# MPCS '96

## Proceedings of the

# Second International Conference on Massively Parallel Computing Systems

*Ischia, Italy*

*May 6 - 9, 1996*

Convened by:

**Istituto di Ricerca sui Sistemi Informatici Paralleli**
**Istituto di Fisica Cosmica e Tecnologie Relative**

In cooperation with:

EUROMICRO

IEEE COMPUTER SOCIETY
50 YEARS OF SERVICE • 1946-1996

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

IEEE Computer Society Press
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1264

IEEE Computer Society Press Order Number PR07600
Library of Congress Number 96-79508
ISBN 0-8186-7600-0 (paper)
Microfiche ISBN 0-8186-7602-7

Editorial production by Regina Sipple and Penny Storms
Cover by Joseph Daigle/Studio Productions
Printed in the United States of America by KNI, Inc.

The Institute of Electrical and Electronics Engineers, Inc.

# Maximum-Throughput Software Pipelining *

Fermín Sánchez and Jordi Cortadella
Department d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona, Spain
fermin@ac.upc.es, jordic@ac.upc.es

## Abstract

*This paper presents UNRET (*unrolling and retiming*), a resource-constrained software pipelining approach aimed at finding a loop schedule with maximum throughput and minimum register requirements. UNRET works in two phases.*

*First, a pipelined loop schedule with maximum throughput is found for a given set of resources. To do this, different unrolling degrees are explored in decreasing order of expected throughput. Farey's series are used to perform such an exploration. For a given unrolling degree and an expected initiation interval, the software pipelining algorithm successively retimes the loop, obtaining different configurations. Scheduling is done for each configuration, thus performing a large exploration of the solution space.*

*Second, the number of registers required by the schedule is reduced. This is done also in two steps, by reducing the iteration time and by rescheduling some operations in the schedule, attempting to reduce the maximum number of variables whose lifetime overlap at any cycle.*

*The algorithm runs in polynomial time. The effectiveness of the proposed approach is shown by presenting results on well-known benchmarks. Results show that UNRET may obtain faster and better schedules than other approaches, also reducing the register requirements. Results also show that UNRET obtains optimal results in most cases.*

## 1 Introduction

*Software pipelining* is a family of techniques aimed at the overlapped execution of several iterations of a loop. These techniques are suitable for compilers for parallel architectures and for high-level synthesis (HLS) of VLSI circuits. In both areas, the concerns of minimizing the *initiation interval* (*II*), reducing the number of registers, and finding efficient schedules with resource constraints are common. Techniques such as *modulo scheduling* [17] or Lam's algorithm [13], among others, have been proposed for parallel architectures, while *loop folding* [7], *loop winding* [5] or *functional pipelining* [10] have been devised for HLS.

This paper presents *UNRET* (*unrolling and retiming*), a new approach for software pipelining with resource constraints. The ideas behind *UNRET* have been used in [22] to propose a software pipelining approach with timing constraints. *UNRET* works as follows: first, a lower bound of the *minimum initiation interval* (*MII*) for any schedule is calculated. Following this, the appropriate (not always optimal) *unrolling degree* (*UD*) of the loop is *analytically* computed, in a much more exhaustive manner than previous methods [12, 19]. Pipelining is achieved by *retiming* the unrolled loop. Once a schedule has been found, the number of required registers is reduced while maintaining the throughput. Similar ideas by using integer linear programming approaches (ILP) [11] or loop transformations [3] have been proposed by other authors, but *UNRET* works significantly faster than [11] and more efficiently than [3].

### 1.1 Contributions

This paper presents the following new contributions with regard to previous approaches:

- *UNRET* explores throughput in two dimensions: the *UD* and the *II*. Current approaches that perform loop unrolling used a fixed *UD* or the theoretical optimal *UD* [12]. If a schedule is not found, the expected *II* is increased. *UNRET* improves the technique proposed in [12] by exploring other (suboptimal) *UD*s. This may result in a higher throughput than by just merely increasing the *II*.

- Software pipelining is reduced to the interleaved combination of two decoupled techniques: *retiming* and *scheduling*. Several configurations are explored by

*This work was supported by the Ministry of Education and Science of Spain, under contract CICYT TIC-95-0419

retiming the loop, and scheduling is done for each configuration. Most approaches perform both tasks simultaneously [7, 10, 17], obtaining inferior results because only one configuration of the loop is considered and some scheduling decisions are taken much too soon. *UNRET* obtains optimal schedules with shorter CPU times in most cases, as shown in Section 6.

- Decoupling *retiming* and *scheduling* also results in effective algorithms for register reduction: *Retiming* may reduce the iteration time of the loop by reducing the variable lifetimes which traverse several iterations. *Scheduling* may reduce the maximum number of variables whose lifetime overlap at any cycle.

  Others researchers have proposed techniques that perform software pipelining and register reduction at a time [2, 15]. Our results show that separating both tasks may produce equal or superior results.

The rest of the paper is organized as follows. An overview of *UNRET* is presented in Section 2. The way to explore the solution space in two dimensions is described in Section 3. The software pipelining approach is described in Section 4. Section 5 shows the algorithm proposed to reduce the number of registers required by the schedule. Section 6 compares *UNRET* with other approaches by using several examples. Finally, Section 7 concludes the paper.

## 2 UNRET overview

### 2.1 Representation of a loop

The scope of this work is limited to single nested loops whose body is a basic block. For multiple-nested loops, *UNRET* is applied to the innermost loop. Conditional sentences can be treated by means of *if-conversion* [1]. The architecture has a limited number of functional units (FUs), possibly pipelined. An operation can take several cycles to be completed and may use different FUs.

