Single-Dimension Software Pipelining for Multi-Dimensional Loops

Hongbo Rong †, Zhizhong Tang ♯, R.Govindarajan ‡, Alban Douillet †, Guang R.Gao †

† Department of Electrical and Computer Engineering
University of Delaware
Newark, DE 19716, USA
{*rong,douillet,ggao*}*@capsl.udel.edu*

♯ Department of Computer Science and Technology
Tsinghua University
Beijing, 100084, China
tzz-dcs@tsinghua.edu.cn

‡ Supercomputer Edn. & Res. Centre
Computer Science & Automation
Indian Institute of Science
Bangalore 560 012, India
govind@serc.iisc.ernet.in

Abstract

Traditionally, software pipelining is applied either to the innermost loop of a given loop nest or from the innermost loop to outer loops. In this paper, we propose a three-step approach, called Single-dimension Software Pipelining (SSP)*, to software pipeline a loop nest at an arbitrary loop level.*

The first step identifies the most profitable loop level for software pipelining in terms of initiation rate or data reuse potential. The second step simplifies the multi-dimensional data-dependence graph (DDG) into a 1*-dimensional DDG and constructs a* 1*-dimensional schedule for the selected loop level. The third step derives a simple mapping function which specifies the schedule time for the operations of the multi-dimensional loop, based on the* 1*-dimensional schedule. We prove that the SSP method is correct and at least as efficient as other modulo scheduling methods.*

We establish the feasibility and correctness of our approach by implementing it on the IA-64 architecture. Experimental results on a small number of loops show significant performance improvements over existing modulo scheduling methods that software pipeline a loop nest from the innermost loop.

1. Introduction

Loop nests are rich in coarse-grain and fine-grain parallelism and substantial progress has been made in exploiting the former [7, 10, 11, 13, 18, 6]. With the advent of ILP (Instruction-Level Parallelism) processors like VLIW, superscalar, and EPIC [1], and the fast growth in hardware resources, another important challenge is to exploit fine-grain parallelism in the multi-dimensional iteration space [14, 23, 20].

Software pipelining [1, 2, 14, 16, 17, 20, 23, 24, 25, 30] is an effective way to extract ILP from loops. While numerous algorithms have been proposed for single loops or the innermost loops of loop nests [1, 3, 16, 17, 19, 24, 25], only a few address software pipelining of loop nests [17, 20, 30]. These methods share the essential idea of scheduling each loop level successively, starting with the innermost one.

In hierarchical modulo scheduling [17], an inner loop is first modulo scheduled and is considered as an *atomic-operation* of its outer loop. The process is repeated until all loop levels are scheduled, or available resources are used up, or dependences disallow further parallelization. The inefficiency due to the filling and draining (prolog and epilog parts) of the software pipelined schedule for the inner loops is addressed in [20, 30].

In this paper, we refer to the above approach as modulo scheduling from the innermost loop, or *innermost-loop-centric modulo scheduling*. The innermost-loop-centric approach naturally extends the single loop scheduling method to the multi-dimensional domain, but has two major shortcomings: (1) it commits itself to the innermost loop first without considering how much parallelism the other levels have to offer. Software pipelining another loop level first might result in higher parallelism. (2) It cannot exploit the data reuse potential in the outer loops [8, 9]. Other software pipelining approaches for loop nests [14, 23] do not consider resource constraints.

Lastly, there have been other interesting work that combine loop transformation (e.g., unimodular transformation, etc.) with software pipelining [32, 8]. However, in these methods, software pipelining is still limited to the innermost loop of the transformed loop nest.

In this paper, we present a unique framework for resource-constrained software pipelining for a class of loop nests. Our approach can software pipeline a loop nest at an arbitrary loop level. It extends the traditional innermost

Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)
0-7695-2102-9/04 $20.00 © 2004 IEEE

loop software pipelining scheme to allow software pipelining to be applied to the most "beneficial" level in a loop nest in order to better exploit parallelism and data reuse potential in the whole iteration space and match it to the available machine resources.

The problem addressed in this paper is formulated as follows: *Given a loop nest composed of n loops $L_1, L_2, \ldots, L_n$, identify the most profitable loop level L_x ($1 \leq x \leq n$) and software pipeline it*. Software pipelining L_x means that the consecutive iterations of L_x will be overlapped at run-time. The other loops will not be software-pipelined. In this paper, we only discuss how to parallelize the selected loop L_x. Its outer loops, if any, remain intact in our approach and can be parallelized later.

