., (l+1) \quad (30)$$

As we did for the case of class $C_{t,1}$, the number of instructions belonging to class $C_{t,2}$ that are executing (at any time t) using the class of computational elements μ_k is given by the following formula:

$$\sum_{\{\forall o_i \in O \text{ and } \beta(o_i) = \mu_k\}} \sum_{n=1}^{\lceil t/II \rceil - 1} \left(\sum_{t(o_i) = Max(1, (t - d(o_i) + 1 - n \cdot II))}^{t - n \cdot II} \delta_t \cdot x_{o_i, t(o_i)} \right), \quad k = 1, 2, ..., \mu, \quad t = 1, 2, ..., (l+1) \quad (31)$$

Expressions (27) and (31) give the number of instructions that are executing at any time t using the class of computational elements μ_k, $k = 1, 2, ..., \mu$. That number must not exceed $|\mu_k|$. Hence, using (27) and (31), the resource constraints to be met by the schedule s are then formally defined as follows:

$$\sum_{\{\forall o_i \in O \text{ and } \beta(o_i) = \mu_k\}} \left(\left(\sum_{t(o_i) = Max(1, (t - d(o_i) + 1))}^{Min((l + 1 - d(o_i)), t)} x_{o_i, t(o_i)} \right) + \left(\sum_{n=1}^{\lceil t/II \rceil - 1} \left(\sum_{t(o_i) = Max(1, (t - d(o_i) + 1 - n \cdot II))}^{t - n \cdot II} \delta_t \cdot x_{o_i, t(o_i)} \right) \right) \right) \le |\mu_k|, \quad k = 1, 2, ..., \mu, \quad t = 1, 2, ..., (l+1) \quad (32)$$

A formal version for Constraint #4 of Figure 4 can be done by using expressions (19) and (21), and replacing l in the right hand side of (21) by L. We then obtain:

$$\left(\sum_{t(o_i) = 1}^{(l + 1 - d(o_i))} t(o_i) \cdot x_{o_i, t(o_i)} \right) + d(o_i) \le (L+1), \forall o_i \in O. \quad (33)$$

All the constraints in Figure 4 are now expressed mathematically. The resulting ILP is given in Figure 5.

$$Minimize\ (L) \quad (34)$$

Subject to:

Constraint #1: (21), (22) and (23).

(Expression (21) can be omitted since it is just a definition that is already replaced in the other constraints).

Constraint #2: (24)

Constraint #3: (32)

Constraint #4: (33)

Figure 5. An ILP derived from Figure 4.

The ILP of Figure 5 depends on II. To solve it, we then need to fix II. If the value of II is not provided by the user, then the following algorithm can be used to determine a such value, and solve this ILP.

Algorithm: Solve_the_ILP

Begin

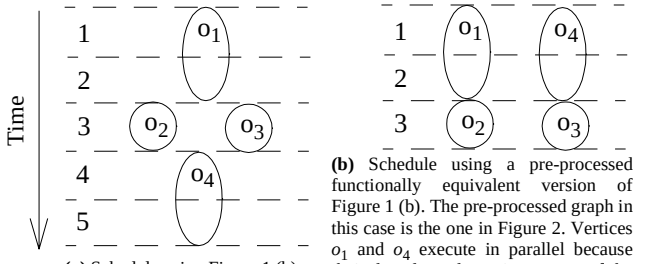
1. A tight value for l could help in reducing the run-time for solving the ILP. Compute a tight value for l by, for instance, using one of some known heuristics for the resource-constrained software pipelining problem. Else, use l defined in Section 4.
2. Fix II to its lower bound using (15). Without loss of generality, we assume that II has an integer value. If $II = (a/b)$, then one can unroll the loop b times, or fix II to the ceiling of (a/b).
3. Solve the ILP using the current value of l and II.
   - 3.1 If no solution is possible, then increment II by 1 and go to step 3. Instead of incrementing II by 1, a binary search in the interval [*value found in step 2*, l] could be used to speed up the algorithm.
   - 3.2 Else report the solution and exit.