A loop is represented by a labelled directed dependence graph, $DG(V, E)$. Each vertex $u \in V$ corresponds to an operation of the loop body. Each edge $e \in E$ corresponds to a data dependence between two operations. Labels of the DG are defined by two mappings, $\lambda$ (iteration) and $\delta$ (dependence distance), in the following way:

- $\lambda(u)$, defined on vertices, denotes the iteration to which the execution of $u$ corresponds in the schedule. $\lambda(u) = i$ will be denoted by $u_i$ in the DG.

- $\delta(u, v)$, defined on edges, denotes the number of iterations traversed by the dependence. $\delta(u, v) = 0$ corresponds to an intra-loop dependence (ILD), and it is represented as $u_i \rightarrow v_i$. $\delta(u, v) > 0$ corresponds to a loop-carried dependence (LCD). An LCD of distance $d$ between $u$ and $v$ is represented as $u_i \xrightarrow{d} v_i$.

### 2.2 Comparison among approaches

#### 2.2.1 Software pipelining without previous unrolling

Figure 1(a) depicts a DG with 5 additions. $\delta(E, A) = 3$ indicates that $E_i$ must be executed to completion before $A_{i+3}$ starts. The remaining dependences are ILDs. For the sake of simplicity, let us assume that the loop is executed by using 4 adders which add in one cycle. A pipelined schedule in two cycles (*II*=2) can be easily found, as shown in Figure 1(d). The throughput (average number of iterations executed per cycle) of such a schedule is $Th = \frac{1}{2}$.

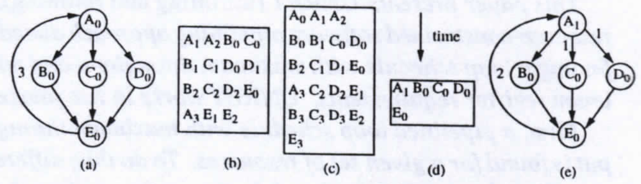

**Figure 1. Example of DG and schedules for 4 adders (a) Example of DG with 5 additions (b) Schedule achieved by UNRET (c) Schedule achieved by using the optimal UD (d) Schedule achieved without unrolling (e) DG corresponding to schedule (d)**

#### 2.2.2 Software pipelining with optimal UD

In order to obtain a minimum initiation interval (*MII*) for the schedule, a new loop representing multiple instances of the loop body is in general required [3]. Unrolling the loop may increase the schedule throughput, but it also produces code and register explosion. Moreover, a prologue and an epilogue are in general necessary to execute an unrolled loop. We will ignore the effect of the prologue and the epilogue on the overall execution time[1]

Two lower bounds on *II* must be taken into account [18]:

- the *minimum initiation interval imposed by resource constraints* (*ResMII*)

  If each iteration of the loop requires using an FU during $C$ cycles, and the architecture has $N$ FUs of such a type, then $II \geq \lceil \frac{C}{N} \rceil$. Therefore, the FU with the maximum such ratio determines a lower bound on *II*.

- the *minimum initiation interval imposed by the cycles formed by the dependences of the loop* (*RecMII*)

  Let us consider a cycle (recurrence) $R$. A feasible schedule must fulfill $II \geq \lceil \frac{ET}{D} \rceil$, where $ET$ is the sum

[1]This neglect can only be done for large loop counts.

of the execution times of the operations in $R$ and $D$ is the sum of the distances of its dependences [17]. The recurrence with the maximum such ratio determines another lower bound on $II$.

The *MII* of the loop is the maximum of the previous lower bounds. In the example of Figure 1(a), *MII*$=\frac{5}{4}$, thus indicating that 4 iterations may be initiated every 5 cycles. In order to find such a schedule, the loop is unrolled 4 times and pipelined, looking for a schedule in 5 cycles [19]. Unfortunately, such a schedule does not exist [4].

When the expected schedule is not found, current approaches [9, 13, 17, 19] increase (by 1 or more than 1 unit [9]) the expected $II$ and try to find a longer schedule by using the same $UD$. The loop is pipelined again, looking now for a schedule of 4 iterations in 6 cycles. Figure 1(c) shows the found schedule, with a throughput $Th = \frac{2}{3}$. This schedule is still far from the optimal schedule, but is better than the schedule found without previous unrolling.

### 2.2.3 UNRET strategy

Similar to other approaches, *UNRET* computes the *MII* and unrolls the loop 4 times. Let $II_K$ be the initiation interval of a schedule comprising $K$ iterations of the loop body. In order to find a schedule which maximizes the execution throughput, *UNRET* explores pairs ($II_K$,K) in decreasing order of expected throughput. After the pair ($II_K$, K)=(5,4) representing maximum throughput (for which no schedule exists), the first pair explored is ($II_K$, K)=(4,3). Therefore, the loop is unrolled 3 times and pipelined until a schedule in 4 cycles is found, as shown in Figure 1(b). The throughput of such a schedule is $Th = \frac{3}{4}$. Although this schedule is not time-optimal (for the lower bound computed), it is better than that obtained by the other approaches. A significant speedup (1.125) is obtained with regard to [9, 13, 17, 19] due to the more exhaustive exploration of the $UD$.

