GURPR—A Method for Global Software Pipelining

Boronn Su, Shiyuan Ding, Jian Wang and Jinshi Xia
Department of Computer Science and Technology
Tsing Hua University
Beijing, China

Abstract

The software pipelining technique is an effective approach to the optimization of loops in array processor programs and microprograms. In this paper we present a global URPR algorithm—GURPR to optimize loops of different structures based on the LURPR method we presented in 1986. We start with a brief introduction to LURPR. Then discuss the pipelining of loops with abnormal entries, conditional exits, more than one path, nested loops and subroutine calls respectively. Finally we present the complete GURPR algorithm.

1. Introduction

The software pipelining technique [1] has proved to be very effective in loop optimization. In 1986 we presented a new approach to implementing software pipelining—the URPR algorithm [9]. Experiments in applying the URPR algorithm to array processor programs [12] and microprograms in graphic processors and signal processors [3] show that the URPR algorithm has a low complexity and can reach a fairly good optimization result [6].

The software pipelining technique has been confined to the optimization of loops composed of a basic block since its idea was presented [1]. So, without exception, the URPR algorithm is only applicable to loops composed of a basic block, having known iteration number and no subroutine calls. But loops in practical programs and microprograms are much more complicated and a more general algorithm is therefore needed. We now present a global software pipelining algorithm—the Global URPR algorithm (hereafter referred to as GURPR)—that is applicable to loops of different structures.

This paper firstly outlines the original Local URPR algorithm (hereafter referred to as LURPR), then, after introducing the idea of global URPR, studies the optimization of loops with abnormal entries, conditional exits, more than one path, nested loops or subroutine calls, and finally presents the GURPR algorithm for global software pipelining.

2. The LURPR Algorithm

The LURPR algorithm was presented on the basis of the UCRR algorithm [7]. It starts by compacting the loop body, then computes the maximal inter-body data dependency/anti-dependency distance between operations in two adjacent unrolled loop bodies, determines the number of unrolled loop bodies, pipelines the unrolled loop bodies, and chooses a new loop body for rerolling. LURPR has a complexity of O(L2), where L is the original loop length. Preliminary tests indicate that the length of optimized code from LURPR is close to manually generated and optimized microcode [6].

The LURPR procedure is shown in Figure 1 and its basic algorithm is as follows.

Algorithm: LURPR

INPUT: Loop \( L = \{O_{P1}, O_{P2}, \ldots, O_{P} \} \) (a vertical list of W1's)

OUTPUT: Pipelined result \( L_{\text{PI}} = \{I_1, I_2, \ldots, I_n \} \) (a packed list of W1's)

BEGIN
1. Compact the loop body \( L \) by using, say Trace Scheduling for Global P Compaction or List Scheduling for local compaction;
2. Build the data dependency graph IB_DAG between two adjacent loop iterations and find inter-body distance \( D \);
3. Calculate \( K = \lfloor L/D \rfloor \);
4. Unroll \( L \) to form \( K \) loop bodies: \( L_{P1}, L_{P2}, \ldots, L_{PK} \);
5. Build IB_DAG's among \( L_{P1}, L_{P2}, \ldots, L_{PK} \) into \( L_{PI} \) one by one, observing the IB_DAG constraints and delaying operations in the current loop if resource conflicts exist.
6. Loop rerolling:
   (a) Find a new loop body \( L_{SI} \) from \( L_{SI} \) that has the fewest cycles and includes all OP's in \( L_{P} \) and delete redundant OP's;
3. Towards a Global URPR

We first of all recall how we conduct global microcode compaction using Tracing Scheduling[1,4]: choose a path that includes several basic blocks, then compact this path using List Scheduling. In order to satisfy the semantic equivalence before and after the compaction, we must follow several rules[1,4] when moving a microoperation: (1) a microoperation can only be moved across block boundary when it satisfies certain conditions; and (2) a bookkeeping phase is needed after the compaction of each path and some microoperation copies must be added to the other paths.

In our global URPR algorithm, we shall, as before, conduct a global compaction inside the loop body first. But we are more interested in studying the case where the microoperation in one loop body moves across the block boundary in other bodies during pipelining. We adopt the same rules for moving microoperations[1,4] in global microcode compaction as those mentioned above, the only difference is that here the conditions are stricter, bookkeeping more complicated and microoperation copies much more.

"Global" loops fall in the following cases:

1. having more than one entry, i.e., apart from normal entry, there is some abnormal entry/entries that enter(s) in the middle of the loop body;
2. having conditional exits in addition to the normal exit;
3. having more than one path;
4. having nested loops; and
5. having subroutine calls.