The above problem can be broken down into two subproblems: how to predict the benefits of software pipelining a given loop level, and how to software pipeline the loop nest at the chosen loop level.

Our solution consists of three steps:

1. *Loop selection*: This step searches for the most profitable loop level in the loop nest. Here profitability can be measured in terms of initiation rate, or data reuse potential, or both.
2. *Dependence simplification and 1-D schedule construction*: The multi-dimensional DDG of the loop nest is reduced to a 1-dimensional (1-D) DDG for the selected loop. Based on the 1-D DDG and the resource constraints, a modulo schedule, referred to as the *1-D schedule*, is constructed for the operations in the loop nest. No matter how many inner loops the selected loop level has, it is scheduled as if it were a single loop.
3. *Final schedule computation*: Based on the resulting 1-D schedule, this step derives a simple mapping function which specifies the schedule time of operations of the multi-dimensional loop.

Since the problem of multi-dimensional scheduling is reduced to 1-dimensional scheduling and mapping, we refer to our approach as Single-dimension Software Pipelining (SSP).

An equally important problem is how to generate compact code for the constructed SSP schedule. The code generation problem involves issues related to register assignment through software and hardware renaming (to handle live range overlapping), controlling filling and draining of the pipelines, and limiting the code size increase. Details are published in [27].

Like many software-pipelining methods, we assume dynamic support for kernel [1]-only code [28]. In the absence of such support, our method can still produce efficient code, although with an increase in code size. We target ILP uniprocessors with support for predication [1, 5]. There is no constraint on the function units. They may or may not be pipelined. There is no constraint on the latencies of operations, either.

We have developed a stand-alone implementation of our method for the IA-64 architecture. The resulting code is run on an IA-64 Itanium machine and the actual execution time is measured. Our initial experimental results on a few loops reveal that the proposed approach achieves significant performance benefits over existing methods. Furthermore, we observe that the SSP approach is beneficial even in the presence of loop transformations, such as loop interchange, loop tiling, and unroll-and-jam.

We remark that the experimental results presented here are preliminary in nature, and are included here mainly to give a feel for the potential benefits of our method. The purpose of this paper is to introduce the SSP concept.

Our method shows several advantages:

- *Global foresight*: Instead of focusing only on the innermost loop, every loop level is examined and the most profitable one is chosen. Furthermore, any criterion can be used to judge the "profitability" in this loop selection step. This flexibility opens a new prospect to combine software pipelining with any other optimal criterion beyond the ILP degree, which is often chosen as the main objective for software pipelining. In this paper, we consider not only parallelism, but also cache effects, which have not been considered by most traditional software pipelining methods [1, 3, 14, 16, 17, 19, 20, 23, 24, 25, 30].
- *Simplicity*: The method retains the simplicity of the classical modulo scheduling of single loops. The scheduling is based on a simplified 1-dimensional DDG and is done only once, irrespectively of the depth of the loop nest. This is another essential difference from previous approaches. Also the traditional modulo scheduling of single loops is subsumed as a special case.
- *Efficiency*: Under identical conditions, our schedule provably achieves the shortest computation time[2] that could be achieved by the traditional innermost-loop-centric approach. Yet, since we search the entire loop nest and choose the most profitable loop level, the execution time may be even shorter. Besides, we consider reuse vector space to achieve potentially higher

1 The kernel of the 1-D schedule.

2 We differentiate between the terms "execution time" and "computation time" of a schedule. The latter refers to the estimated execution time of the schedule, while the former refers to the *actual* execution time measured by running the schedule on a real machine.

exploitation of data locality, which, in turn, further reduces the actual execution time of the schedule.

The method presented here can be applied to both imperfect and perfect loop nests [26, 27, 29]. However, for simplicity reasons, we restrict ourselves to perfect loop nests in this paper.

This paper is organized as follows. Section 2 introduces the basic concepts and briefly reviews modulo scheduling. Then we motivate our study by a simple example in Section 3. Section 4 discusses our method in detail. We prove the correctness and the efficiency of our approach in Section 5. Experimental results and performance comparison with other methods are reported in Section 6. A discussion on related work and concluding remarks are presented in Sections 7 and 8.

2. Basic Concepts

An *n-deep perfect loop nest* is composed of loops $L_1, L_2, \cdots, L_n$, respectively, from the outermost to the innermost level. Each loop $L_x (1 \leq x \leq n)$ has an index variable i_x and an index bound $N_x \geq 1$. The index is normalized to change from 0 to $N_x - 1$ with unit step. The loop body is assumed to have no branches; branches, if any, are converted to linear code by if-conversion [5].