## 2.3 UNRET algorithm

The strategy followed by *UNRET* is as follows:

1. Calculate *MII*$=\max($*RecMII*, *ResMII*$)$ (section 2.2.2).
2. Find $UD$ $(K)$ and expected $II$ $(II_K)$ which maximize throughput (section 3).
3. Unroll the loop $K$ times.
4. Find new labellings $\lambda$ and $\delta$ and schedule the DG until a schedule in $II_K$ cycles is found (section 4).
5. If no schedule in $II_K$ cycles is found, find new values for $K$ and $II_K$ (throughput is explored in decreasing order), and goto 3 (section 3).
6. If a schedule in $II_K$ cycles is found, reduce the number of required registers (section 5).

The following sections present more details on the main phases of *UNRET*: the calculation of the *UD* for throughput maximization (Section 3), the software pipelining algorithm (Section 4) and the way to reduce the number of required registers (Section 5).

## 3 Optimal UD

The throughput of a schedule is the ratio between the $UD$ of the loop and the length ($II$) of the schedule. The variations in both magnitudes determine the *solution space* (all possible throughputs) for a given loop. When an optimal schedule is not found, the *solution space* must be examined and a criterion must be established to determine when an "acceptable" solution has been found.

We will use the example from section 2 to illustrate how pairs ($II_K$, K) are generated. Figure 2 depicts a diagram representing the pairs to be explored. Each point in the diagram represents a pair ($II_K$, K). *MaxII*, the maximum number of cycles of a schedule, is defined by the designer to limit the number of pairs to explore ($II_K \leq$ *MaxII*). Otherwise, the number of pairs is infinite and, therefore, they cannot be ordered and explored in decreasing order of throughput. Small values of *MaxII* may lead to good solutions [20].

All pairs representing the same throughput fall in a line. The points to explore are contained in the triangle delimited by the line $II_K$= *MaxII*, the $x$ axis ($K = 0$) and the line $Th = \frac{1}{MII}$. Only points with integer values for $K$ and $II_K$ represent valid schedules[2]. For two different pairs belonging to the same line, *UNRET* first explores the pair representing the smallest $UD$.

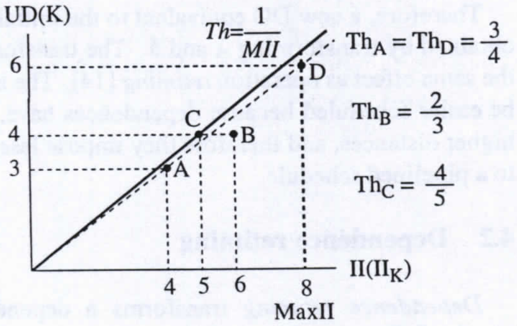

Figure 2. Throughput diagram for the example of Figure 1

Figure 2 represents the throughput achieved by the schedules of the example shown in Figure 1. Point $C$ represents a theoretical optimal (but non-existing) schedule. *UNRET* finds a schedule at point $A$ (3 iterations in 4 cycles). Other

[2] Some of these schedules may not exist.

approaches [9, 13, 17, 19] attempt to find a schedule by increasing $II$, obtaining a solution for point $B$ (4 iterations in 6 cycles) These approaches explore the solution space in one dimension (parallel to the $x$ axis for a fixed $K$), whilst *UNRET* explores two dimensions: $II_K$ and $K$. The mathematical formulation for the generation of pairs ($II_K$, K) is based on *Farey's series* [23]. *Farey's Series* or order $Z$ ($F_Z$) defines the sequence (in increasing order) of all the reduced fractions with nonnegative denominator $\leq Z$. This series contains all the points within the triangle limited by $MaxII = Z$ [20]. For example, $F_5$ is the series of fractions:

$$\frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \cdots$$

Let $\frac{X_i}{Y_i}$ be the $i$th element of the series. $F_Z$ can be generated by the following recurrence:

- The first two elements are $\frac{X_0}{Y_0} = \frac{0}{1}$ and $\frac{X_1}{Y_1} = \frac{1}{Z}$
- The generic term $\frac{X_K}{Y_K}$ can be calculated as:

$$X_{K+2} = \left\lfloor \frac{Y_K + Z}{Y_{K+1}} \right\rfloor \cdot X_{K+1} - X_K \quad Y_{K+2} = \left\lfloor \frac{Y_K + Z}{Y_{K+1}} \right\rfloor \cdot Y_{K+1} - Y_K$$

## 4 Software pipelining

### 4.1 Equivalent DGs

Provided that $A_{i+d} \xrightarrow{d} B_i$ and $A_i \rightarrow B_i$ represent the same dependence in a DG, two different labellings $(\lambda, \delta)$ and $(\lambda', \delta')$ are equivalent (they represent the same loop) if, $\forall (u, v) \in E$, the following condition holds:

$$\lambda(v) - \lambda(u) + \delta(u, v) = \lambda'(v) - \lambda'(u) + \delta'(u, v) \quad (1)$$

Therefore, a new DG equivalent to the initial DG can be obtained by transforming $\lambda$ and $\delta$. The transformation has the same effect as operation *retiming* [14]. The new DG can be easier scheduled because dependences have, in general, higher distances, and therefore they impose less constraints to a pipelined schedule.

### 4.2 Dependence retiming

*Dependence retiming* transforms a dependence $e = (u, v)$ of distance $d$ into another dependence of distance $d + 1$ according to equation (1), by executing the following algorithm:

```
function dependence_retiming(DG, e); { $e = (u, v)$ }
    $\lambda'(u) := \lambda(u) + 1$;
    for each $(u, w) \in E$ do $\delta'(u, w) := \delta(u, w) + 1$;
    for each $(w, u) \in E$ do $\delta'(w, u) := \delta(w, u) - 1$;
endfunction;
```

*Dependence retiming* must be used in edges so that no negative distances are produced in the DG. *Dependence retiming* produces an equivalent DG, since Equation (1) is fulfilled for each edge. Figure 3(a) shows that an ILD not belonging to any recurrence can always be transformed into an LCD. Figure 3(b) shows that an ILD belonging to a recurrence is shifted across the recurrence when *dependence retiming* is used on an edge of the recurrence.

