Code-Size Conscious Pipelining of Imperfectly Nested Loops

Mohammed Fellahi Albert Cohen
ALCHEMY Group, INRIA Futurs, Orsay, France
{*first.last*}@inria.fr

Sid Touati
PR*i*SM, University of Versailles, France
Sid.Touati@uvsq.fr

Abstract

This paper is a step towards enabling multidimensional software pipelining of non-perfectly nested loops on memory-constrained architectures. We propose a method to pipeline multiple *inner* loops without increasing the size of the loop nest, apart from an *outermost* prolog and epilog. We focus on the domain of media and signal processing, where short inner loops are common and where embedded constraints drive the selection of code-size conscious algorithms. Our first results indicate that the additional constraints associated with the method do not impede the extraction of significant amounts of instruction-level parallelism. In addition to preserving precious scratch-pad or cache memory, our method also avoids the performance overhead of prologs and epilogs resulting from pipelined inner loops with short trip count.

Keywords Loop nest optimization, software pipelining, multidimensional retiming, code size, startup time.

1. Introduction

Software pipelining optimizes the exploitation of instruction-level parallelism (ILP) inside inner loops, with a reduced code size overhead compared to aggressive loop unrolling [23]. Even better, pipelining alone (without additional unrolling for register allocation) does not increase loop *kernel* size: typically, due to tighter packing of instructions, kernel size typically decreases on VLIW processors.

Unfortunately, most applications contain multiple hot inner loops occurring inside repetitive control structures; this is typically the case of streaming applications in media (or signal) processing, nesting sequences of filters (or transforms) inside long-lived outer loops (e.g., the so-called *time* loop). When considering embedded or stream computing systems with scratch-pad memories, fitting more inner loops on a single scratch=pad generally means higher performance, as streaming codes are generally bandwidth hungry.[1] As a result, current compilers have to trade ILP in inner loops for scratch-pad resources, leading to suboptimal performance.

2. Problem Statement

For the sake of simplicity, we will first consider two levels of nested loops, with a single outer loop, called the *global loop*, enclosing one or more inner loops, called *phases*.[2] We show that the conflict between pipelining and code size can often be a side-effect of separating the optimization of individual inner loops. *We show how to pipeline all phases without any overhead on the size of the global outer loop.*

[1] On multi-core architectures like the IBM Cell, although all efforts where made to maximize inter-core/scratch-pad connectivity, intra-core register bandwidth is still an order of magnitude higher.

[2] An analogy with a stream-oriented coarse grain scheduling algorithm [15].

Technically, we propose to modulo-schedule [23] as many phases as possible, while merging the prolog of each outer iteration of a phase with the epilog of its previous outer iteration. Such prolog-epilog merging is enabled by an outer loop retiming (or shifting) [16, 6] step, at the cost of a few additional constraints on modulo scheduling. It is then possible to reintegrate the merged code block within the pipelined kernel, restoring the loop to its original number of iterations. This operation is not always possible, and depends on the outer loop's dependence cycles. Indeed, after software-pipelining, prolog-epilog merging may affect phases that are in dependence with statements shifted by the software pipeline (along an inner loop). This makes our problem more difficult than in the perfectly nested case [24].

We consider low-level code after instruction selection but before register allocation; we thus ignore scalar dependences except def-use relations, and break inductive def-use paths whenever a simple closed form exists. We assume all dependences are uniform, modeled as constant distance vectors whose dimension is the common nesting depth between the source and sink of the dependence [1]. We capture all dependences in a directed graph G labeled with dependence vectors.