The *iteration space* of the loop nest is a finite convex polyhedron [31, 7]. A node in the iteration space is called an *iteration point*, and is identified by the index vector $\boldsymbol{I} = (i_1, i_2, \cdots, i_n)$. The instance of any operation o in this iteration point is denoted by $o(\boldsymbol{I})$. An L_x *iteration* is one execution of the L_x loop. Thus the L_x loop has a total of N_x number of iterations. One such iteration is also an iteration point if L_x is the innermost loop.

We use $(o_1 \rightarrow o_2, \delta, \boldsymbol{d})$ to represent a data dependence from operations o_1 to o_2 in the loop nest, where o_1 and o_2 are called *source* and *sink* of the dependence, respectively; $\delta \geq 0$ is the *dependence latency*; and $\boldsymbol{d} = \langle d_1, d_2, \cdots, d_n \rangle$ is the *distance vector*,where d_1 is the distance at the outermost level, and d_n the innermost.

The *sign of a vector* is that of its first non-zero element, either positive or negative. If all elements are 0, the vector is a *null or zero vector*.

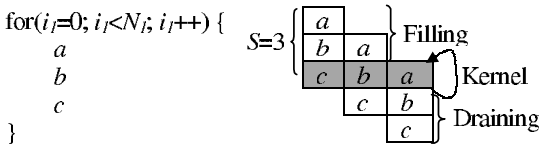
Software pipelining [1, 2, 8, 14, 16, 17, 20, 24, 25, 30] exposes instruction-level parallelism by overlapping successive iterations of a loop. Modulo scheduling (MS) [16, 17, 24, 25] for single loops is an important and probably the most commonly used approach of software pipelining. A detailed introduction can be found in [4].

In modulo scheduling, instances of an operation from successive iterations are scheduled with an *Initiation Interval (II)* of T cycles.

The schedule length of a (modulo) schedule is defined as the length or execution time of a single iteration. Let the *schedule length* of each iteration be l cycles. Then each iteration is composed of $S = \lceil \frac{l}{T} \rceil$ number of *stages*, with each stage taking T cycles.

The schedule consists of three phases: *filling the pipeline*, *repetitively executing the kernel*, and *draining the pipeline*.

Example: Fig.1a shows an example loop. Assume there are two dependences $(a \rightarrow b, 1, \langle 0 \rangle)$ and $(b \rightarrow c, 1, \langle 0 \rangle)$. Fig.1b shows a modulo schedule for the loop with $T = 1$, and $S = 3$.



(a) An example loop (b) Modulo schedule

Figure 1. Modulo scheduling of a single loop

3. Motivation and Overview

In this section, we motivate our method with the help of a simple 2-deep perfect loop nest. Subsequently, we illustrate our scheduling approach using the same example throughout the paper.

3.1. A Motivating Example

Fig. 2 shows a perfect loop nest in C language[3] and its data dependence graph (DDG). To facilitate understanding and without loss of generality, in this example, we assume that each statement is an operation. In the DDG, each node represents an operation and an edge represents a dependence labeled with the distance vector.

The inner loop has no parallelism due to the dependence cycle *a*→*b*→*a* at this level. Thus modulo scheduling of the innermost loop cannot find any parallelism for this example. Innermost-loop-centric software pipelining approach exposes extra parallelism, based on the modulo schedule, by overlapping the filling and draining of the pipelines between successive outer loop iterations. Since modulo scheduling cannot find any parallelism, there is no filling or draining and therefore no overlapping. Thus, innermost-loop-centric software pipelining cannot find any parallelism, either.

3 This loop nest certainly could be parallelized in a number of other ways, too. We only use it for illustration purposes.

```
L1:for (i1=0; i1 < N1; i1++){
L2:   for (i2=0; i2 < N2; i2++){
        a: U[i1 + 1][i2]=V[i1][i2]+
                         U[i1][i2];
        b: V[i1][i2 + 1]=U[i1 + 1][i2];
      }
  }
```

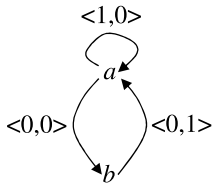


Figure 2. A loop nest and its DDG

One may argue that a loop interchange transformation before software pipelining will solve this problem. Unfortunately, that will destroy the data reuse in the original loop nest: for large arrays each iteration point will introduce 3 cache misses for accessing $U[i_1 + 1][i_2]$, $V[i_1][i_2]$, and $U[i_1][i_2]$.