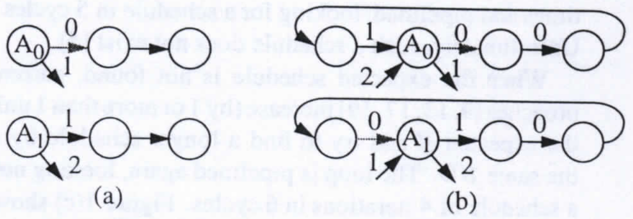

Figure 3. Dependence retiming

### 4.3 Software pipelining algorithm

For an expected $II$, a dependence $e = (u, v)$ is called a *scheduling dependence* (it constrains the scheduling) if $T(u) - II \cdot \delta(e) > 0$, $T(u)$ being the execution time of $u$. Let $G$ and $G'$ be two equivalent graphs. We define $G$ is *better for scheduling* than $G'$ if the critical path (CP) of $G$ is shorter. For equal CP length, we define $G$ is *better than* $G'$ if $G$ has less scheduling dependences than $G'$. To continuously produce *better* graphs, the edge to retime is selected among those that are head or tail of a CP. These edges can be easily transformed in a way such that no negative dependences are created after *dependence retiming*.

As example, DG from Figure 1(e) is better for scheduling than DG from Figure 1(a), and a schedule in two cycles can be found by using four resources, as shown in Figure 1(d). The loop pipelining algorithm selects and retimes dependences until a schedule is found or no further retiming can be done such that a better DG is found. The algorithm is sketched in Figure 4. It executes in $O(V^2E + VE^2)$. To compute the execution time, we have considered that the scheduling algorithm executes in $O(V^2 + VE)$ time, and the repeat loop in *retiming_and_scheduling* is executed $n \cdot |E|$ times, where $n$ is an integer which depends on the distance of the dependences [20].

Retiming implicitly pipelines the loop. Therefore, a pipelined schedule of the loop may be found by simply scheduling the operations in the DG (any known approach, as *list scheduling* [6] or *force directed scheduling* [16], can be used). We use *list scheduling* for its efficiency and low complexity, given that the scheduler may be potentially called many times by *retiming_and_scheduling*.

*UNRET* deals with irregular resource execution patterns. Since the expected initiation interval of the schedule is known in advance, operations may overlap their execution

```
function $retiming\_and\_scheduling(DG, II_K, K)$;
    $DG_{best} := unroll(DG, K)$;
    $DG' := DG_{best}$;
    Repeat
        $S$ :=scheduling($DG'$);
        if found schedule in $II_K$ cycles
            then return $S$ endif;
        $e$:=select an edge for retiming from $DG'$;
        if no edge can be selected
            then return schedule not found endif;
        $DG'$ := dependence_retiming ($DG'$, $e$);
        if $DG'$ is better than $DG_{best}$
            then $DG_{best} := DG'$ endif;
    Forever;
endfunction;
```

**Figure 4. Retiming_and_scheduling algorithm**

among consecutive schedules[3]. This feature allows *UNRET* to find non-rectangular schedules [21]. All the complexity concerning resource constraints, multiple-cycle and pipelined FUs is hidden within the scheduling algorithm. Since other authors have presented several contributions on these topics, no details will be given in the paper.

## 5 Register optimization

### 5.1 Variable lifetime

An absolute lower bound on the number of registers required for a schedule is the maximum number of variables whose lifetimes overlap at any cycle ($MaxLive$). Experiments have shown that $MaxLive$ is very close to the number of registers required after performing register allocation [9]. Therefore, we will use $MaxLive$ to approximate the number of registers required by a schedule.

Let us consider an ILD $e = (u, v)$. The variable lifetime of $e$ may be different according to the target architecture. In a superscalar architecture, the variable lifetime spreads from the starting of $u$ to the starting of $v$. In a VLIW architecture, the variable lifetime spreads from the starting of $u$ to the completion of $v$. Other models can also be defined [20]. For example, in HLS, the variable lifetime spreads from the completion of $u$ to the cycle in which the FUs executing $v$ do not require the input data anymore.

In order to reduce the registers required for a schedule, we use a two-step approach:

1. Variable lifetimes are shortened by reducing the *SPAN*.
2. $MaxLive$ is next reduced by *incremental scheduling*.

[3]For example, an operation can start at cycle *II* and continue the execution at the first cycle of the following iteration.

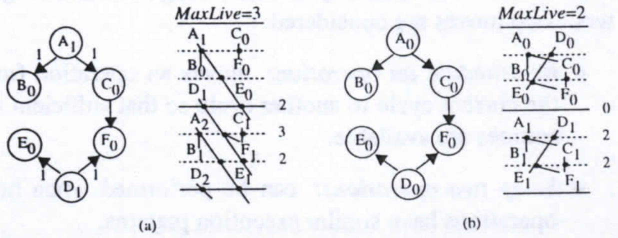

**Figure 5. Example of SPAN reduction (a) DG example before SPAN reduction and scheduling with MaxLive=3 (b) DG after SPAN reduction and scheduling with MaxLive=2**

### 5.2 SPAN reduction

The *SPAN* of a DG is defined as $\lambda_{max} - \lambda_{min} + 1$, where $\lambda_{max}$ and $\lambda_{min}$ are the maximum and minimum values for $\lambda$ respectively. In general, a reduction of the *SPAN* leads to a reduction in the iteration time, the variable lifetimes, the number of registers required to store partial results across iterations and the size of the prologue and the epilogue.