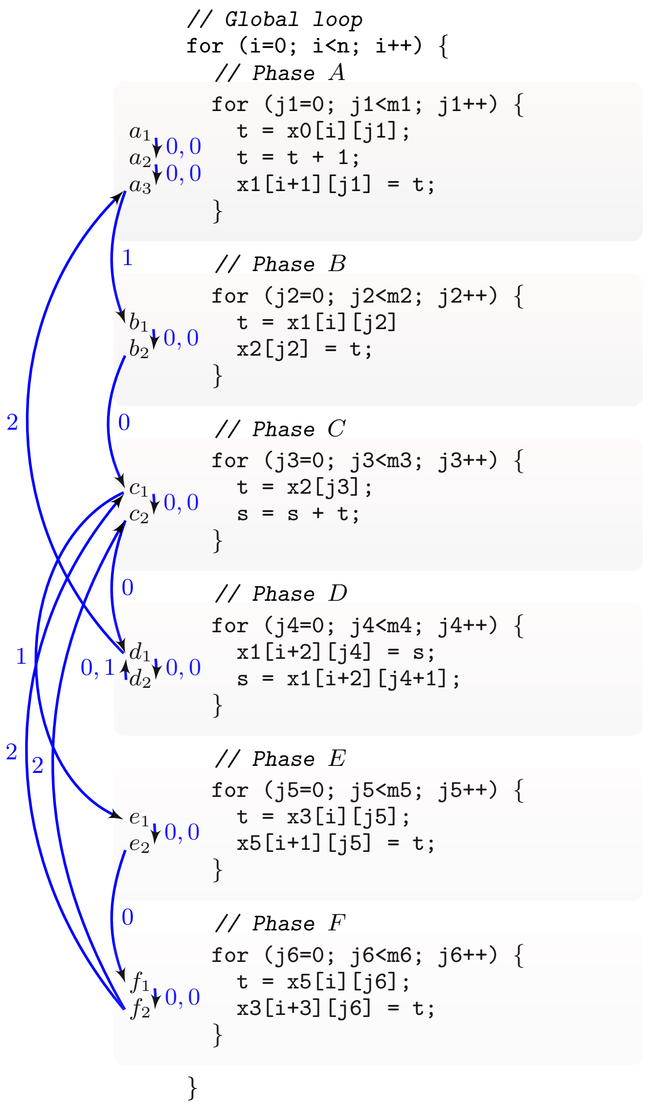
2.1 Running Example

Our running example is given in Figure 1. Statements and phases are labeled. Both intra-phase and inter-phase dependence vectors are shown.

The classical approach to the optimization of such an example is (1) to look for high-level loop fusions that may improve the locality in inner loops, often resulting in array contraction and scalar promotion opportunities [1, 29], and (2) to pipeline the phases (inner loops) whose trip count is high enough. We assume the first loop fusion step has been applied, and that further fusion is hampered by complex dependence patterns or mismatching loop trip counts. The result of the second step is sketched using statement labels in Figure 2; notice the *modified termination condition* in pipelined phases. As expected, this dramatically improves ILP, at the expense of code size increase. In addition, some ILP is lost in the prolog and epilog of each phase, and this results in accumulated overhead across the execution of the global loop.

The alternative is to shift the prolog of each pipelined phase, advancing it by one iteration of the global loop, then to merge it with the corresponding epilog of the previous iteration of the global loop. This is not always possible, and we will show in the following sections how to formalize the selection of phases subject to pipelining as a linear optimization problem.

Back to our running example, a possible solution is to pipeline and apply prolog-epilog merging to phases A, B, E and F. The code after advancing the prologs of pipelined phases is outlined in Figure 3; notice the outermost prolog — resulting from advancing the first global iteration of the phase prologs — and epilog — the last global iteration of phase epilogs. Yet this code is incorrect, for two reasons.



```
// Global loop
for (i=0; i<n; i++) {
  // Phase A
  for (j1=0; j1<m1; j1++) {
    t = x0[i][j1];
    t = t + 1;
    x1[i+1][j1] = t;
  }
  // Phase B
  for (j2=0; j2<m2; j2++) {
    t = x1[i][j2]
    x2[j2] = t;
  }
  // Phase C
  for (j3=0; j3<m3; j3++) {
    t = x2[j3];
    s = s + t;
  }
  // Phase D
  for (j4=0; j4<m4; j4++) {
    x1[i+2][j4] = s;
    s = x1[i+2][j4+1];
  }
  // Phase E
  for (j5=0; j5<m5; j5++) {
    t = x3[i][j5];
    x5[i+1][j5] = t;
  }
  // Phase F
  for (j6=0; j6<m6; j6++) {
    t = x5[i][j6];
    x3[i+3][j6] = t;
  }
}
```

Figure 1. Running example

```
// Global loop
for (i=0; i<n; i++)
A     $a_1$
      $a_1 \| a_2$
      for (j1=0; j1<m1-2; j1++)
        $a_1 \| a_2 \| a_3$
      $a_2 \| a_3$
      $a_3$

B     $b_1$
      for (j2=0; j2<m2-1; j2++)
        $b_1 \| b_2$
      $b_2$

C     $c_1$
      for (j3=0; j3<m3; j3++)
        $c_1 \| c_2$
      $c_2$

D     for (j4=0; j4<m4-1; j4++)
        $d_1$
        $d_2$

E     $e_1$
      for (j5=0; j5<m5-1; j5++)
        $e_1 \| e_2$
      $e_2$

F     $f_1$
      for (j6=0; j6<m6-1; j6++)
        $f_1 \| f_2$
      $f_2$
```