End.

Lemma 1 : *The ILP of Figure 5 produces a relative optimal solution to Problem 1.*

Proof: Assume that we have 2 adders and 2 multipliers. Using the graph of Figure 1 (b), the ILP in Figure 5 produces the schedule depicted on Figure 6 (a), which has $L = 5$. However, it is possible to get a schedule like the one of Figure 6 (b) with $L = 3$, by first pre-processing the graph before passing it to the ILP.

❑



(a) Schedule using Figure 1 (b).

(b) Schedule using a pre-processed functionally equivalent version of Figure 1 (b). The pre-processed graph in this case is the one in Figure 2. Vertices o_1 and o_4 execute in parallel because their data dependency is now not of the same iteration since $w(e_{o_4, o_1}) = 1$.

Figure 6. Schedule for two functionally equivalent graphs.

One might want to know why the input graph leads the ILP in missing the absolute minimal value for L? The answer is that the input graph imposes already a partial sequencing of the vertices (instructions of the loop's body). As one can deduce from expression (24), any two vertices of the graph that are connected by an arc having a weight equal to zero can never execute in the same time even if we have unlimited number of resources (this because the destination of that arc can start executing only after the source of the arc finishes executing). To avoid that situation, one then needs to reduce the number of arcs having a weight equal to zero. More precisely, one needs to reduce the length (in terms of time units) of

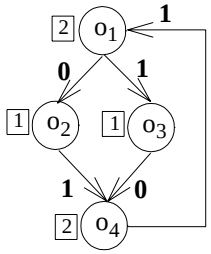
paths composed by arcs of weight equal to zero (0-weight paths); this task is nothing else than the pre-processing we have mentioned above. The question now is how can that pre-processing be done? We focus in the rest of this section on answering that question.

As we have introduced in Section 3, there is a close relationship between a directed cyclic graph modeling a loop and a directed cyclic graph modeling a synchronous sequential digital circuit. By thinking of the former graph as a directed cyclic graph modeling a synchronous sequential digital circuit, the weight of each arc can then be viewed as the number of registers on that arc. In this case, basic retiming can be used to move registers, thereby defining one possible pre-processing we are looking for. The pre-processing we did to obtain Figure 6 (b) is in fact a retiming, and the pre-processed graph passed to the ILP in this case is the one of Figure 2.

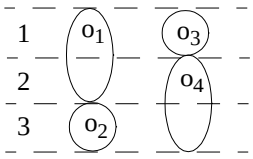
In the case of limited resources, the pre-processing must be done during the schedule determination. Indeed, let us assume that we have now 1 adder and 2 multipliers instead of 2 adders and 2 multipliers assumed in proof of Lemma 1. Graphs in Figure 2 and Figure 7 (a) are two possible retimed graphs of Figure 1 (b). The graph in Figure 2 is used to produce Figure 6 (b). If we again use graph in Figure 2 for the new resource constraints, we obtain a schedule with $L = 4$. Vertices o_1, o_2, o_3 and o_4 will be assigned to time steps 1, 3, 4, and 3, respectively. Vertices o_2 and o_3 are serialized since we have only 1 adder. However, if we use Figure 7 (b) we obtain the schedule in Figure 7 (b) with only $L = 3$. Hence, it is then clear that retiming cannot be de-coupled from the schedule determination step.

We now agree that the pre-processing must be done during the schedule determination. The question is how can this be done? The pre-processing in our case is computing a retiming to be applied to the vertices of the graph. The retiming must be valid which means it must satisfy expression (3). The weight of each arc after any retiming is defined by equation (1). Since the retiming will be computed during the schedule determination, this implies that the weight of each arc is now an unknown variable but that variable is equal to:

$$w(e_{o_i, o_j}) + r(o_j) - r(o_i),\ \forall e_{o_i, o_j} \in E\,. \tag{35}$$



(a) Retiming Figure 1(b) by applying the retiming vector $\{0, 0, 1, 1\}$ to the vector of vertices $\{o_1, o_2, o_3, o_4\}$.