The algorithm to reduce the *SPAN* calculates the maximum value for $\lambda$ ($\lambda_{max}$) by exploring all nodes in the DG. Following this, $\lambda_{max}$ is iteratively decreased until a DG with minimum *SPAN* is found (minimum *SPAN*=*UD*) or the critical path of the current DG is longer than the expected initiation interval.

A transformation called *reduce_index* is used to reduce $\lambda_{max}$. *Reduce_index(u)* is based on *dependence retiming*, and it is only used in nodes so that the transformed DG has non-negative dependences. In order to reduce the *SPAN*, a node is selected among those whose index is $\lambda_{max}$. The node which produces the DG with the shortest critical path is selected at each iteration of the loop.

```
function reduce_index($u$);
    $\lambda'(u) := \lambda(u) - 1$;
    for each $e = (u, v) \in E$ do $\delta'(e) := \delta(e) - 1$;
    for each $e = (v, u) \in E$ do $\delta'(e) := \delta(e) + 1$;
endfunction;
```

After each index reduction, *retiming_and_scheduling* is executed, attempting to reduce the number of scheduling dependences in the DG. This is done without increasing the *SPAN*. Figure 5 shows an example of *SPAN* reduction.

### 5.3 Incremental scheduling

*Incremental scheduling* reduces variable lifetimes by rescheduling some operations within the schedule without changing their iteration index. Operations to be moved are selected among those that produce (or consume) a variable whose lifetime crosses a cycle consuming $MaxLive$ registers. Two moves are considered:

- *Re-schedule an operation:* moves an operation from the current cycle to another cycle so that sufficient resources are available.
- *Swap two operations:* can be performed when both operations have similar execution patterns.

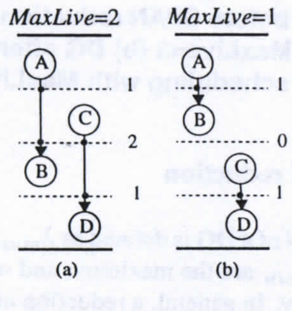

**Figure 6. Incremental scheduling**

Figure 6 shows an example of how *incremental scheduling* may reduce $MaxLive$. The execution time of the register optimization algorithm is $O(V^3E + VE^2)$ [20]. Despite the timing complexity seems rather high, results show that little CPU-time is required to find the final schedule.

## 6 Comparison with other approaches

We have borrowed from [8] a set of 24 benchmark loops selected from assorted scientific programs such as Livermore Loops, SPEC, Linpack and Whetstone. As in [8], we assume a result latency of 1 cycle for add, subtract, store, and move instructions, 2 cycles for multiply and load, and 17 cycles for divide. We also assume that all the functional units are fully pipelined. Although this assumption is not realistic for dividers, we follow it here for the sake of future comparisons with other approaches.

The results obtained by *UNRET* are compared with the results obtained by Huff's slack scheduling (SLACK) in [9], Wang and Eisenbeis' FRLC [24], Govindarajan et al.'s SPILP [8] and LLosa et al.'s HRMS [15]. All the methods use heuristics to find the schedule except SPILP, which uses an ILP formulation.

Table 1 shows the efficacy of *UNRET* in finding an optimal schedule. In order to make the comparisons, we will assume an architecture with 1 FP adder, 1 FP multiplier, 1 FP divisor and 1 load/store unit. The results obtained by SLACK, FRLC and SPILP have been previously reported in [8] by using a Sparc-10/40 workstation. We have used the same type of machine to measure the time required by *UNRET*. For each benchmark, Table 1 shows the *MII*,

| Application Program | | MII | SLACK | | FRLC | | SPILP | | UNRET | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | II | T | II | T | II | T | II | T |
| SPEC SPICE | L1 | 1 | 1 | 0.01 | 2 | 0.02 | 1 | 0.82 | 1 | 0.06 |
| | L2 | 6 | 7 | 0.03 | 6 | 0.03 | 6 | 12.4 | 6 | 0.08 |
| | L3 | 6 | 6 | 0.02 | 6 | 0.02 | 6 | 0.72 | 6 | 0.08 |
| | L4 | 10 | 12 | 0.10 | 12 | 0.03 | 11 | 3.60 | 11 | 0.96 |
| | L5 | 2 | 2 | 0.02 | 2 | 0.02 | 2 | 0.70 | 2 | 0.08 |
| | L6 | 2 | 3 | 0.03 | 17 | 0.03 | 2 | 7.67 | 2 | 0.21 |
| | L7 | 3 | 3 | 0.03 | 17 | 0.01 | 3 | 0.70 | 3 | 0.10 |
| | L8 | 3 | 5 | 0.03 | 3 | 0.02 | 3 | 3.15 | 3 | 0.08 |
| | L10 | 3 | 3 | 0.02 | 3 | 0.02 | 3 | 1.88 | 3 | 0.08 |
| SPEC DODUC | L1-f | 20 | 20 | 0.03 | 20 | 0.03 | 20 | 4.35 | 20 | 0.20 |
| | L3 | 20 | 20 | 0.03 | 20 | 0.03 | 20 | 1.03 | 20 | 0.20 |
| | L7 | 2 | 2 | 0.01 | 18 | 0.03 | 2 | 0.70 | 2 | 0.11 |
| SPEC-FP. | L1 | 20 | 20 | 0.03 | 20 | 0.02 | 20 | 0.93 | 20 | 0.11 |
| Livermore | L1 | 3 | 5 | 0.05 | 4 | 0.02 | 3 | 1.97 | 3 | 0.16 |
| | L5 | 3 | 3 | 0.05 | 3 | 0.02 | 3 | 0.73 | 3 | 0.06 |
| | L23 | 9 | 9 | 0.13 | 9 | 0.12 | 9 | 233 | 9 | 0.46 |
| Linpack | L1 | 2 | 2 | 0.02 | 3 | 0.02 | 2 | 2.62 | 2 | 0.06 |
| Whetstone | L1 | 17 | 18 | 0.17 | 18 | 0.08 | 17 | 4.25 | 17 | 0.31 |
| | L2 | 6 | 7 | 0.03 | 17 | 0.03 | 6 | 2.05 | 6 | 0.13 |
| | L3 | 5 | 5 | 0.03 | 5 | 0.02 | 5 | 0.73 | 5 | 0.10 |
| | C1 | 4 | 4 | 0.02 | 4 | 0.02 | 4 | 0.75 | 4 | 0.05 |
| | C2 | 4 | 4 | 0.02 | 4 | 0.02 | 4 | 1.87 | 4 | 0.08 |
| | C4 | 4 | 4 | 0.01 | 4 | 0.03 | 4 | 1.85 | 4 | 0.08 |
| | C8 | 4 | 4 | 0.02 | 4 | 0.02 | 4 | 1.77 | 4 | 0.08 |