Figure 2. Software pipelining all phases independently

- The inter-phase dependence from statement e_2 to statement f_1 is violated, since f_1 in the prolog of phase F has been anticipated before one full iteration of phase E; some instances of this violation are depicted by a bold arc on Figure 3. To fix this violation, one may shift the whole phase E, advancing it by one iteration of the global loop. This is possible since the only inter-phase dependence targeting phase E (statement e_1) has a non-null distance.
- A similar problem exists with the inter-phase dependence from statement b_2 to statement c_1; some instances of this violation are depicted by a bold dashed arc on Figure 3. Yet we will see that this violation cannot be fixed by shifting, due to the accumulation of shifting constraints on the cycle of inter-phase dependences involving A, B, C and D. We choose not to pipeline C in the following; we will later demonstrate the optimality of this choice after formalizing the global optimization problem.

The final code after pipelining all phases but C,[3] prolog-epilog merging, shifting E, and reintegrating the merged prologs and epilogs into the kernels is outlined in Figure 4; notice the *modified termination condition* on the global loop, and the *restored termination condition* on the pipelined phases (due to prolog-epilog merging).

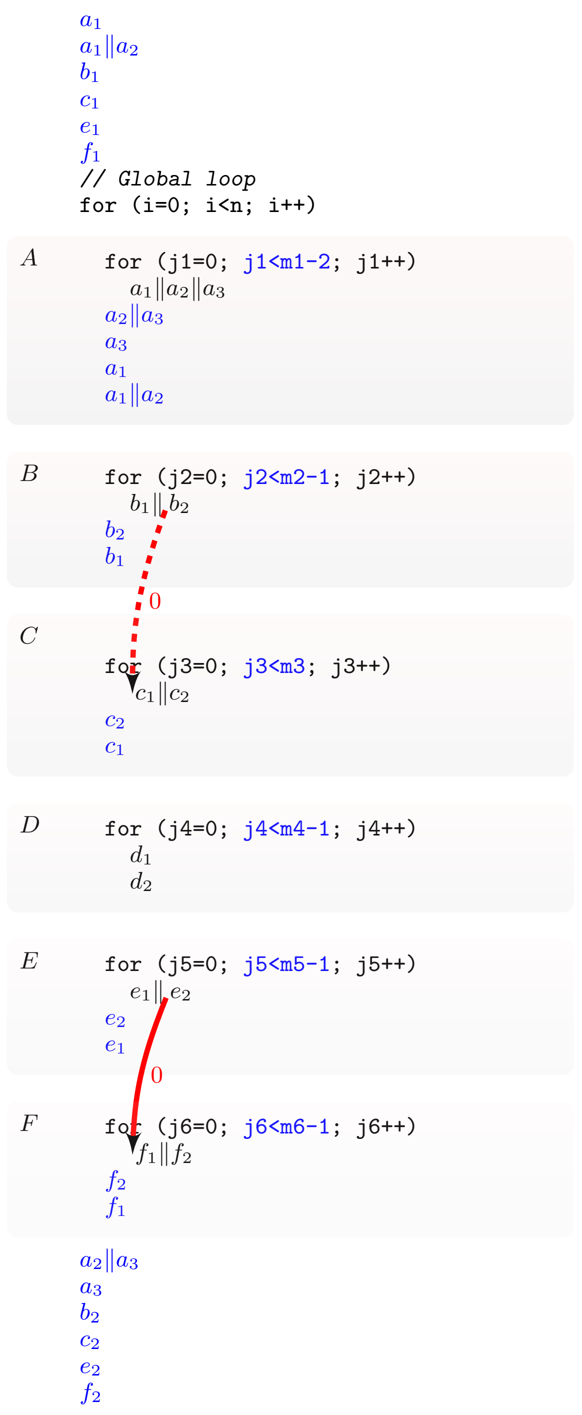
The body of the global loop recovered its original size, and prolog/epilog overhead has disappeared. This major improvement was done at the minor expense of the loss of ILP on phase D, and some extra code outside the global loop, due to the global shifting of phase E.[4]

2.2 Inter-Phase Dependences

In the following, shifting is understood as *advancing* the execution of a statement by one or more iterations. For example, shifting b_1 implies that the first iteration of b_1 (or more) will end up in a prolog of phase B; this prolog will have to be merged with the epilog of this phase for the previous iteration of the outer loop.

[3] Attempting to pipeline D does not bring any ILP.

