Microprocessing and Microprogramming 38 (1993) 429-436
North-Holland

Resource-Constrained Pipelining Based on Loop Transformations

F. Sánchez and J. Cortadella[a*]

[a]Polytechnic University of Catalonia, Dep. of Computer Architecture,
Campus Nord, Mòdul D6, Gran Capità s/n, Barcelona-E08071, Spain
E-mail: fermin@ac.upc.es and jordic@ac.upc.es

In this paper a novel technique for resource-constrained loop pipelining is presented. *RCLP* is based on several dependence graph operations: loop unrolling, operation retiming, resource-constrained scheduling, and span reduction. All these operations are focused to find a minimum length schedule able to be executed with a limited number of resources and thus maximizing resource utilization. The results obtained show that *RCLP* is superior to the existing software-pipelining-based approaches. This technique has also been evaluated with 300 randomly generated loop depedence graphs. In all cases a time-optimal schedule has been found.

1. Introduction

In order to exploit the parallelism inside the loops, different *software pipelining* techniques [1] have been developed. Techniques to achieve software pipelining fall into three broad classes: *loop folding*[2] techniques, *loop winding*[3] techniques and *loop unrolling*[4] techniques.

Loop winding techniques optimize loops by introducing partial overlaps between the execution times of successive loop iterations, attempting to find an execution window. This window usually contains one loop body, with operations belonging to different iterations of the loop. *Functional pipelining* techniques [5] are a special case of *loop winding* techniques. These techniques allow a new iteration i to start before the execution of the iteration $i-1$ have been completed

Loop folding techniques are based on *retiming* loop operations [6] in order to find an optimal schedule containing operations belonging to different iterations. *Modulo Scheduling* techniques [7] are a particular case of *loop folding* techniques. They use the resource and recurrence constraints to determine a tight lower bound on Initiation Interval and delay operations in order to resolve resource conflicts.

Loop unrolling techniques are based on unrolling the loop and compacting operations until an execution pattern is found. In general, the loop is unrolled and compacted until a repeated pattern is found.

*This work was supported by CYCYT TIC-91-1036

In this paper we present *RCLP*, a new technique based on loop unrolling and retiming. It can be considered a *loop folding* technique, in which the new loop body is composed of several loop iterations. *RCLP* exploits parallelism inside and across loop iterations under resource constraints, attempting to find a schedule with maximum resources efficiency and minimum span. Although resource-constrained scheduling is an NP-hard problem [8], the results obtained show that a time-optimal solution can be found in most cases.

The paper is organized as follows. Section 2 contains the basic definitions used in the paper. Section 3 describes loop transformations and the loop pipelining algorithm without resource constraints. Section 4 describes the resource-constrained loop pipelining algorithm. Section 5 presents the results and compares them with the results obtained by other methods. Finally, section 6 concludes the paper.

2. Basic Definitions

2.1. Representation of a loop

Our scope will be limited to single nested DO-like loops whose body is a basic block, i.e. neither containing other conditional nor loop statements in the loop body. We will also assume any functional unit (resource) can execute any operation in one cycle.

In DO-like loops, the number of executed iterations is known before the execution of the loop. If the loop executes K times, *iteration* i will denote the i-th execution of the loop body. It will be

assumed, without loss of generality, that i ranges between 0 and $K-1$. If u is an operation of the loop, u_i will denote the execution of operation u at the i-th iteration. Data dependences between operations of a loop fall into two categories: *Local dependences* between operations from the same iteration (u_i and v_i) and *Global* or *Loop-carried dependences* between operations from different iterations (u_i and v_j, $i < j$).

A loop will be represented by a triplet called π-*graph*, $\pi =< G, \lambda, \delta >$, where $G(V, E)$, *the dependence graph*, is a directed graph and λ (*index*) and δ (*distance*) are two mappings $\lambda : V \mapsto \mathcal{N}$ and $\delta : E \mapsto \mathcal{N}$. Each vertex $v \in V$ represents an operation of the loop body. The number of operations of the loop will be denoted by I_π. Each edge $e = (u, v) \in E$ represents a depedence between operations u and v.

In general, an edge (u, v) represents a dependence between operations $u_{i+\lambda(u)}$ and $v_{i+\lambda(v)+\delta(u,v)}$, where $i \in \mathcal{N}$. Grafically, it will be depicted as $u_{\lambda(u)} \xrightarrow{\delta(u,v)} v_{\lambda(v)}$.

We denote as *initial* π-*graph* a π-*graph* $\pi =< G, \lambda, \delta >$ in which $\forall u \in V, \lambda(u) = 0$.

