Resource-Constrained Pipelining Based on Loop Transformations

F. Sánchez and J. Cortadella

*Polytechnic University of Catalonia, Dep. of Computer Architecture,
Campus Nord, Mòdul D6, Gran Capità s/n, Barcelona-088071, Spain
E-mail: fermin@ac.upc.es and jordi@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 dependence 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 to start before the execution of the iteration has 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.

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 resource 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

*This work was supported by CYCIT TIC-91-1096
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 $u_j$) and Global or Loop-carried dependences between operations from different iterations ($u_i$ and $u_j$, $i < j$).

A loop will be represented by a triplet called \(\pi\)-graph, \(\pi = < G, \lambda, \delta >\), where \((G(V, E))\), the dependence graph, is a directed graph and \(\lambda\) (index) and \(\delta\) (distance) are two mappings \(\lambda : V \rightarrow \mathcal{N}\) and \(\delta : E \rightarrow \mathcal{N}\). Each vertex \(v \in V\) represents an operation of the loop body. The number of operations of the loop will be denoted by \(I_\pi\). Each edge \(e = (u, v) \in E\) represents a dependence 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 \(i \in \mathcal{N}\). Graphically, it will be depicted as \(u_{\lambda(u)} \rightarrow v_{\lambda(v)}\).

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

**Definition 2.1: Equivalent \(\pi\)-graphs**

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

\[\lambda(v) - \lambda(u) + \delta(u, v) = \lambda'(v) - \lambda'(u) + \delta'(u, v)\] (1)

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

**Definition 2.2: \(G_\pi\) (Scheduling Graph)**

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

\(G_\pi\) contains all local dependences of \(\pi\). Given a \(\pi\)-graph \(\pi = < G, \lambda, \delta >\), we will define a local path \(p\) in \(G_\pi\) as a path in \(G_\pi\). The length of a path \(p\) is the number of vertices traversed by \(p\). It 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 \(\pi\))**

A schedule of a \(\pi\)-graph \(\pi\) is a mapping \(S: V \rightarrow \mathcal{N}\) that fulfills \(\forall (u, v) \in E, S(u) < S(v)\).

**Definition 2.4: \(I(\pi)\) (Initiation Interval of a Schedule)**

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

\[I(S(\pi)) = \max_{u \in V} S(u) - \min_{u \in V} S(u) + 1\]

\(I\) denotes the interval between the initiation of two consecutive iterations in the schedule. \(I\) also denotes the number of cycles of the schedule.

**Definition 2.5: \(T_\pi\) (Execution Time of \(\pi\))**

The execution time of a \(\pi\)-graph, \(T_\pi\), is the length of the critical path in its scheduling graph.

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

**2.2. Minimum initiation interval and Parallelism of a Loop**

A loop can be represented by different and equivalent \(\pi\)-graphs. While \(T_\pi\) is a property of each \(\pi\)-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 \(\pi\)-graph will be indistinctly used, and denoted by \(\text{MINI}_\pi\), as
well as parallelism of a π-graph and parallelism of a loop, denoted by $P_\pi$. 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 $\delta_R$ 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 $u_{i+1_R}$.

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 $L_R = \delta_R$.

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 necessary to include a prolog and an epilog for the correct execution of the loop[9].

Definition 2.6: $MII_\pi$ (Minimum initiation interval of $\pi$)

The minimum initiation interval of $\pi$ is

$$MII_\pi = \max_{R \in \delta E} L_R$$

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

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

$MII_\pi$ 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](image)

**Definition 2.7: $P_\pi$ (Parallelism of $\pi$)**

The parallelism of a π-graph is $P_\pi = MII_\pi$.

**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: $P_{S(\pi)} = \frac{\sum_{\pi \in \pi}}{T_{\pi}}$.

**Definition 2.9: Span($\pi$) (Span of $\pi$)**

The span of a π-graph is defined as

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

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 $MII_\pi$ is an integer, since $T_\pi$ denotes a number of cycles. In general, a π-graph representing multiple instances of the loop body will be required to force $MII_\pi$ to be an integer value. It will be called a multiple-instanted 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 $\pi^m$ in which each operation and each dependence have been replicated $m$ times.

For each vertex $A$ representing operations $A_i$ in $\pi$, $\pi^m$ has $m$ vertices ($A^0, A^1, \ldots, A^{m-1}$), so that each $A^i$ represents all the operations $A_i$ such
that \( i \mod m = j \). Likewise, for each dependence \( A \xrightarrow{d/m} B \) in \( \pi \), there will be \( m \) dependences \( A \xrightarrow{(j+m)\mod m} B \) in \( \pi^m \). Figure 2 shows an example of \( \pi \)-graph unrolling.

![Figure 2. \( \pi \)-graph unrolling](image)

\( \pi \) and \( \pi^m \) represent the same loop. However, the body of the loop represented by \( \pi^m \) will be executed only \( \frac{d}{m} \) times (minus some constant that depends on the \text{prolog} \) and \text{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 \( \pi \) be the original \( \pi \)-graph, with \( MII_{\pi} = \frac{d}{m} \).
2. Calculate \( m = \frac{b}{\text{gcd}(a, b)} \) (to generate the smallest \( M \)-graph with integer \( MII \)).
3. Generate \( \pi^m = \text{unroll}(\pi, m) \). The minimum initiation interval will be \( MII_{\pi^m} = \frac{a}{\text{gcd}(a, b)} \).