[4] The first iteration of the global loop executes E only, while the last iteration executes every phase but E.



```
$a_1$
$a_1||a_2$
$b_1$
$c_1$
$e_1$
$f_1$
// Global loop
for (i=0; i<n; i++)

A     for (j1=0; j1<m1-2; j1++)
        $a_1||a_2||a_3$
      $a_2||a_3$
      $a_3$
      $a_1$
      $a_1||a_2$

B     for (j2=0; j2<m2-1; j2++)
        $b_1||b_2$
      $b_2$
      $b_1$

C     for (j3=0; j3<m3; j3++)
        $c_1||c_2$
      $c_2$
      $c_1$

D     for (j4=0; j4<m4-1; j4++)
        $d_1$
        $d_2$

E     for (j5=0; j5<m5-1; j5++)
        $e_1||e_2$
      $e_2$
      $e_1$

F     for (j6=0; j6<m6-1; j6++)
        $f_1||f_2$
      $f_2$
      $f_1$

$a_2||a_3$
$a_3$
$b_2$
$c_2$
$e_2$
$f_2$
```

Figure 3. Advancing prologs of pipelined phases (incorrect code)

Since the dependence from a_3 to b_1 is carried by the outer loop, its associated distance (0) does not tell anything about the precise iterations of b_1 within phase B that are in dependence. Shifting b_1 along the inner loop — by any positive amount — is thus equivalent to shifting the whole phase B by 1 iteration of the outer loop. This observation is key to converting our prolog-epilog merging problem into a classical retiming one.

```
// Prolog for shifted iteration of E
$e_1$
for (j5=0; j5<m5; j5++)
  $e_1||e_2$

// Prologs of A, B, and F
$a_1$
$a_1||a_2$
$b_1$
$f_1$

// Global loop
for (i=0; i<n-1; i++)

A     for (j1=0; j1<m1; j1++)
        $a_1||a_2||a_3$

B     for (j2=0; j2<m2; j2++)
        $b_1||b_2$

C     for (j3=0; j3<m3; j3++)
        $c_1$
        $c_2$

D     for (j4=0; j4<m4; j4++)
        $d_1$
        $d_2$

E     for (j5=0; j5<m5; j5++)
        $e_1||e_2$

F     for (j6=0; j6<m6; j6++)
        $f_1||f_2$

// Epilog for shifted interation of E
$e_2$
for (j1=0; j1<m1; j1++)
  $a_1||a_2||a_3$
for (j2=0; j2<m2; j2++)
  $b_1||b_2$
for (j3=0; j3<m3; j3++)
  $c_1$
  $c_2$
for (j4=0; j4<m4; j4++)
  $d_1$
  $d_2$
for (j6=0; j6<m6; j6++)
  $f_1||f_2$

// Epilogs of A, B, and F
$a_2||a_3$
$a_3$
$b_2$
$f_2$
```

Figure 4. Software pipelining with prolog-epilog merging

3. Characterization of Pipelinable Phases

From the global dependence graph G with multidimensional dependence vectors, the *phase dependence graph* G_p is defined as follows:

- nodes of G_p are the phases;
- an arc links a phase A to a phase B if and only if there is a path in G from a statement a of A to statement b of B; to avoid spurious transitively covered arcs, we also require this path to contain a single inter-phase arc;
- the distance associated with an arc of G_p is the sum of the distances, for the dimension of the global loop, along the corresponding path from a to b in G.

Arcs in G_p will be called *phase dependences*. They correspond to one inter-phase dependence and zero or more transitively-covered intra-phase dependence.

Notice the distance associated with a phase dependence takes into account non-zero distances along the outer dimension of intra-phase dependences.

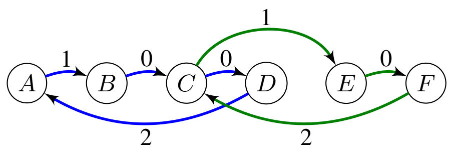

$$\begin{tikzcd}
A \arrow[r, bend left, "1"] & B \arrow[r, bend left, "0"] & C \arrow[r, bend left, "0"] \arrow[rr, bend left, "1"] & D \arrow[lll, bend left, "2"] & E \arrow[r, bend left, "0"] & F \arrow[lll, bend left, "2"]
\end{tikzcd}$$

Figure 5. Phase dependence graph

Figure 5 shows the phase dependence graph for the running example.

3.1 Causality Condition

Every time a phase is software pipelined, we just showed that merging its prolog and epilog is equivalent — when considering G_p — to shifting the whole phase by 1. To guarantee that all phases can be pipelined and their prolog and epilog merged, it is thus sufficient that every forward arc in G_p has distance $d > 0$, and any backward arc has distance $d > 1$.