Definition 2.1 : Equivalent π-graphs

$\pi =< G, \lambda, \delta >$ *and* $\pi' =< G, \lambda', \delta' >$, *are two* equivalent π-*graphs (represent the same loop) if* $\forall (u, v) \in E$

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

Definition 2.1 states that a dependence can be represented by different mappings, λ and δ, as far as equation 1 is fulfilled. This is the key feature used in our proposal, which searches for the appropriate mappings λ and δ that allow schedule operations to achieve a maximum pipelining degree.

Definition 2.2 : G_π (Scheduling Graph)

The scheduling graph *of a* π-graph *is a graph,* $G_\pi(V, E_\pi)$, *in which* $E_\pi = \{e \in E \mid \delta(e) = 0\}$.

G_π contains all *local dependences* of π. Given a π-graph $\pi =< G, \lambda, \delta >$, we will define a *local path* p in G as a path in G_π. The length of a path p is the number of vertices traversed by p, and can be calculated as $l(p) = |p| + 1$.

We will denote as a *maximal local path (MLP)* a local path not included in any other local path. The longest of the MLPs is called *critical path*. If a local path is maximal then there is a vertex u such that for all $(x, u) \in E, \delta(x, u) > 0$. Otherwise the path would not be maximal. Vertex u will be called the *head vertex* of the path. Likewise, the vertex v such that for all $(v, x) \in E, \delta(v, x) > 0$ is called the *tail vertex* of the path.

Since dependences induce a partial ordering on the execution of operations, a loop *schedule* is an assignment from the loop body operations to execution cycles so that dependences are fulfilled.

Definition 2.3 : $S(\pi)$ (Schedule of π)

A schedule *of a* π-*graph* π *is a mapping* $S : V \mapsto \mathcal{N}$ *that fulfils* $\forall (u, v) \in E_\pi, S(u) < S(v)$.

Definition 2.4 : $\Pi_{S(\pi)}$ (Initiation interval of a Schedule)

The initiation interval *of a schedule* $S(\pi)$ *is*

$$\Pi_{S(\pi)} = \max_{u \in V} S(u) - \min_{u \in V} S(u) + 1$$

Π denotes the interval between the initiation of two consecutive iterations in the schedule. Π also denotes the number of cycles of the schedule.

Definition 2.5 : T_π (Execution Time of π)

The execution time *of a* π-graph, T_π, *is the length of the* critical path *in its scheduling graph.*

Given a π-graph, T_π defines the minimum number of cycles required to execute the loop body represented by π. T_π also indicates the minimum number of cycles between the initiation of consecutive iterations for any schedule of π. Therefore, for any schedule $S(\pi)$, $\Pi_{S(\pi)} \geq T_\pi$.

2.2. Minimum initiation interval and Parallelism of a loop

A loop can be represented by different and equivalent π-graphs. While T_π is a property of each π-graph, the *minimum initiation interval* and the *parallelism* are properties of the loop itself, and not of its representation. For this reason, the terms *minimum initiation interval of a loop* and *minimum initiation interval of a* π-*graph* will be indistinctly used, and denoted by MII_π, as

well as *parallelism of a π-graph* and *parallelism of a loop*, denoted by $\mathcal{P}_\pi$. The *parallelism of a π-graph* is the average number of operations executable in a cycle when the loop is executed with its minimum initiation interval.

We will show the *minimum initiation interval* can be computed by considering only the *initial π-graph* of the loop.

A loop with no recurrences (also called *vector* loop) has $MII_\pi = 0$. Finding a time-optimal schedule for a vector loop is an easy task [9], and has no interest to the scope of this paper. Recurrences delimit the degree of parallelism achievable for a loop (loop with recurrences are called *recurrent* loops). Our interest is focused on pipelining of recurrent loops.

Let us call R the set of edges that form a recurrence and $|R|$ the number of elements in R. Let us define δ_R as

$$\delta_R = \sum_{(u,v)\in R} \delta(u,v)$$

For any operation u such that $(u,v) \in R$, u_i must be scheduled at least $|R|$ cycles before $u_{i+\delta_R}$.

Let us call L_R (*latency of a recurrence*) the minimum number of cycles between the execution of u_i and u_{i+1} (initiation of consecutive iterations) imposed by recurrence R. Then $L_R = \frac{|R|}{\delta_R}$

Figure 1(a) depicts a loop with one recurrence R and $L_R = 2$, thus indicating that an iteration can be initiated every two cycles. An equivalent π-graph and its corresponding schedule are shown in Figures 1(b) and 1(c). In general it will be neccessary to include a prolog and an epilog for the correct execution of the loop[9].