(b) Schedule of **(a)** using 1 adder and 2 multipliers.

Figure 7. The right retiming for the pre-processing can always be obtained only if retiming and scheduling were unified.

In the ILP of Figure 5, the only constraints that depend on the weight of arcs are the data dependency constraints which are expressed by (24). By using (24) and (35), we obtain

$$\left(\sum_{t(o_j)=1}^{(l-d(o_j))} t(o_j)\cdot x_{o_j, t(o_j)}\right) - \left(\sum_{t(o_i)=1}^{(l-d(o_i))} t(o_i)\cdot x_{o_i, t(o_i)}\right)$$
$$\geq\ (d(o_i) - II\cdot(w(e_{o_i, o_j}) + r(o_j) - r(o_i))),\ \forall e_{o_i, o_j} \in E \tag{36}$$

where (i.e., (37) expresses the fact that retiming must be valid)

$$r(o_j) - r(o_i) \geq -w(e_{o_i, o_j}),\ \forall e_{o_i, o_j} \in E\,, \tag{37}$$

and

$$r_{o_i} \in Z,\ \forall o_i \in O\,. \tag{38}$$

By putting together all the development above, an ILP that combines both the scheduling and the pre-processing (i.e., applying basic retiming) is given by Figure 8.

$$Minimize\ (L) \tag{39}$$

Subject to:

Constraint #1:	(22) and (23).
Constraint #2:	(36)
Constraint #3:	(32)
Constraint #4:	(33)
Valid retiming:	(37)
Retiming takes values on Z:	(38)

Figure 8. Unifying scheduling and retiming to optimally solve *Problem1*.

The ILP of Figure 8 depends on II. To solve it, we then need to fix II. Again, if the value of II is not provided by the user, then the algorithm *Solve_the_ILP* can be used to determine a such value, and solve this ILP.

7. MINIMIZING PEAK POWER UNDER RESOURCES, LATENCY, AND INITIATION INTERVAL CONSTRAINTS

Suppose that we want to accelerate the loop in Figure 1 (a) to achieve a latency $L = 4$ and initiation interval $II = 4$, using two adders and two multipliers. And assume that each adder (multiplier) has execution delay equal to 1ns (2ns) and power consumption equal to 20 mW (100mW). We previously showed that without retiming, applying software pipelining on that loop will lead to $L = 5$. A possible retiming that allows to obtain $L = 4$ is the one that leads to the graph in Figure 2. Using the graph in Figure 2, we obtain two possible schedules given by Figure 9. These schedules satisfy timing constraints (i.e., $L = II = 4$), but differ in terms of peak power. The power consumed at each time step is given on the right hand side of each schedule. The peak power for Figure 9 (a) is 100mW while it is only 70mW for Figure 9 (b). Our objective is then to propose an approach that allows to compute periodic schedules (i.e., to realize software pipelining) that meet timing and resource constraints but require the minimum peak power consumption. More precisely, our objective is to solve the following problem:

Problem 2: *Given a directed cyclic graph $G = (O, E, d, w)$ modeling a loop, our objective is to find a valid periodic schedule s with a target period II and a latency L, under resources constrains, but with a minimal peak power consumption.*

The left hand side of expression (32) gives the number of instructions that are executing at any time $t = 1, 2, ..., (l + 1)$ using the class of computational elements μ_k, $k = 1, 2, ..., \mu$. If we do not take care about which class of computational elements is used at time t, then from (32) the number of instructions that are executing at time t is:

$$\left(\sum_{t(o_i) = Max(1, (t - d(o_i) + 1))}^{Min((l + 1 - d(o_i)), t)} x_{o_i, t(o_i)} \right) + \left(\sum_{n = 1}^{\lceil t/II \rceil - 1} \left(\sum_{t(o_i) = Max(1, (t - d(o_i) + 1 - n \cdot II))}^{t - n \cdot II} \delta_t \cdot x_{o_i, t(o_i)} \right) \right)$$