This is of course too restrictive, and in general we are back to a traditional retiming problem [16]. Pipelining all phases is possible if and only if, for any cycle C,

$$\sum_{p \in C} d_p - \mathit{nb\_backward\_edges}(C) \geq \mathit{nb\_phases}(C). \quad (1)$$

We can state a more general result.

Let us define

$$k_C \stackrel{\text{def}}{=} \sum_{p \in C} d_p - \mathit{nb\_backward\_edges}(C). \quad (2)$$

THEOREM 1. *For every cycle, the number of phases that can be safely pipelined is greater than or equal to k_C.*

This is only a lower bound, as we did not captured in G_p whether pipelining a phase did result in an intra-phase shifting of the specific statements involved in some inter-phase dependence.

Proof. Let us prove this result. Let $a(i, p, j)$ denote an instance of instruction a, given an iteration i of the global loop, a phase p and an iteration j of p. Let $t_{a(i,p,j)}$ denote the execution time of $a(i, p, j)$ and $a(i, p)$ denote the set of instances of a at global loop iteration i.

$$b(i', p') \text{ depends on } a(i, p) \text{ with dependence distance } d \\ \implies \forall j, j', t_{a(i,p,j)} < t_{b(i',p',j')} \text{ and } i \leq i'. \quad (3)$$

Indeed, a phase dependence in G_p between p and p' corresponds to dependences between two sets of statement instances $a(i, p)$ and $b(i', p')$.

Software pipelining p' may imply shifting occurrences of instruction a. We call c_j the associated shifting distance along p', c_i the shifting distance along the global loop, and we consider two cases.

Forward edge. If p' depends on p with distance d and p' follows p in the loop nest, c_i must be chosen such that $d \geq 0$.

Backward edge. If p' depends on p with distance d and p' precedes p in the loop nest, c_i must be chosen such that $d > 0$.

We may compute c_i, taking into account the global loop shifts over outgoing arcs, the distance d, and whether p' is pipelined or not. The global loop shifts and d are the classical retiming variables and parameters. What happens to p' can be modeled easily, as we previously observed in Section 2.2 that shifting along an inner loop by any amount c_j can be compensated by shifting along the global loop by 1.

Therefore software pipelining p' will increase the total pressure over a cycle by at most 1. This constraint can be modeled by decrementing the distance d when p' is pipelined. We are back to a classical retiming problem, from which we deduce that p' can be pipelined if decrementing d does not induce any cycle with negative or null distance in G_p.

A simple recurrence on the number of pipelined phases concludes the proof.

3.2 Necessary and Sufficient Condition

In the absence of any information about the statements involved as sink and source of phase dependences, one may only assume that pipelining a phase will incur a shifting constraint along the global loop. In this case, the sufficient condition becomes a necessary one, and the previous proof can be extended to show that the number of phases that can be pipelined while merging prologs and epilogs is exactly k_C, as defined by (2).

Conversely, when considering the full dependence graph G, it is possible to constrain the pipelining of individual phases so that to forbid any inner loop shifting on some specific statements (targets of inter-phase dependences). This will allow to further pipeline some phases without impacting retimability of the global loop. We will come back to this extension when describing the complete algorithm.

4. Global Optimization Problem

Based on Theorem 1, we can formalize the software pipelining of multiple inner loops with prolog-epilog merging as a global optimization problem.

4.1 Multidimensional Knapsack Problem

First of all, the causality preservation condition in Theorem 1 needs to be extended to cover the whole phase dependence graph G_p. Indeed, software-pipelining k_C phases for each cycle C may create a retiming conflict, as a phase may belong to several cycles and can be chosen to be software-pipelined for one cycle and not for another.

The subject is not to software pipeline exactly k_C phases for each cycle C but to minimize the global outer loop execution time. Since it is not possible to software pipeline more than k_C phases for each cycle C, we have to maximize an objective function under some constraints. The objective function associated with the (static) cycle count for the loop nest is the sum over all phases p of

$$\mathit{profit}_p = \mathit{seqtime}_p - m_p II_p,$$

where $\mathit{seqtime}_p$ is the number of cycles to execute phase p and II_p is the initiation interval for the pipelined version of phase p. Let $w_{Cp} \in \{0, 1\}$ denote whether phase p belongs to cycle C. The optimization problem is the following:

$$\left\{ \begin{array}{ll} \text{variables:} & \forall p \in \{1, \dots, \mathit{nb\_phases}\}, X_p \in \{0, 1\} \\ \text{objective:} & \max \sum_{p=1}^{\mathit{nb\_phases}} \mathit{profit}_p X_p \\ \text{constraints:} & \forall C \in \{1, \dots, \mathit{nb\_cycles}\}, \\ & \qquad \sum_{p=1}^{\mathit{nb\_phases}} w_{Cp} X_p \leq k_C \end{array} \right. \tag{4}$$

This is a multidimensional Knapsack problem, a well known NP-complete problem; unlike the one-dimensional case, there is no known pseudo-polynomial algorithm [21] but some heuristics give good results [22].