Definition 2.6 : MII_π (Minimum initiation interval of π)

The minimum initiation interval *of π is*

$$MII_\pi = \max_{R\subseteq E} L_R$$

Theorem 2.1 *If π and π' are equivalent π-graphs, then $MII_\pi = MII_{\pi'}$. Proof: see [9].*

Theorem 2.2 $T_\pi \geq MII_\pi$. *Proof: see [9].*

MII_π can be found in polynomial time by using *Karp's algorithm* [10] to calculate the weight of the *minimum mean-weight cycle* in a graph.

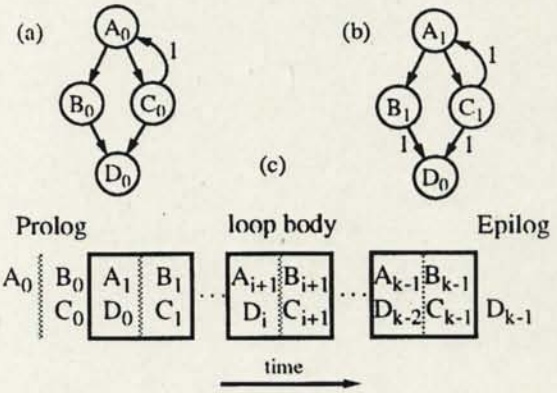


Figure 1. Scheduling of an R-loop

Definition 2.7 : $\mathcal{P}_\pi$ (Parallelism of π)

The parallelism *of a π-graph is* $\mathcal{P}_\pi = \frac{I_\pi}{MII_\pi}$

Definition 2.8 : $P_{S(\pi)}$(Parallelism of $S(\pi)$)

The Parallelism of a Schedule *is the average number of instructions per cycle issued by the schedule:* $P_{S(\pi)} = \frac{I_\pi}{II_{S(\pi)}}$

Definition 2.9 : SPAN(π) (Span of π)

The span *of a π-graph is defined as*

$$\text{SPAN}(\pi) = \max_{u\in V} \lambda(u) - \min_{u\in V} \lambda(u) + 1$$

3. Loop pipelining without resource constraints

3.1. π-graph Transformations

3.1.1. π-graph unrolling

We are interested in finding a *π-graph* in which $T_\pi = MII_\pi$. This is only possible when MII_π is an integer, since T_π denotes a *number of cycles*. In general, a π-graph representing multiple instances of the loop body will be required to force MII_π to be an integer value. It will be called a *multiple-instanced π-graph (M-graph)*.

An *M-graph* can be obtained by *unrolling* m times the original π-graph. The *M-graph* is another *initial π-graph* π^m in which each operation and each dependence have been replicated m times.

For each vertex A representing operations A_i in π, π^m has m vertices $(A^0, A^1, \ldots, A^{m-1})$, so that each A^j represents all the operations A_i such

that $i \bmod m = j$. Likewise, for each dependence $A \xrightarrow{d} B$ in π, there will be m dependences $A^j \xrightarrow{\lfloor d/m \rfloor} B^{(j+d) \bmod m}$ in π^m. Figure 2 shows an example of π-graph unrolling.

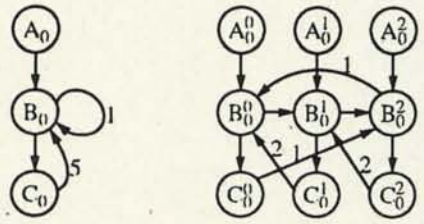


Figure 2. π-graph unrolling

π and π^m represent the same loop. However the body of the loop represented by π^m will be executed only $\frac{k}{mt}$ times (minus some constant that depends on the *prolog* and *epilog*).

Theorem 3.1 $MII_{\pi^m} = mMII_\pi$. *Proof:See [9].*

An M-graph with integer $T_{\pi^m} = MII_{\pi^m}$ can be obtained as follows:

1. Let π be the original π-graph, with $MII_\pi = \frac{a}{b}$
2. Calculate $m = \frac{b}{gcd(a,b)}$ (to generate the smallest M-graph with integer *MII*)
3. Generate $\pi^m = unroll\ (\pi, m)$. The minimum initiation interval will be $MII_{\pi^m} = \frac{a}{gcd(a,b)}$

3.1.2. Shifting local dependences

This section proposes *shift_local*, a transformation to decrease the execution time T_π of a π-graph until MII_π.