We present a detailed discussion for each of the five cases as follows:

3.1 Abnormal Entries

Abnormal entries (e.g., e in Fig. 2(a)) impose certain constraints to the pipelining.

A semantic error will occur in pipelining if microoperations in body1, body2,... are moved upwards above the abnormal entry e in body3 and program happens to enter the loop at e. Since microoperations in the other bodies above e are never executed.

Therefore we suggest that body3 should not be allowed to move above the abnormal entry e in body1, but the rest bodies could still be placed according to the inter-body dependency distance D only requiring that body3 be partly or completely included in the rolled loop body (See Fig. 2).

3.2 Conditional Exits

Generally, when applying the software pipelining technique to loop optimization, we assume the iteration number is pre-set and known in advance. But in the practice the termination of many loops, especially loops in microprograms, depends on some conditional test. The exits of these loops are of a different type and we term them "conditional exits" or "abnormal exits". The discussion of loops with conditional exits are intrinsically identical to the discussion of loops with unknown iteration numbers.

Two problems will occur if we directly apply URPR to loops with conditional exits:

1. Some operations in body1, body2,... may have already been executed before the execution of the conditional exit in bodyi, if control happens to leave the loop at the conditional exit, these prematurely executed operations may affect the final result and lead to a semantic error. Hence, more attention should be paid to conditional exits. Fig. 3(a) shows a loop with a conditional exit, in which body3 is referenced by condexit(i).

2. After the execution of condexit(i), some operations in body1, body2,... may not have been executed and the control happens to leave the loop at condexit(i), semantic errors may occur. Therefore we prevent these errors by putting copies of the necessary but have-not-been-executed operations between the conditional exit and the top of body3, and we refer to these copies as the "postlude" of the conditional exit (OPT) in Fig. 3(b).

On the basis of the discussion above, we make the following suggestions:

(i) The conditional exit should be placed as close to the top of the loop body as possible in compaction;
(ii) All the live registers at the top of body3 are supposed to be referenced by condexit. Dependency caused by this should be taken into account when DAG and IBDAG are being built and when D is being computed;
(iii) After rerolling, a "postlude" of the conditional exit should be added to avoid semantic errors.

Fig. 3 illustrates applying URPR to a loop with a conditional exit.

3.3 More Than One Path

The computation of vector division [9] is among the many examples of loops with more than one path.

For convenience, we shall base our discussion on a simple but typical flowchart given in Fig. 4(a), which has a branch and a rejoin. We use B1, B2, B3, and B4 to denote basic blocks. We first convert this flowchart to an equivalent form as in Fig. 4(b), then we pipeline the two paths respectively, reroll the loop and make semantic adjustments.

1. The mutual influence between the two paths needs to be well observed in pipelining. Take the example of path1 to path2 when pipelining path1:

a) some operations in B3 of body1 move upwards above the branch of body2, and they include the access to register R by i)read or ii)write;
b) control enters path2 from path1 at the branch in the ith iteration;
c) i) R is written by some operation in B4, or
   ii) R is live at the top of B5, then corresponding semantic errors will occur:
   i) the read R operation in body1 is executed before the write R operation in body2; or
ii) the write R operation in bodyi is executed before the read R operation in bodyi+1.

To avoid these errors, we advise the following "Branch Precedence Rule" to compute D: for deciding the precedence between the branch in bodyi and any operation in bodyi+1, all the live registers at the top of the other path are supposed to be read by the branch and all the registers being written in the other path to be written by the branch.

2. During the semantic adjustments, special attention should be paid to the case where control leaves one path and enters the other. Take the example of control entering path2 from path1. We make the following observations:

a) On control switching, some should-have-been-executed operations in path1 have not been executed and therefore need to be done before the execution of path2.

b) Some necessary conditions for pipelining should be satisfied. In LURPR [9], we assumed the actual iteration number n to be no less than the number of unrolled bodies, K. for if n<K., the efficiency will not be improved and the chances of semantic errors will increase. In the case of loops with two paths, we demand that the number of undone iterations, n', be no less than the number of unrolled bodies of path2, K., when control flows from path1 to path2. The same rule applies when control flows from path2 to path1, in which case we must guarantee n' >=K. To realize this, we have to examine the number of undone iterations at the exits of both paths. If the number is too small, the control must go to some "postlude" instead of entering the pipeline.

c) The entry point to path2 must be correct and optimal, and some operations may need to be added before control reaches path2. Two principles must be followed when we determine the entry point:

i) the continuity requirement of hardware pipelining units must be guaranteed: and

ii) the number of operation copies should be confined to the minimum.