4.2 Algorithm

1. If for every cycle
   $$k_C \geq \mathit{nb\_phases}$$
   then software-pipeline each phase independently.
2. Otherwise:
   - solve the multidimensional knapsack problem to identify which are the k_C phases to pipeline;
   - retime the global outer loop, considering phase dependences in G_p, reducing their distance by one every-time the sink phase has been pipelined and contains intra-phase shifted statements at the sink of an inter-phase dependence; this step is guaranteed to terminate according to Theorem 1.
3. As an optional extension, pipeline all remaining phases with the additional constraint that statements at the sink of an inter-phase dependence may not be shifted; this may be easily modeled in any modulo scheduling algorithm by placing such statements initially in column 0 [23]. This step is guaranteed not incur global retiming constraints.
4. Generate code, gathering all prologs and epilogs from pipelined phases, and iterating on them according to the retiming of the global outer loop.

5. Back to the Running Example

Figure 2 showed how to software pipeline all phases independently. This allows to compute the initiation interval II_p for every phase p. The profit of pipelining a phase is the difference in (static) execution cycles between executing the original inner loop body and the pipelined version. Figure 6 shows the profit for all phases in the running example, assuming the trip counts of all phases are identical and equal to $m = m_1 = \cdots = m_6$.

Phase	A	B	C	D	E	F
Profit	$2m$	m	m	0	m	m

Figure 6. Profit table

The graph G_p was given in Figure 5. It consists of two cycles, $(ABCD)$ and (CEF). These cycles share phase C, which makes the optimization problem even more interesting as a naive approach may select C to be pipelined for one cycle but not for the other. Figure 7 shows k_C, the maximum number of phases that can be pipelined for each cycle.

Cycle	$ABCD$	CEF
k_C	2	2

Figure 7. Cycle retiming constraints

Overall, we have to solve the following optimization problem:

$$\left\{ \begin{array}{l} X_j \in 0, 1 \\ \max(2X_1 + X_2 + X_3 + X_5 + X_6) \\ X_1 + X_2 + X_3 + X_4 \leq 2 \\ X_3 + X_4 + X_5 + X_6 \leq 2 \end{array} \right.$$

A greedy approximation of the solution orders phases from the most profitable phase to the less profitable one, and selects as many phases as possible for software pipelining, while respecting the k_C constraint for every cycle C. The result for the running example is to pipeline A, C, and E, with a total profit of $4m$.

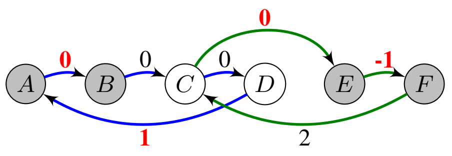


Figure 8. Modified phase dependence graph after pipelining A, B, E and F

The multidimensional knapsack solution is better: phases A, B, E, F are pipelined, with a total profit of $5m$. Figure 8 shows the modified phase dependence graph, where pipelined phases are shaded, and the decremented distances of incoming arcs appear in a bold face — following the retiming model of the proof of Theorem 1. Notice phase C is more profitable than B, but pipelining B instead gives us a chance to choose another phase for the other cycle and increases the total profit. This corresponds to a speedup of $13/(13-8) = \mathbf{1.625}$.

As this example shows, it may be overall more effective to pipeline less profitable phases but maximize the profit on every cycle. This observation is very natural when the phases have a different trip count, but our running example shows that this may also occur when cycles in the phase dependence graph are not disjoint.

The resulting code, with prolog-epilog merging and generation of the global loop's prolog and epilog was shown in Figure 4.

6. Ongoing and Future Work

Our method goes beyond incrementally extending software pipelining to nested loops. It features a slightly more complex change in the schedule of statement instances, compared to plain retiming. This raises many questions, some of which are discussed below.

6.1 Experimental Studies

We are currently implementing the algorithm. The lack of reference dependence graphs for multi-dimensional pipelining problems will limit our ability to compare with previous work in the area; it is also the reason why we are not able yet to present experimental figures. However, studying well known media, signal-processing and numerical codes, we have gathered enough results to become confident about the practical applicability of the technique.