$$\forall o_i \in O, \quad t = 1, 2, ..., (l + 1) \qquad (40)$$

Let $\rho_{o_i, t}$ be the power consumed by the operation o_i at any time step $t = 1, 2, ..., (l + 1)$. The total power ρ_t consumed by operations that are executing at any time t is the sum of their $\rho_{o_i, t}$'s. Hence, using (40), ρ_t is formally defined as:

$$\rho_t = \left(\left(\sum_{t(o_i) = Max(1, (t - d(o_i) + 1))}^{Min((l + 1 - d(o_i)), t)} \rho_{o_i, t} \cdot x_{o_i, t(o_i)} \right) + \left(\sum_{n = 1}^{\lceil t/II \rceil - 1} \left(\sum_{t(o_i) = Max(1, (t - d(o_i) + 1 - n \cdot II))}^{t - n \cdot II} \rho_{o_i, t} \cdot \delta_t \cdot x_{o_i, t(o_i)} \right) \right) \right) .$$

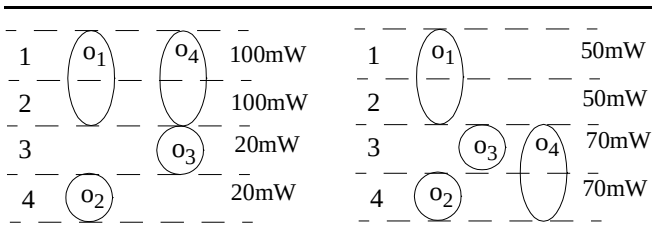
$$\forall o_i \in O, \quad t = 1, 2, ..., (l + 1) \qquad (41)$$

The peak power is defined as: $PeakPower = Max_{t = 1, 2, ..., (l + 1)}(\rho_t)$. This implies that:

$$PeakPower \geq \rho_t, \quad t = 1, 2, ..., (l + 1) \ . \qquad (42)$$

When the latency L is fixed to a target value, then expression (39) can be omitted, and the resulting ILP allows to compute a valid periodic schedule with period II (which is the initiation interval), and latency L. That resulting ILP can then be extended to solve *Problem 2*. Indeed, what we have to do is to add expressions (41) and (42) to the constraints of that resulting ILP and then replace (39) by the following expression:

$$Minimize(PeakPower) . \qquad (43)$$



(a) peak power = 100mW. **(b)** peak power = 70mW.

Figure 9. Schedules may differ in terms of peak power.

8. DISCUSSIONS AND RELATED WORK

Software pipelining is not a new technique. It has been proposed since many years to optimize timing for parallel processors like VLIW and superscalar ones. Many approaches has been proposed to the problem of realizing that technique in the case of unlimited as well as limited number of resources. In the case of unlimited number of resources, that problem is optimally solvable in polynomial run-time. The problem is NP-hard in general in the case of limited resources. Due to space limitation, the reader can consult [4][8] for a literature review of many proposed approaches to that problem. We restrict ourself to approaches that are close to the problem we address in this paper.

Rotation scheduling [7] is a heuristic that realizes software pipelining under resource constraints with a shorter initiation interval II. While the approach in [6] starts with a tight II an iteratively increases it when a schedule cannot be found, *rotation scheduling* finds an approximate solution to the problem and then iteratively improve it. The value of II is iteratively shortened by rotating some vertices of the graph and then re-scheduling them. Each rotation is in fact a retiming. The heuristic does not control the latency L of the schedule, which might lead to a large value of L. Recall that having a large value for L implies that the code size of the *Prologue* and the *Epilogue* after applying software pipelining will be large. An approach to reduce the code size of the *Prologue* and *Epilogue* is proposed in [3].

To the best of our knowledge, this is the first paper that addresses the problem of minimizing peak power consumption for a target latency and initiation interval, and under resource constraints.