**Table 1. Results obtained by different approaches by using an architecture with 1 FU of each type**

the initiation interval (*II*) of the schedule found by each approach and the time required to find the schedule (T). The large initiation interval obtained by FRLC for examples SPEC-SPICE 6 and 7, and for SPEC-DODUC 7, is because such an algorithm has been programmed without considering overlapping among the schedules of successive (pipelined) iterations. As SPILP, *UNRET* obtains the *MII* for all cases except for SPEC-SPICE loop 4, but in less time. Since SPILP is an ILP approach, it obtains optimal results. Therefore, we conclude that no schedule exists in the calculated *MII*. Note that no unrolling has been necessary to obtain optimal schedules in all cases.

In order to show the improvement in throughput achieved by unrolling the loop, Table 2 shows the results obtained by using an architecture with of 3 FP-adders, 2 FP-multipliers, 1 FP-divisor and 2 load/store units. Results are compared to HRMS [15], an efficient modulo scheduling algorithm focused to reduce the register pressure. First columns show the *II* and the number of registers (Reg) required for each benchmark on each approach. Last columns show the pair which produces the schedule ($II_K$, K), the difference between the number of registers required by *UNRET* and HRMS (diff reg), and the throughput improvement achieved by unrolling the loop (speed up). Table 2 shows that *UNRET* obtains the same throughput as HRMS for the same *UD*. However, the utilization of the optimal *UD* improves the throughput achieved by HRMS in 21% of all cases (see examples SPEC-SPICE 7 and 8, Livermore 1, and Whetstone C4 and C8). This is because, in general, systems which do not perform unrolling try to find a schedule in $\lceil MII \rceil$ cycles when *MII* is not an integer. Note that optimal schedules are found

| Application Program | | HRMS | | UNRET | | $II_K/K$ | diff reg | speed up |
|---|---|---|---|---|---|---|---|---|
| | | II | Reg | II | Reg | | | |
| SPEC-SPICE | L1 | 1 | 5 | 1 | 5 | | | |
| | L2 | 2 | 14 | 2 | 14 | | | |
| | L3 | 6 | 3 | 6 | 3 | | | |
| | L4 | 1 | 11 | 1 | 11 | | | |
| | L5 | 2 | 2 | 2 | 2 | | | |
| | L6 | 2 | 34 | 2 | 34 | | | |
| | L7 | 2 | 39 | 1.5 | 53 | 3/2 | +14 | 4/3 |
| | L8 | 2 | 6 | 1.5 | 9 | 3/2 | +3 | 4/3 |
| | L10 | 3 | 3 | 3 | 3 | | | |
| SPEC-DODUC | L1-f | 2 | 1 | 2 | 1 | | | |
| | L3 | 2 | 7 | 2 | 8 | | +1 | |
| | L7 | 2 | 28 | 2 | 28 | | | |
| SPEC-FPPPP | L1 | 2 | 4 | 2 | 4 | | | |
| Livermore | L1 | 2 | 14 | 1.5 | 2 | 3/2 | +6 | 4/3 |
| | L5 | 3 | 5 | 3 | 5 | | | |
| | L23 | 8 | 17 | 8 | 14 | | -3 | |
| Linpack | L1 | 1 | 13 | 1 | 13 | | | |
| Whetstone | L1 | 17 | 7 | 17 | 8 | | +1 | |
| | L2 | 6 | 9 | 6 | 9 | | | |
| | L3 | 5 | 5 | 5 | 5 | | | |
| | C1 | 4 | 2 | 4 | 2 | | | |
| | C2 | 2 | 4 | 2 | 4 | | | |
| | C4 | 2 | 6 | 1.33 | 7 | 4/3 | +1 | 3/2 |
| | C8 | 2 | 1 | 1.33 | 11 | 4/3 | +1 | 3/2 |