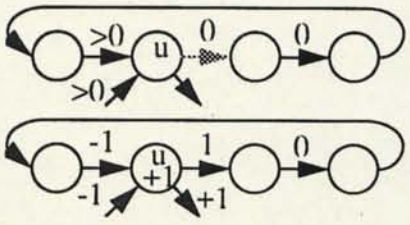


Figure 3. Shift Local Transformation

The function *Shift_local* transforms a local dependence $e = (u, v)$ into a global dependence by performing the following steps:

- $\lambda'(u) := \lambda(u) + 1$
- $\forall (u, x) \in E,\ \delta'(u, x) := \delta(u, x) + 1$
- $\forall (x, u) \in E,\ \delta'(x, u) := \delta(x, u) - 1$

However, *shift_local* may create new local dependences[2]. Figure 3 illustrates how the transformation operates. *Shift_Local* always generates an equivalent π-graph [9].

3.2. Finding a π-graph with $T_\pi = MII_\pi$

In order to find a π-graph such that $T_\pi = MII_\pi$, the length of the critical path must be equal to MII_π (see definition 2.5). Thus, if $T_\pi > MII_\pi$ then π must be transformed to reduce T_π.

Given a π-graph π, the following algorithm finds a π-graph π' with $T_{\pi'} = MII_{\pi'}$:

function *find_min_π-graph* (π);
$m := \frac{b}{gcd(a,b)}$;
$\pi' := unroll(\pi, m)$;
while $\exists$ MLP p in π' with $l(p) > MII_\pi$ **do**
  $e := head\_edge(p)$;
  $\pi' := shift\_local\ (\pi', e)$
endwhile;
return(π');
end;

Find_min_π-graph always halts and can be executed in $O(E^3)$-time [9].

Figure 4 shows an example of finding a π-graph with minimum execution time by creating first an M-graph with integer *MII* and then reducing the critical path on the scheduling graph.

4. Resource-Constrained Loop Pipelining

4.1. Basic Definitions

In this section we propose a method that attempts optimal loop pipelining with resource constraints. We will denote by $S^N(\pi)$ and $\Pi_{S^N(\pi)}$ a schedule and the initiation interval of a schedule constrained to the use of N resources (RC-schedule).

[2] In order to guarantee the generation of a legal π-graph ($\forall e \in E, \lambda(e) \geq 0$) we always apply the transformation to a dependence $e = (u, v)$ such that u is the head vertex of an MLP. An equivalent transformation can be easily derived for the tail vertex of an MLP.

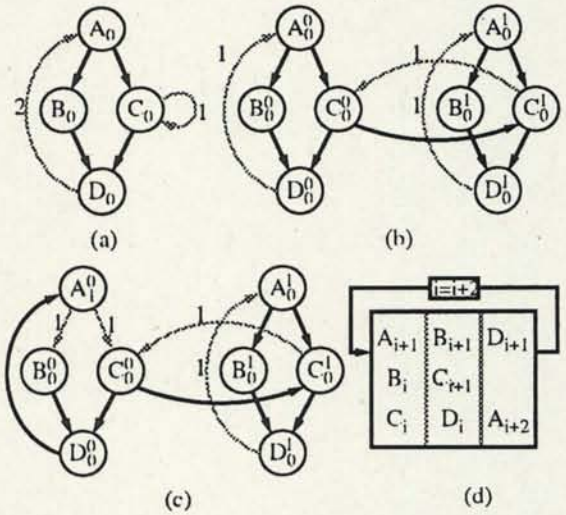


Figure 4. Finding a π-graph with minimum T_{π^m}. (a) Initial π-graph π ($MII_\pi = 3/2$). (b) M-graph π^2 ($MII_{\pi^2} = 3, T_{\pi^2} = 4$). (c) Transformed π-graph π'^2 ($T_{\pi'^2} = 3$). (d) Schedule.

Definition 4.1 $U_{S^N(\pi)}$**(Resource Utilization of** $S^N(\pi)$**)**

The Resource Utilization of an RC-schedule *is defined as* $U_{S^N(\pi)} = \frac{I_\pi}{N\Pi_{S^N(\pi)}}$; $U_{S^N(\pi)} \in (0, 1]$

$U_{S^N(\pi)} = 1$ denotes the case in which all the resources are fully used during the execution of the loop. In order to evaluate the merit of an RC-schedule we define the *Maximum Utilization of Resources*, which is an upper bound of the resource utilization of any RC-schedule for a given number of resources.