9. EXPERIMENTAL RESULTS

The objective of this experimentation is to test the effectiveness, in terms of relative timing improvement (which has a relationship with code size of the *Prologue* and the *Epilogue* of the optimized loop), relative peak power reduction, and execution time of the proposed approach. To this end, we think of cyclic graphs modeling some real-life filters as cyclic graphs modeling loops. The names of these filters are given in the first column of Tables 1 and 2.

We assume that we have an hypothetical processor with 3 adders and 2 multipliers. Each adder has execution delay equal to 1ns and power consumption equal to 20mW. Each multiplier has execution delay equal to 2ns and power consumption equal to 100mW.

All the experiments were done using an UltraSparc 10 with 1GB RAM. For results in Table 1 and 2, we developed a C++ tool and implemented the algorithm *Solve_ILP* to solve ILPs in Figures 5 and 8 as well as the one described in Section 7. The input of the tool is a graph modeling each filter, as well as resource constraints and their related features. For step 3 of the algorithm *Solve_ILP*, we used the *lp_solve* tool available at [2].

For the case of Table 1, the C++ tool reports a lower and an upper bounds on the latency L (the second and third columns, respectively), a lower bound on II using the right hand side of expression (15) (see fourth column), the value of II used to compute the schedule (fifth column; it contains the value of II used to solve the ILP in Figures 5 and 8). Columns 6 and 7 of Table 1 report the value of L and the run-time when the ILP in Figure 5 is solved. Columns 8 and 9 report the value of L and the run-time when the ILP in Figure 8 is solved. Column 10 reports relative reduction of the latency defined as $(L(\text{column } 6) - L(\text{column } 8)) / (L(\text{column } 8)) \times 100$. As it can be observed, relative reduction of the latency is 60.19% on average, and the run-time for solving the two ILPs is less than 30s on average.

The C++ tool is also used to assess the approach proposed in Section 7 to minimize peak power consumption. We use the same circuits as those used in Table 1. We fixed II and L to the minimal values found in Table1 (see column 2 and 3 of Table 2). Obtained results are summarized by Table 2. For column 4, we first solve

Figure 8 without (39) and then we compute the peak power. The run-time for this task is reported in column 5. For column 6, we solve the ILP proposed in Section 7 to minimize peak power consumption, and then we compute the peak power of the resulting schedule. The run-time for this task is reported in column 7. Column 8 reports relative reduction of peak power, which is defined here as:

$$\frac{(PeakPower(\text{column 4}) - PeakPower(\text{column 6}))}{PeakPower(\text{column 6})} \times 100 .$$

As we can observe from Table 2, the proposed approach is able to reduce peak power consumption by 13.17% on average even thought L and II are set to their minimal values. If L and II are set to values greater than the used ones, then more peak power reduction could be obtained. Indeed, for the circuit named *Example* in Table 2, this table shows that peak power was not reduced. However, in Section 7 we showed that peak power for that circuit can be reduced from 100mW to 70mW when $L = II = 4$.

Table 1. Case of Minimizing Latency.

Circuit Name	Lower Bound On Latency	Upper Bound On Latency	Lower Bound On Initiation Interval (II)	II Used	M1: Latency Without Retiming	Run-Time For M1 (Sec.)	M2: Latency With Retiming	Run-Time For M2 (Sec.)	Latency Improvement (Relative-Improvement In %)
Example	3	6	3	3	5	0.01	3	0.01	66.67
Correlator_Order_3	3	7	3	3	5	0.02	3	0.06	66.67
Correlator_Order_4	4	10	3	3	7	0.08	4	0.36	75.00
Correlator_Order_5	4	13	4	4	9	0.34	4	1.82	125.00
BiquadratiqFiltr	4	12	4	4	6	0.56	4	135	50.00
PolynomDivider	4	13	4	4	7	0.24	4	5.55	75.00
TransFIR	3	8	3	3	5	0.1	4	0.63	25.00
ThreTapNonRecDigiFiltr	4	8	4	4	5	0.09	4	0.03	25.00
DES	6	17	6	6	8	38.27	6	80.3	33.33
					Average	**4.41**		**24.86**	**60.19**

Table 2. Case of Minimizing Peak Power.