**Table 2. Comparison of throughput and register requirements for VLIW processors by using 3 FP adders, 2 FP multipliers, 1 FP divisor and 2 load/store units**

| Application Program | | MS | | after SR | after IS | time SR | time IS | diff |
|---|---|---|---|---|---|---|---|---|
| | | II | Reg | | | | | |
| SPEC-SPICE | L1 | 1 | 5 | 5 | 5 | 0.06 | 0.01 | |
| | L2 | 2 | 15 | 15 | 14 | 0.10 | 0.01 | -1 |
| | L3 | 6 | 3 | 3 | 3 | 0.10 | 0.01 | |
| | L4 | 10 | 11 | 11 | 11 | 0.33 | 0.01 | |
| | L5 | 2 | 2 | 2 | 2 | 0.06 | 0.01 | |
| | L6 | 2 | 35 | 34 | 34 | 0.08 | 0.01 | -1 |
| | L7 | 2 | 41 | 41 | 41 | 0.08 | 0.01 | |
| | L8 | 2 | 6 | 6 | 6 | 0.06 | 0.01 | |
| | L10 | 3 | 3 | 3 | 3 | 0.10 | 0.01 | |
| SPEC-DODUC | L1-f | 20 | 10 | 10 | 10 | 0.18 | 0.01 | |
| | L3 | 21 | 8 | 8 | 7 | 0.13 | 0.01 | -1 |
| | L7 | 2 | 28 | 28 | 28 | 0.08 | 0.01 | |
| SPEC-FPPPP | L1 | 20 | 4 | 4 | 4 | 0.13 | 0.01 | |
| Livermore | L1 | 2 | 16 | 16 | 15 | 0.11 | 0.01 | -1 |
| | L5 | 3 | 5 | 5 | 5 | 0.08 | 0.01 | |
| | L23 | 8 | 18 | 15 | 14 | 0.33 | 0.01 | -4 |
| Linpack | L1 | 1 | 13 | 13 | 13 | 0.08 | 0.01 | |
| Whetstone | L1 | 17 | 8 | 8 | 8 | 0.26 | 0.01 | |
| | L2 | 6 | 9 | 9 | 9 | 0.15 | 0.01 | |
| | L3 | 5 | 5 | 5 | 5 | 0.13 | 0.01 | |
| | C1 | 4 | 2 | 2 | 2 | 0.11 | 0.01 | |
| | C2 | 2 | 4 | 4 | 4 | 0.10 | 0.01 | |
| | C4 | 2 | 6 | 6 | 6 | 0.06 | 0.01 | |
| | C8 | 2 | 10 | 10 | 10 | 0.06 | 0.01 | |

**Table 3. Register reduction in a modulo scheduling algorithm by assuming a VLIW processor with 3 FP adders, 2 FP multipliers, 1 FP divisor and 2 load/store units**

in all cases. Therefore, Farey's series do not require to be explored for these loops with the given architecture. Table 2 also shows that increasing the *UD* also increases (in general) the number of registers required by the schedule.

HRMS is a good algorithm (from the point of view of register pressure) that performs software pipelining and register reduction at a time. Since HRMS and *UNRET* obtain similar results, we conclude that separating both tasks (software pipelining and register reduction) may produce very promising results (note that the register reduction phase of *UNRET* consumes little CPU-time). In order to demonstrate this claim, we have executed the register reduction algorithm over the schedules generated by a simple modulo scheduling. The heuristics used by the modulo scheduling to select which instruction must be scheduled are based on the topology of the DG (top-down). Table 3 shows the results obtained. For each benchmark, the first two columns show the initiation interval and the register requirements of the schedule found by the modulo scheduling (MS). The next columns show the registers used by the schedule after each step of the register reduction algorithm: *SPAN* reduction (SR) and incremental scheduling (IS), as well as the CPU-time used by each step. The final column (diff) shows the register reduction achieved.

Finally, in order to evaluate how far are the results obtained by *UNRET* from the optimal ones, we have used an integer linear programming approach [4] to calculate the optimal initiation interval and the minimal number of registers. Results in Table 4 show that *UNRET* obtains optimal results in the initiation interval for all cases, and optimal results in the number of required registers in almost all cases. First columns present the loop example. The optimal initiation interval and the minimal number of registers computed by [4] are next presented. The following columns show the II and the number of registers (Reg) used by *UNRET*. Finally, last column indicates those examples for which *UNRET* requires more registers than [4].

## 7 Conclusions

This paper presents *UNRET*, a new algorithm for resource-constrained software pipelining. *UNRET* works with multiple-cycle (possibly pipelined) functional units. An operation may use different types of functional units while executes. *UNRET* is based on the exploration of the throughput achievable by a schedule as a function of the unrolling degree. For a target throughput, we analytically compute both the number of times the loop must be unrolled and the expected initiation interval of the schedule, exploring the solution space in two dimensions. In order to perform software pipelining, *dependence retiming*, a loop transformation at graph level, is proposed. The number of required registers is reduced after a schedule is found.