We summarize the optimizing procedure of loops with two paths as follows:

a) Pipeline each path respectively and apply the Branch Precedence Rule when computing D:

b) Semantic adjustments: add necessary copies for each branch operations in both paths, add necessary test operations to examine whether control can enter the other path and determine the correct and optimal entry point:

c) Combine identical operations in the "postlude".

Algorithm DSUB:

1. Compute the inter-body distance D', neglecting the subroutine call:

2. Compute the dependency distance dCALL-TOP between operations above CALL in bodyi+1 and CALL in bodyi. (See Fig.7):

3. Compute the dependency distance dBOTTOM-CALL between CALL in bodyi and operations below CALL in bodyi+1; (As shown in Fig.7).

b) As shown in Fig.8(b) where CALLn+1 is a
conditional \texttt{CALL} and "b" is the return point. We convert Fig.8(b) to an equivalent form in Fig.8(c) and apply the \texttt{CONDCALL} algorithms.

\textbf{Algorithm \texttt{CONDCALL}}:

1. Pipeline path1 and add all the registers live in the subroutine to the read register set of branch and all the registers to be written in the subroutine to its write register set.
2. Pipeline path2: add all the registers live in B2 to the read register set of the branch and all the registers to be written in B2 to its write set, and then apply the same method as presented in a).
3. Bookkeep the overall flow and combine identical operations. If the return point falls on "a", the procedure becomes easier. From previous experiments\cite{8}, we learnt that micro-subroutine call would worsen the optimization by blocking microoperations from moving and we may well anticipate that the same will also happen to global pipelining. Therefore we suggest in cases where a short execution time is the first consideration, not using the above-presented method instead, substituting each subroutine call operation with a copy of the actual subroutine.

\textbf{Algorithm \texttt{GURPR}}:

\textbf{INPUT}: an arbitrary loop. 
\textbf{OUTPUT}: the new loop body and corresponding "prelude" and "postlude":

\textbf{BEGIN} 
1. Compact all the loops, The HTS algorithm\cite{7} applied in LURPR is no more suitable here, for it cannot satisfy the need of placing the conditional exit as close to the top as possible. Instead, Trace Scheduling is adoptable since it tends to place branches as early as possible. Also, assigning higher priority to the conditional exit and its processors will also help its early scheduling.
2. Analyze the loop structure and examine nested loops;
3. If loop L, is nested within loop L, then determine whether to pipeline both loops or pipeline one of them by following the rules introduced in section 3.4.
4. Pipeline the loop(s): 
   (1) If the loop body consists of a basic block, then apply LURPR;
   (2) If there exist more then one path in the loop body, then for each path:
      i) Compute the read and write register sets of the branch of these path by applying the Branch Precedence Rule;
      ii) Compute the read and write register sets of the conditional exit by applying the Conditional Exit Precedence Rule;
      iii) For any unconditional subroutine call, apply the DSUB algorithms;
      iv) For any conditional subroutine call, apply the \texttt{CONDCALL} algorithms;
      v) Call LURPR: If there exits any abnormal entry, then include the first body in the rerolled loop;
   (3) Bookkeeping (after rerolling)
      i) Construct necessary "prelude" and "postlude" at any branches of each path and add necessary undone iteration number testing operations;
      ii) Determine the entry point to each path and make necessary semantic adjustments;
      iii) Combine identical operations in the "postlude";
\textbf{END}.

\textbf{5. Discussion}

We believe loops of any kinds can be regarded as a combination of the basic situations above discussed, and therefore GURPR should be able to solve the software pipelining technique for any arbitrary loop. Although GURPR is much more complicated than LURPR and produces more operation copies, it is still worth applying when time efficiency\cite{8} is more important than space efficiency. The rest of this section will discuss two more cases:

\textbf{5.1 GURPR with timing constraint}

The requirement of operation continuity of hardware pipeline units\cite{9} is by nature the requirement of timing constraint. We make the following considerations:

1) When doing global compaction, we may transfer the pipelining continuity relation between OPi and OPj, to a timing pair (min, max), which means OPi can only be and must be executed in a cycle not earlier than the min-th cycle and not later than the max-th cycle after the execution of OPj. By making this transfer, we may use the algorithm presented in\cite{10} directly to conduct global compaction.
2) When pipelining the loop bodies, we place them one by one, hence even if there exist timing constraints, scheduling failure like in microcode compaction with timing constraints will not happen. That is to say, the pipelining phase is surely to succeed.
3) It is obvious that a new rerolled loop body satisfying timing constraints (i.e., the operational continuity) is always obtainable as long as a successful schedule can be reached during global compaction, for under the worst circumstances the compacted loop body can serve as the new loop body.
4) The following procedures outline an algorithm to examine if a new rerolled loop body satisfies timing constraints:

\textbf{a)} For this rerolled loop body, find all the operation pairs (OPi, OPj) that have timing constraint relation:
\begin{align*}
&\min_{i->j} < \max_{i->j} \\
&\min_{i->j} < \max_{i->j} (\text{if } i>j) \\
&\max_{i->j} < \min_{i->j} (\text{if } i<j)
\end{align*}

\textbf{b} For each pair (OPi, OPj),
\begin{enumerate}
\item Where OPi at stage i and OPj at stage j, suppose the rerolled loop body has a length of L,1<=i<j<=L, the timing between OPi and OPj is defined to be \((\min_{i->j}, \max_{i->j})\);
\item The rerolled loop body is said to satisfy timing constraints if and only if each operation pair (OPi, OPj) satisfies
\begin{align*}
&\min_{i->j} < \max_{i->j} (\text{if } i=j) \\
&\max_{i->j} < \min_{i->j} (\text{if } i<j)
\end{align*}
\end{enumerate}
5.2 Recurrence problems

We regard the recurrence problem here as a reference to the result obtained in the previous iteration(s) by an operation in the current iteration. Recurrence problems have a great affection on pipelining with URPR, for they may prolong the interbody distance (as shown in Fig.9).

We use the expression

\[ X(I)=f(X(I+K)) \quad (K<0) \]

to denote a recurrence problem, where \( f \) shows a certain operation on \( X \), and \( I \) to denote the loop index.

We divide our discussion into three with respect to the value of \( K \):

1) \( I*K \) is a positive integer. In this case, an operation in the \( N+1 \)st iteration references the result of the \( N \)th iteration (where \( N=I+K \)) and this makes it impossible to overlap body \( N+1 \) and body \( N \).

For a known iteration number \( n \), we may pipeline the first \( N \) bodies and the rest \( n-N \) respectively. For a unknown iteration number (e.g., that of loops with conditional exits), we may still use URPR by making necessary bookkeeping to recompute some wrong results.

2) \( K\leq-2 \). In this case, no recurrence exists between two adjacent bodies and they can overlap with each other to a fairly high extent. So a good pipelining result can be reached, for the interbody distance \( D \), which is largely dependent on data dependency between two adjacent bodies, is small.

3) \( K=-1 \). In this case, operations in the current iteration references the result of the last iteration. This affection is great to the pipelining of adjacent bodies and we suggest applying the method presented in ([11]) to mitigate it.

Moreover, if \( K \) is a variable, we can choose an integer \( K \) with the minimum absolute value and can solve the recurrent problem in the ways discussed above.

6. Conclusion

GURPR is a method to implement global software pipelining. All the problems raised in (9) have been solved in this paper. An example of global software pipelining of vector division code with GURPR is provided in Fig.10. Currently the GURPR algorithm is under implementation and it is expected that GURPR can satisfy more practical applications.

Acknowledgement

We wish to thank Prof. R.A.Wueller of Colorado State University for his valuable encouragement and significant help in improving the paper.

We gratefully acknowledge the support of the National Science Foundation of China.

References


Fig. 1 URPR Procedure

(a) The loop body after expansion
(b) The K unrolled loop bodies
(c) Pipelining of the K bodies
(d) The final rerolled result

Fig. 2 Applying URPR to loop with abnormal entry

(a) A loop with abnormal entry
(b) Pipelining
(c) Rerolled result

Fig. 3 Applying URPR to loop with conditional exit

(a) A loop with a conditional exit
(b) Final result
(a) A loop with two paths

(b) Respective pipelining of both paths

(c) Semantic adjustmenting after rerolling

(d) Final result

Fig. 5 Applying URPR to loop with two paths
OP₅ is a subroutine CALL operation
OP₁ writes R₁, which is live at top of the subroutine
OP₉ reads R₂, which is written in the subroutine

Fig. 7 Determination of inter_body distance D in loop with unconditional subroutine call

(a) Unconditional call (b) Conditional call with return point (c) Conditional call with return point
at a
at b

Fig. 8

(a) A loop with K = 1
OP₁ reads R, OP₅
writes R.
(b) Bodyᵣ references results of body₁, which prolongs inter_body distance D to equal the loop length
(c) A bad pipelining due to impossibility of overlapping

Fig. 9 Influence of recurrence on applying URPR
Fig. 10 Applying GURPR to vector division code sequence