Indeed, a large number of codes exhibit a global *time* loop and a series of inner loops: this is the case for loop nests resulting from the scheduling of Synchronous Data-Flow graphs (SDF) [10, 2, 15, 14]. The static scheduling of SDF is an important problem arising from the mapping of data-flow applications onto multi-processors. In most cases, these codes have no inter-phase dependence cycles: the multidimensional knapsack problem is trivial, and we know that all phases can be freely pipelined, maximizing the ILP gains without code size increase inside the global loop.

In addition, when mapping and generating code for SDF, it is necessary to allocate the FIFO buffers to communicate live data between actors [10]. Enforcing a bound on a buffer will result into a back-pressure, modeled as a memory-based anti-dependence [20]. Some of these dependences may eventually build cycles with non-unit distance similar to those of the running example.

As a generalization, many dependence cycles in nested loops involve memory-based — write-after-write (output) and write-after-read (anti) — dependences. In this case, renaming and privatization [27, 17] allow to eliminate any unnecessary dependence that does not carry the flow of a value [11]. To reduce the memory footprint, one generally combine these expansion techniques with array contraction (a.k.a. folding) [12], or the more general storage mapping optimization [25, 4].

Overall, combining preliminary transformations including loop rerolling, fusion and distribution, unroll-and-jam, privatization, contraction, if-conversion and inlining [1], we found excellent applications in production-quality implementations of 802.11a, JPEG and MPEG4 (de)compression, GNU radio and polyphase filtering.[5]

6.2 Managing Code Growth With Predicated Execution

Our method results in code growth outside the global loop. This is much less harmful to performance than inner prologs and epilogs, yet it may still cause some problems on memory-constrained embedded architectures. In addition, this growth is amplified by the accumulation of global loop shifts induced by the prolog-epilog merging constraints. Rather than fully hoisting the prolog and epilog code, an alternative code generation strategy for predicated ISAs consists in guarding the phases with predicate registers conditioned by the global loop iterator. This is well known for single-dimensional pipelining: Intel ICC uses rotating predicate registers and a dedicated counted loop instruction to collapse the prolog and epilog inside the modulo-scheduled kernel in most cases [7, 9]. A extension of this technique could be crafted here, with the additional complexity to manage two levels of nesting and predication; it would also induce some predicate manipulation overhead in the absence of hardware support for nested rotating registers. Eventually, unlike the single-loop prolog-epilog collapsing via predication [9], our method would not incur any performance overhead: pipeline depth has no influence on startup time, as prologs and epilogs are hoisted outside the global loop.

6.3 Managing Register Pressure

There is an unfortunate side-effect of retiming a prolog (resp. epilog) along the global loop: any live variable entering (resp. leaving) the pipelined kernel will interfere with *every variable in other phases*. The effect on register pressure can be disastrous [26]. There are multiple ways to tackle with this problem.

- The increased pressure is comparable to aggressive scheduling of unrolled or fused loops [18, 3]. We have seen that prolog-epilog merging competes with loop fusion when dependence patterns are simple enough; our technique has similar impact on register pressure, which should be encouraging given the practical importance of loop fusion among loop optimizations for memory locality and ILP enhancement.
- It is always possible to spill/fill live variables at phase boundaries. This operation is very likely to be cheaper than running through the low-ILP epilog of a deeply pipelined inner loop. It is even more likely to be shorter, especially on architectures with ISA support for register spill/refill like IA64's register stack engine [19], register windows (Sparc), or multi-push/multi-pop operations (CISC).
- When applying our technique to coarser grain, process scheduling, interprocess communication goes through FIFO channels generally implemented as scratch-pad memories or caches [5, 14]. The increased pressure is similar to the effect of array renaming in this case, and rarely a practical problem compared to array privatization [5].

6.4 Integration Into a Loop Nest Optimization Framework

Associated with our ongoing experimental work, we are working on three different extensions.

First of all, it will be easy to generalize the algorithm to arbitrarily deeply nested loops, provided a strategy to pipeline individual perfectly nested loops can be defined [24], as well as a code generation technique accommodating for imperfectly nested phases.

In addition, we will try to combine our method with other forms of multi-level pipelining and combined pipelining and unroll-and-jam [3, 24], to maximize the extraction of fine grain parallelism through multiple levels of shifting. E.g., considering phase C of the running example, it is possible to improve ILP by shifting c_1, advancing it by one iteration of the *global* loop.

Eventually, we plan integrate our technique into a more general loop nest optimization framework, based on the polyhedral model [13, 28]; this framework may not allow to model fine-grain resources (instruction-level reservation tables), but will extend to even more important optimizations for locality and parallelism [8].