Circuit Name	Initiation Interval Used	Latency Used	M3: PeakPower Not Minimized (In mW) (Retiming Is Used)	Run-Time For M3 (Sec.)	M4: PeakPower Minimized (In mW) (Retiming Is Used)	Run-Time For M4 (Sec.)	Relative PeakPower Reduction (%)
Example	3	3	100	0.02	100	0.01	0.00
Correlator_Order_3	3	3	120	0.02	100	0.03	20.00
Correlator_Order_4	3	4	140	0.15	140	0.74	0.00
Correlator_Order_5	4	4	160	0.1	140	6.59	14.29
BiquadratiqFiltr	4	4	160	0.21	120	0.13	33.33
PolynomDivider	4	4	160	0.1	140	6.41	14.29
TransFIR	3	4	120	0.14	120	0.48	0.00
ThreTapNonRecDigiFiltr	4	4	120	0.06	100	0.2	20.00
DES	6	6	140	5.24	120	8.19	16.67
			Average	**0.67**		**2.53**	**13.17**

10. CONCLUSIONS

For loops optimized by software pipelining, we have showed that there is a relationship between the latency and the code size. An increase of latency implies an increase of the code size. Also, decreasing latency implies reducing the idleness of computational elements. We have showed that optimizing loops by only applying software pipelining can lead to sub-optimal value of the latency compared to the case of unifying basic retiming and software pipelining. We have proposed an ILP to realize that unification.

For software pipelined loops, concurrency between instructions increases, which implies that more computational elements are operating at the same time. Thus, peak power would increase. However, by choosing a good schedule, it is possible to reduce peak power consumption while still having the same target timings. Indeed, peak power can be reduced by using the ILP that we have proposed in this paper. To the best of our knowledge, this proposed approach is the first one in the literature that deals with peak power consumption in the context of software pipelining.

The proposed ILPs are flexible and could be extended to address other problems related to software pipelining. Indeed, as an example of such problems is the problem of reducing the number of registers. That problem can be solved with the proposed ILPs by adding constraints into the constraints of these ILPs.

ACKNOWLEDGEMENT

The authors would like to thank the three anonymous reviewers for their valuable comments from which this paper has benefited.

REFERENCES

[1] C.E. Leiserson and J.B. Saxe, "Retiming Synchronous Circuitry," *Algorithmica*, pp. 5-35, Jan., 1991.

[2] The LP_Solve Tool: ftp://ftp.ics.ele.tue.nl/pub/lp_solve/

[3] Q.Zhuge, B.Xiao, E.H.MSha, "Code size reduction technique and implementation for software-pipelined DSP applications," *ACM Trans. on Embedded Computing Systems*, V.2, N.4, November 2003, pp. 590-613.

[4] V.Allan, R.B.Jones, R.M.Lee, S.J.Allan, "Software Pipelining," *ACM Computing Surveys*, Vol. 27, No. 3, September 1995, pp. 367-432.

[5] A.Dasdan, R.K.Gupta, "Faster Maximum and Minimum Mean Cycle Algorithms for System Performance Analysis," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, V.17, N.10, Oct. 1998.

[6] B.R.Rau, "Iterative Modulo Scheduling," *International Journal of Parallel Programming*, 24 (1), pp. 3-64, 1996.

[7] L.F.Chao, A.S.LaPaugh, E.H.M.Sha, "Rotation scheduling: A loop pipelining algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, V.16, N.3, March 1997, pp. 229-239.

[8] B.R.Rau, J.A.Fisher, "Instruction-level parallel processing: history, overview, and perspective," *The Journal of Supercomputing*, V.7, N.1, 1993, pp. 9-50.