Definition 4.2 $U^N_{max}(\pi)$ **(Maximum Utilization of Resources)**

The Maximum Utilization of Resources *of any RC-schedule for a given* π*-graph is*

$$U^N_{max}(\pi) = \begin{cases} \frac{\mathcal{P}_\pi}{N} & \text{if } N > \mathcal{P}_\pi \\ 1 & \text{if } N \leq \mathcal{P}_\pi \end{cases}$$

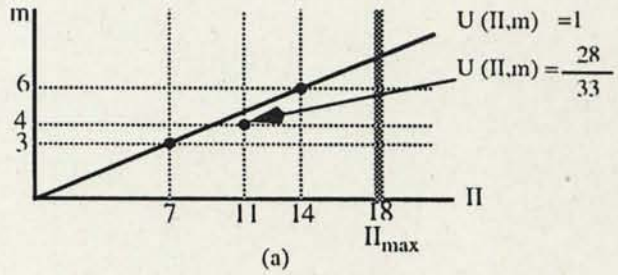
Definition 4.3 $P_{S^N(\pi)}$**(Parallelism of** $S^N(\pi)$**)**

The Parallelism of an RC-schedule *can be defined as* $P_{S^N(\pi)} = \frac{I_\pi}{\Pi_{S^N(\pi)}} = NU_{S^N(\pi)}$

4.2. Finding an RC-schedule with maximum Utilization of Resources

4.2.1. Graphic Representation of (Π, m)

The pairs (Π, m) can be represented in a diagram as shown in Figure 5(a). A pair (Π, m) represents the fact that m iterations are executed in Π cycles. Therefore, given a loop represented by π and a number of resources N, each pair (Π, m) represents a potential schedule in which resource utilization can be also denoted by $U(\Pi, m)$. Thus, $U(\Pi, m) = \frac{mI_\pi}{N\Pi}$



(a)

U(Π,m)	1	1	35/36	49/51	14/15	14/15	...
Π	7	14	12	17	5	10	...
m	3	6	5	7	2	4	...

(b)

Figure 5. (a) Graphic representation of pairs $U(\Pi, m)$, with $N = 3$ and $I_\pi = 7$. (b) Generation of pairs (Π, m) in decreasing order of $U(\Pi, m)$

Our goal is to find an RC-schedule with maximum resource utilization ($U_{S^N(\pi^m)} = U^N_{max}(\pi)$). *RCLP* explores pairs (Π, m) in decreasing order of $U(\Pi, m)$ until a schedule with $\Pi_{S^N(\pi^m)} = \Pi$ is found. Since the space (Π, m) is not finite it cannot be fully explored. For this reason, we restrict the maximum number of cycles (Π_{max}) of any schedule.

4.2.2. Generation of pairs $(\Pi$,m)

The resource utilization of any RC-schedule of π^m in Π cycles can be represented by the fraction $\frac{r}{y} = \frac{mI_\pi}{N\Pi}$. In order to generate all the possible values for (Π, m) in decreasing order of $U(\Pi, m)$,

we use the recurrent definition of Farey's [3] series [11] and the solutions of the resulting linear diophantine equation $xNII - ymI_\pi = 0$

Figure 5(b) shows an example of the generated pairs for the diagram of Figure 5(a).

4.2.3. π-graph Transformations and Resource-Constrained Scheduling

The algorithm to find an RC-schedule with maximum utilization of resources explores the pairs (II, m) until a schedule $S^N(\pi^m)$ such that $II_{S^N(\pi^m)} = II$ is found. For each pair (II, m) such that $U(II, m) \leq U^N_{max}(\pi)$, π^m is built and iteratively transformed by using *shift_local*. After each transformation the π-graph is scheduled in order to check whether $II_{S^N(\pi^m)} = II$ or not.

A function is defined to heuristically evaluate the goodness of a π-graph for scheduling, and provide a criterion to quit the search when a schedule with length II has not been found. The two main parameters considered by the function *better* are the length of the critical path and the number of local dependences in the graph. The algorithm to find a RC-schedule is next sketched:

```
function find_schedule(π,N,II,m):boolean:
π' := unroll(π, m); {builds π^m}
π'' := π';
Repeat
    sch:=scheduling(π'', N);
    if (II_S(π'',N) = II) return(true);
    selected:=select_edge(π'', e);
    {selects head edge or tail edge}
    if (selected) then
        mark_edge(π'', e);
        π'' := shift_local(π'', e);
        if better(π'', π') then
            π' := π'';
            unmark_edges(π'');
        endif
    endif
Until not(selected) ;
return (false);{schedule not found}
end
```

Among all the existing scheduling algorithms we have chosen *List Scheduling* [12] for its efficiency and low time complexity. However, the proposed approach does not preclude the use of other scheduling algorithms.