We have shown the effectiveness of *UNRET* by using well-known benchmarks. We have also shown, by means of an example, how *UNRET* may improve the results obtained by other techniques which explore the solution space in only one dimension.

| Application Program | | OPTIMAL | | UNRET | | diff |
|---|---|---|---|---|---|---|
| | | II | Reg | II | Reg | reg |
| SPEC-SPICE | L1 | 1 | 3 | 1 | 3 | |
| | L2 | 6 | 5 | 6 | 5 | |
| | L3 | 6 | 2 | 6 | 2 | |
| | L4 | 11 | 8 | 11 | 8 | |
| | L5 | 2 | 1 | 2 | 1 | |
| | L6 | 2 | 15 | 2 | 15 | |
| | L7 | 3 | 15 | 3 | 15 | |
| | L8 | 3 | 5 | 3 | 5 | |
| | L10 | 3 | 2 | 3 | 2 | |
| SPEC-DODUC | L1-f | 2 | 5 | 2 | 6 | +1 |
| | L3 | 2 | 3 | 2 | 4 | +1 |
| | L7 | 2 | 18 | 2 | 18 | |
| SPEC-FPPPP | L1 | 2 | 2 | 2 | 2 | |
| Livermore | L1 | 3 | 6 | 3 | 7 | +1 |
| | L5 | 3 | 3 | 3 | 3 | |
| | L23 | 9 | 1 | 9 | 1 | |
| Linpack | L1 | 2 | 5 | 2 | 5 | |
| Whetstone | L1 | 17 | 5 | 17 | 5 | |
| | L2 | 6 | 6 | 6 | 6 | |
| | L3 | 5 | 4 | 5 | 4 | |
| | C1 | 4 | 1 | 4 | 1 | |
| | C2 | 4 | 2 | 4 | 2 | |
| | C4 | 4 | 4 | 4 | 4 | |
| | C8 | 4 | 8 | 4 | 8 | |

**Table 4. Comparison with optimal results for superscalar processors (1 FU of each type)**

## Acknowledgments

We would like to thank Q. Ning, R. Govindarajan, Eric R. Altman and Guang G. Gao for supplying the data dependence graphs used for comparisons, R. M. Badia for supplying the optimal results in Table 4 and J. LLosa for supplying the schedules found by HRMS and the top-down modulo scheduling.

## References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proc. of the 10th ACM Annual Symp. on Principles of Programming Languages*, pages 177–189, Jan. 1983.

[2] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proc. of the 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 122–131, Apr. 1991.

[3] F. Catthoor, W. Geurts, and H. De Man. Loop transformation methodology for fixed-rate video, image and telecom processing applications. In *Proc. of the Int. Conf. on Application-Specific Array Processors (ASAP94)*, 1994.

[4] J. Cortadella, R. M. Badia, and F. Sánchez. A mathematical formulation of the loop pipelining problem. Technical Report UPC-DAC-1995-36, Department of Computer Architecture (UPC), Oct. 1995.

[5] E. Girczyc. Loop winding: A data flow approach to functional pipelining. In *Proc. of the Int. Symp. Circuits and Systems (ISCS)*, pages 382–385, May 1987.

[6] E. Goffman Jr. *Computer and Job Scheduling Theory*. John Wiley and Sons, New York, 1976.

[7] G. Goossens, J. Vandewalle, and H. De Man. Loop optimization in register-transfer scheduling for DSP systems. In *Proc. of the 26th Design Automation Conf. (DAC)*, pages 826–831, 1989.

[8] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proc. of the 27th Annual Int. Symp. on Microarchitecture (MICRO27)*, pages 85–94, Nov. 1994.

[9] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the 6th Conf. Programming Languages Design and Implementation*, pages 258–267, 1993.

[10] C.-T. Hwang, Y.-C. Hsu, and Y.-L. Lin. Scheduling for functional pipelining and loop winding. In *Proc. of the 28th Design Automation Conf. (DAC)*, pages 764–769, June 1991.

[11] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Trans. on Computer-Aided Design*, 10(4):464–475, Apr. 1991.

[12] R. Jones and V. Allan. Software pipelining: A comparison and improvement. In *Proc. of the 23rd Annual Workshop on Microprogramming and Microarchitecture (MICRO23)*, pages 46–56, Nov. 1990.

[13] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of the ACM SIGPLAN88 Conf. on Programming Languages Design and Implementation*, pages 318–328, June 1988.

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

[15] J. LLosa, M. Valero, E. Ayguadé, and A. González. Hypernode reduction modulo scheduling. In *Proc. of the 28th Annual Int. Symp. on Microarchitecture (MICRO28)*, pages 350–360, Nov. 1995.

[16] P. Paulin and J. Knight. Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Trans. on Computer-Aided Design*, 8(6):661–679, June 1989.

[17] B. Rau and C. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. of the 14th Annual Workshop on Microprogramming*, pages 183–198, Oct. 1981.

[18] B. Rau, M. Schlansker, and P. Tirumalai. Code generation schema for modulo scheduled loops. In *Proc. of the 25th Annual Int. Symp. on Microarchitecture (MICRO25)*, pages 158–169, Dec. 1992.

[19] M. Rim. *High-Level Synthesis of VLSI Designs for Scientific Programs*. PhD thesis, University of Wisconsin-Madison, 1993.

[20] F. Sánchez. *Loop Pipelining with Resource and Timing Constraints*. PhD thesis, Universitat Politècnica de Catalunya (Spain), 1995.

[21] F. Sánchez and J. Cortadella. Resource-constrained software pipelining for high-level synthesis of DSP systems. In M. Moonen and F. Catthoor, editors, *Algorithms and Parallel VLSI Architectures III*, pages 377–388. Elsevier, 1995.

[22] F. Sánchez and J. Cortadella. Time-constrained loop pipelining. In *Proc. of the Int. Conf. Computer-Aided Design (ICCAD)*, pages 592–596, Nov. 1995.

[23] M. Schroeder. *Number Theory in Science and Communication*. Springer-Verlag, 1990.

[24] J. Wang and C. Eisenbeis. Decomposed software pipelining: A new approach to exploit instruction level parallelism for loops programs. In *Proc. of the IFIP WG 10.3, Working Conf. on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, January 1993.