7. Conclusion

Software pipelining nested loops is not a new idea. Yet our prolog-epilog merging method may appear as the most natural extension to single loop pipelining. Indeed, it avoids the code size and startup time overhead of nested prologs and epilogs: these advantages are exactly the motivations backing software pipelining in favor of loop unrolling. We formalize the prolog-epilog merging idea, combining single loop pipelining and multidimensional retiming, modeling the global scheduling constraints as a multidimensional knapsack problem.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[2] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. J. in Computer Simulation*, 4(2):155–182, 1994.

[3] S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In *Proceedings of the 29th Hawaii Intl. Conf. on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*. IEEE, 1996.

[4] P. Carribault and A. Cohen. Application of storage mapping optimization to register promotion. In *Intl. Conf. on Supercomputing (ICS'04)*, pages 247–256, St-Malo, France, June 2004.

[5] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom memory management methodology*. Kluwer Academic, 1998.

[6] A. Darte, G.-A. Silber, and F. Vivien. Combining Retiming and Scheduling Techniques for Loop Parallelization and Loop Tiling. *Parallel Processing Letters*, 7(4):379–392, 1997.

[5] All benchmarks studied in the context of the ACOTES and SARC FP6 European projects:
`http://www.hitech-projects.com/euprojects/ACOTES`,
`http://www.sarc-ip.org`.

[7] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt. Overlapped loop support in the Cydra 5. In *Intl Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'89)*, pages 26–38, Apr. 1989.

[8] A. Douillet and G. R. Gao. Software-pipelining on multi-core architectures. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'07)*, Brasov, Romania, Sept. 2007. To appear.

[9] C. Dulong, R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr. An overview of the Intel IA-64 compiler. *Intel Technical Journal*, Q4, 1999.

[10] D. G. M. E. A. Lee. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers*, 36(1):24–25, 1987.

[11] P. Feautrier. Array expansion. In *Intl. Conf. on Supercomputing (ICS'88)*, pages 429–441, St. Malo, France, July 1988.

[12] G. Gao, R., V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *LCPC'5 Fifth Workshop on Languages and Compilers for Parallel Computing, LNCS 757*, pages 281–295, New Haven, august 1992.

[13] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl. J. of Parallel Programming*, 2006. Special issue on Microgrids. 57 pages.

[14] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, San Jose, CA, Oct. 2006.

[15] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. In *LCTES'03*, June 2003.

[16] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, Dec. 1991.

[17] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *Principles of Programming Languages (PoPL'93)*, pages 2–15, Charleston, South Carolina, Jan. 1993.

[18] K. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, july 1996.

[19] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, pages 44–55, Mar. 2003.

[20] A. Moonen, M. Bekooij, and J. van Meerbergen. Timing analysis model for network based multiprocessor systems. In *Proc. of ProRISC, 15th annual Workshop of Circuits, System and Signal Processing*, pages pages 91–99, Veldhoven, The Netherlands, Nov. 2004.

[21] R. Parra-Hermandez and N. J. Dimopoulos. A new heuristic for solving the multichoice multidimensional knapsack problem. *IEEE Transactions on Systems, Man, and Cybernetics — Part A: Systems and Humans*, 35(5), Sept. 2005.

[22] J. Puchinger, G. R. Raidl, and U. Pfershy. The multidimensional knapsack problem: Structure and algorithms. *Technical Report No. 006149 INFORMS Journal of Computing*, Mar. 2007.

[23] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, New York, NY, USA, 1994. ACM Press.

[24] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao. Single-dimension software pipelining for multi-dimensional loops. In *Proceedings of the International Symposium on Code generation and Optimization(CGO'04)*, Mar. 2004.

[25] M. M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independant storage mapping for loops. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, 8, 1998.

[26] S. Touati and C. Eisenbeis. Early Control of Register Pressure for Software Pipelined Loops. In *Proceedings of the International Conference on Compiler Construction (CC)*, Warsaw, Poland, Apr. 2003. Springer-Verlag.

[27] P. Tu and D. Padua. Automatic array privatization. In *Languages and Compilers for Parallel Computers (LCPC'93)*, number 768 in LNCS, pages 500–521, Portland, Oregon, Aug. 1993.

[28] N. Vasilache, A. Cohen, and L.-N. Pouchet. Automatic correction of loop transformations. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'07)*, Brasov, Romania, Sept. 2007. To appear.

[29] S. Verdoolaege, M. Bruynooghe, G. Janssens, and F. Catthoor. Multi-dimentsional incremetal loops fusion for data locality. In *ASAP*, pages 17–27, 2003.