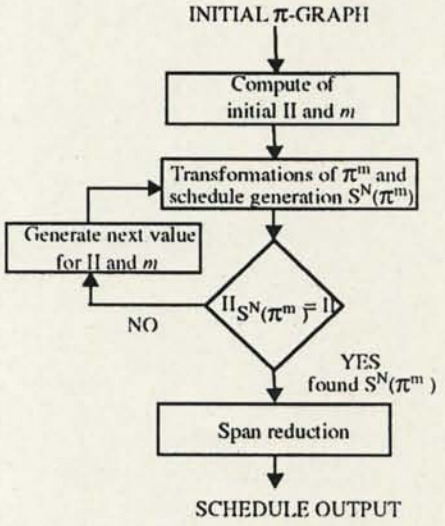


Figure 6. General view of *RCLP*

4.2.4. Span reduction

Once an RC-schedule has been found we try to reduce the span while mantaining $II_{S^N(\pi^m)} = II$. A reduction in the span results in:

- A reduction in the size of the prolog and the epilog.
- A reduction in the number of registers needed to store temporarily results across iterations.

The idea of the algorithm is as follows: First, the maximum value for $\lambda(u)$ is computed by exploring all nodes of the graph. Then, this value is iteratively decreased until a graph with minimum span is found (SPAN $= m$). The graph is transformed (by means of *shift_local*) and scheduled at each step by looking for a schedule of II cycles. The final schedule is the last found schedule in which $II_{S^N(\pi^m)} = II$.

Figure 6 shows in a flow diagram the steps followed by the method.

[3] The Farey series $F_{NII_{max}}$ contains in increasing order of magnitude all the fractions with denominator lower than or equal to NII_{max}

5. Results and comparison with other methods

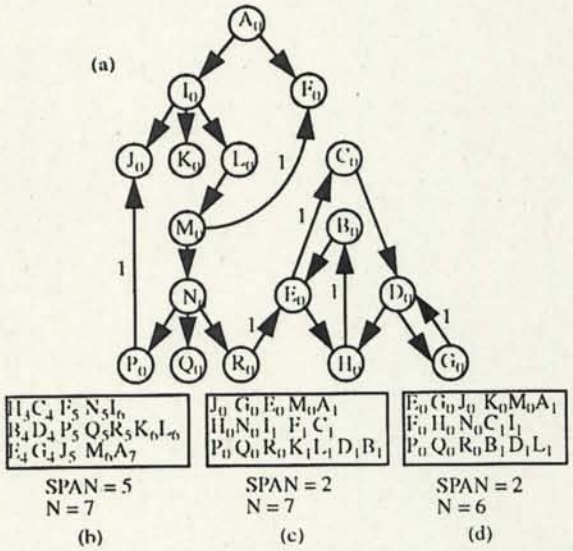


Figure 7. Comparison between different methods (a) An example of π-graph (b) Schedule obtained by using perfect pipelining (c) Schedule obtained by using the algorithm described in [3] (d) Schedule obtained by using *RCLP*

In order to compare the proposed technique with other methods we will use the example used by Cytron [13] and shown in Figure 7(a). Figures 7(b) and 7(c) depict the schedule obtained by Perfect Pipelining [14] and the technique proposed in [15] respectively.

None of these techniques has been devised for resource-constrained scheduling and, therefore, the minimum number of required resources is given. [14] obtains a schedule with maximum parallelism degree. [15] obtains a schedule with the same resource utilization and the same parallelism as [14] with smaller span. Both techniques use seven resources ($N = 7$). We obtain a schedule that requires only six resources ($N = 6$), while mantaining the same parallelism and *span* as [15], thus improving resource utilization.

In summary, *RCLP* optimizes *initiation interval*, resource utilization and *span* (thus reducing size of the prolog and the epilog).

We have tested our approach with 300 randomly generated loops, attempting to find RC-Schedules with a different number of resources. In all cases we have always obtained RC-Schedules with maximum resource utilization.

Figure 8 depicts some illustrative examples of these randomly generated loops and some RC-Schedules with different number of resources.

6. Conclusions and future work

We have presented a novel technique for resource-constrained loop pipelining. The method is based on transforming the dependence graph of the loop into another one where the loop has been unrolled and the operations have been retimed. The method optimizes initiation interval, span and resource utilization of the schedule.