### 3.1.2. Shifting local dependences

This section proposes \text{shift\_local}, a transformation to decrease the execution time \( T_{\pi} \) of a \( \pi \)-graph until \( MII_{\pi} \).

![Figure 3. \text{Shift\_Local Transformation}](image)

The function \text{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, \text{shift\_local} may create new local dependences. Figure 3 illustrates how the transformation operates. \text{Shift\_Local} always generates an equivalent \( \pi \)-graph [9].

### 3.2. Finding a \( \pi \)-graph with \( T_{\pi} = MII_{\pi} \)

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

Given a \( \pi \)-graph \( \pi \), the following algorithm finds a \( \pi \)-graph \( \pi^t \) with \( T_{\pi^t} = MII_{\pi^t} \):

```python
function find_min_\pi-graph (\pi):
    m := \frac{b}{\text{gcd}(a, b)};
    \pi^t := \text{unroll}(\pi, m);
    while \exists MLP p in \pi^t with l(p) > MII_{\pi^t} do
        c := \text{head\_edge}(p);
        \pi^t := \text{shift\_local}(\pi^t, c);
    endwhile;
    return(\pi^t);
end:
```

\text{Find\_min\_\pi-graph} always halts and can be executed in \( O(E^3) \)-time [9].

Figure 4 shows an example of finding a \( \pi \)-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 \( H_{S^N(\pi)} \) 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 \( \pi \)-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.
4.2. Finding an RC-schedule with maximum Utilization of Resources

4.2.1. Graphic Representation of \((II, m)\)

The pairs \((II, m)\) can be represented in a diagram as shown in Figure 5(a). A pair \((II, m)\) represents the fact that \(m\) iterations are executed in \(II\) cycles. Therefore, given a loop represented by \(\pi\) and a number of resources \(N\), each pair \((II, m)\) represents a potential schedule in which resource utilization can be also denoted by \(U(II, m)\). Thus, \(U(II, m) = \frac{28}{33}\).

![Diagram](image)

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

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

4.2.2. Generation of pairs \((II, m)\)

The resource utilization of any RC-schedule of \(\pi^m\) in \(II\) cycles can be represented by the fraction \(\xi = \frac{mL}{NII}\). In order to generate all the possible values for \((II, m)\) in decreasing order of \(U(II, m)\),

![Diagram](image)
we use the recurrent definition of Farey's series \(^3\) and the solutions of the resulting linear diophantine equation \(xN + ym + \lambda = 0\).

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

4.2.3. \(\pi\)-graph Transformations and Resource-Constrained Scheduling

The algorithm to find an RC-schedule with maximum utilization of resources explores the pairs \((H, m)\) until a schedule \(S^N(\pi^m)\) such that \(H_{\Sigma^N(\pi^m)} = H\) is found. For each pair \((H, m)\) such that \(\lambda(H, m) \leq \lambda_{\text{max}}(\pi)\), \(\pi^m\) is transformed and iteratively transformed by using \text{shift} local. After each transformation the \(\pi\)-graph is scheduled in order to check whether \(H_{\Sigma^N(\pi^m)} = H\) or not.

A function is defined to heuristically evaluate the goodness of a \(\pi\)-graph for scheduling, and provide a criterion to quit the search when a schedule with length \(H\) has not been found. The two main parameters considered by the function \text{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:

```plaintext
function find_schedule(\pi, N, H, m): boolean:
    \pi' := unroll(\pi, N); \{builds \pi^m\}
    \pi'' := \pi';

Repeat
    sched := scheduling(\pi'', N);
    if (\(H_{\Sigma^N(\pi''}, N) = H\) return (true);
    selected := select_edge(\pi'', \epsilon);
    \{selects head edge or tail edge\}
    if (selected) then
        mark_edge(\pi'', \epsilon);
        \pi'' := shift_local(\pi'', \epsilon);
        if better(\pi'', \pi') then
            \pi' := \pi'';
            unmark_edges(\pi'');
        endif
    endif
Untill not(selected);
return (false); \{schedule not found\}
end
```

Among all the existing scheduling algorithms

\(^3\) The Farey series \(F_{\Sigma N \lambda_{\text{max}}}\) contains in increasing order of magnitude all the fractions with denominator lower than or equal to \(N \lambda_{\text{max}}\)

we have chosen \(\text{List Scheduling}\) \(^{[12]}\) for its efficiency and low time complexity. However, the proposed approach does not preclude the use of other scheduling algorithms.

![General view of RCLP](image)

4.2.4. Span reduction

Once an RC-schedule has been found we try to reduce the span while maintaining \(H_{\Sigma^N(\pi^m)} = H\).

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 (\(\text{Span} = m\). The graph is transformed (by means of \text{shift} local) and scheduled at each step by looking for a schedule of \(H\) cycles. The final schedule is the last found schedule in which \(H_{\Sigma^N(\pi^m)} = H\).

Figure 6 shows in a flow diagram the steps followed by the method.
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 [9] (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 maintaining 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-cycle operations and pipelined functional units are forseen.

Acknowledgments

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

REFERENCES

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


Microprocessing and Microprogramming

VOLUME 38, NUMBERS 1–5
SEPTEMBER 1993

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.)