Experiments done with a large number of randomly generated loops and benchmarks proposed in the literature show this technique is superior to the current existing ones and time-optimal schedules can be obtained in most cases.

As future work, extensions to multi-cyle operations and pipelined functional units are forseen.

Acknowledgments

We would like to thank Marc Noy for his helpful comments about Farey's series.

REFERENCES

1. Lam, M. "A Systolic Array Optimizing Compiler" *PhD thesis, Carnegie Mellon Univ*, 1987.
2. Goosens, G., Vandewalle, J. and De Man, H. "Loop Optimization in Register-Transfer Scheduling for DSP systems". *IEEE Design Aut. Conf.*, 1989, pp. 826-831.
3. Girczyc, E.F. "Loop Winding- A data flow approach to Functional Pipelining" *Proceedings of the IEEE ISCAS*, May 1987, pp. 382-385.
4. Aiken, A. and Nicolau, A. "Perfect Pipelining: A new Loop Parallelization Technique". *Proc of the 1988 European Symp. on Programming* March 1988. *Springer Verlag Lecture Notes in Computer Science*

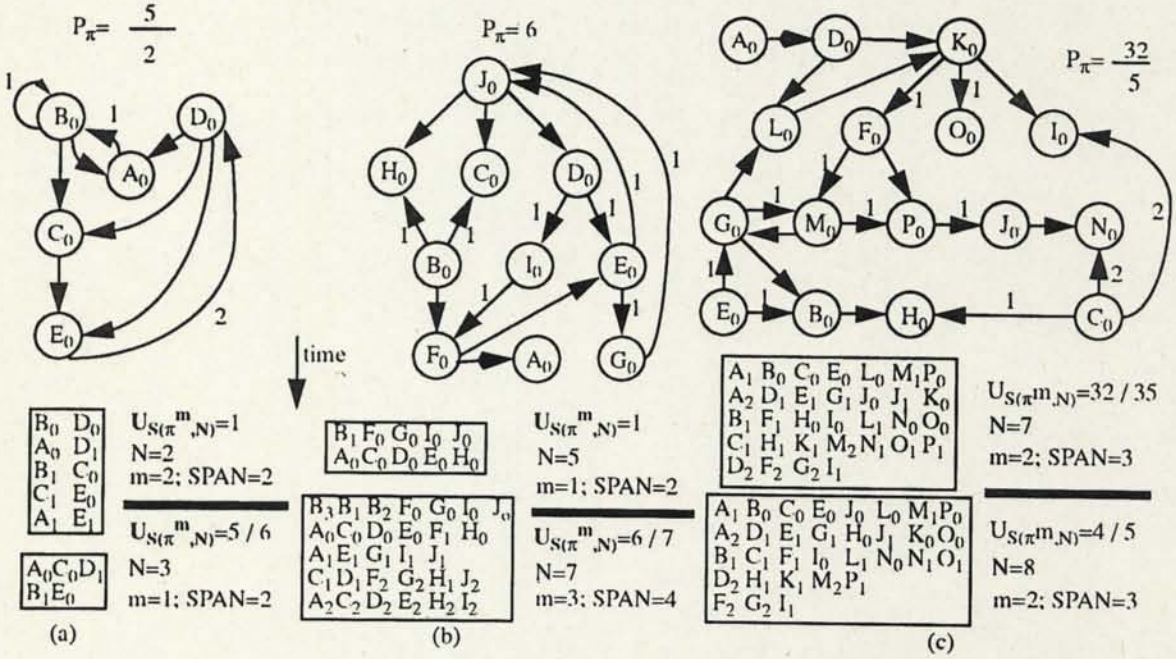


Figure 8. Three randomly generated loops and some RC-Schedules

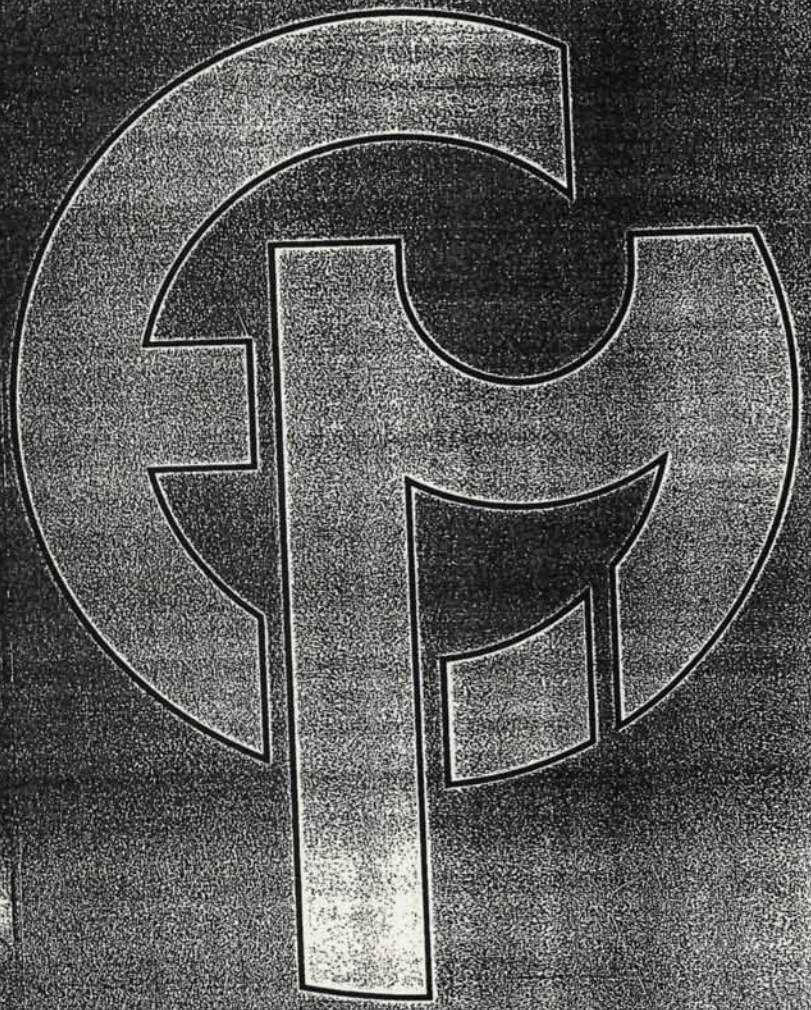
5. Park, N. and Parker, A.C. "Sehwa: A software package for synthesis of pipelines from behavioral specifications". *IEEE Trans. on Computer-Aided design*. March 1988, pp. 356-370.
6. Leiserson, C.E. and Saxe, J.B. "Retiming synchronous circuitry". *Algorithmica,6*, 1991, pp. 5-35.
7. Rau, B.R. and Glaeser, C.D. "Some scheduling techniques and an easily scheddulable horizontal architecture for high performance scientific computing". *Proc. of the fourteenth Annual Work. on Microprog.*, Oct. 1981, pp. 183-198
8. Garey, M. R. and Johnson, D.S. "A Guide to the theory of NP-Completeness". W. H. Freeman and Company, 1979.
9. Sánchez, F. and Cortadella, J. "RCLP: A novel approach for Resource-Constrained Loop Pipelining". *Technical Report*, CEPBA, num. 93/06. May, 1993.
10. Karp, R. "A characterization of the minimum cycle mean in a digraph". *Discrete Mathematics*, 23, 1978 : pp. 309-311.
11. Schroeder, M. R. "Number theory in Science Communication". *Springer-Verlag*,1990.
12. Davidson, S. et al. "Some experiments in local microcode compaction for horizontal machines". *IEEE Trans. Comput.*, July 1981, pp.460-477.
13. Cytron, R. "Compiler-Time Scheduling and Optimization for Asynchronous Machines". *Ph. D. thesis, Univ. of Illinois at Urbana-Champaign*. 1984.
14. Aiken, A. and Nicolau, A. "Optimal loop parallelization". *Proc. of the 1988 ACM SIGPLAN Conf. on Prog. Lang. Des. and Imp.*, 1988, pp. 308-317.
15. Cheng-Tsung Hwang, Yu-Chin Hsu and Youn-Long Lin. "Scheduling for Functional Pipelining and Loop Winding". *20th ACM/IEEE Design Automation Conference*June 1991, pp. 764-769.

Microprocessing and Microprogramming

VOLUME 38, NUMBERS 1–5
SEPTEMBER 1993

ISSN 0165-6074
MMICDT 38 (1–5) (1993) 1–862

The EUROMICRO Journal



COMPLETE VOLUME
PROCEEDINGS EUROMICRO 93
OPEN SYSTEM DESIGN:
HARDWARE, SOFTWARE AND
APPLICATIONS
Barcelona, September 6–9, 1993

NORTH-HOLLAND

Published for EUROMICRO by NORTH-HOLLAND
(a division of Elsevier Science Publishers B.V.)