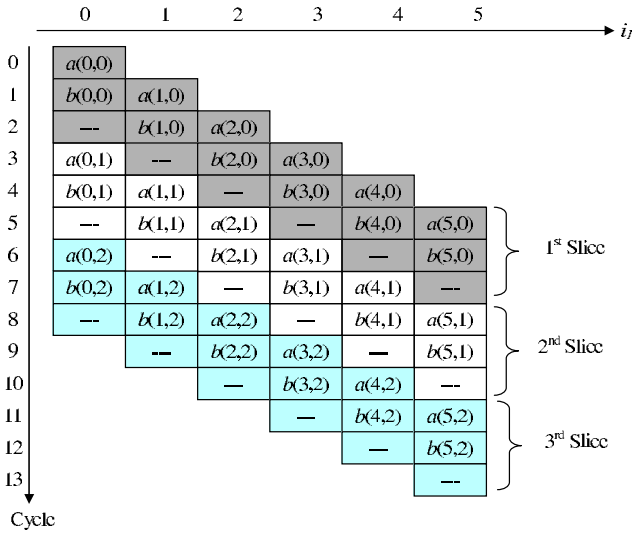
3.2. Overview and Illustration of Our Approach

The above example shows the limitation of the traditional software pipelining: It cannot see the whole loop nest to better exploit parallelism. Nor can it exploit the data reuse potentials of outer loops. This raises the question: Why not select a better loop to software pipeline, not necessarily the innermost one? This question brings the challenging problem of software pipelining of a loop nest. The challenge comes from two aspects: how to handle resource constraints? And how to handle the multi-dimensional dependences?

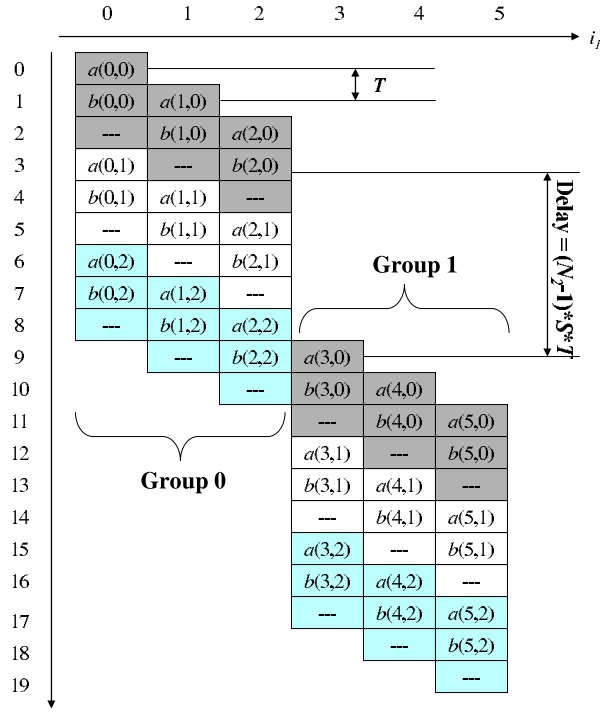
3.2.1. Which Loop to Software Pipeline? Parallelism is surely one of the main concerns. On the other hand, cache effects are also important and govern the actual execution time of the schedule. However, it is very hard to consider cache effects in traditional software pipelining, mainly due to the fact that it focuses on the innermost loop. Provided that an arbitrary loop level in a loop nest can be software pipelined, we can search for the most profitable level, measured by parallelism or cache effect, or both. Any other objective can also be used as a criterion.

3.2.2. How to Software Pipeline the Selected Loop? Suppose we have chosen a loop level, for simplicity, say L_1. We allocate the iteration points to a series of *slices*, and software pipeline each slice. Although any software pipelining method can be used, in this paper we focus on modulo scheduling.

For any $i_1 \in [0, N_1)$, iteration point $(i_1, 0, \cdots, 0, 0)$ is allocated to the first slice, and the next point $(i_1, 0, \cdots, 0, 1)$, in the lexicographic order, to the second slice, etc. If there were no dependences, and no constraints on the resources, all L_1 iterations could be executed in parallel, while the iteration points within each



(a) Software pipelined slices



(b) The final schedule after cutting and pushing down the slices

Figure 3. Illustration

of them are executed sequentially. However, due to dependences at the L_1 loop level, we may execute the iterations of L_1 in an overlapped (software pipelined) manner. Furthermore, due to the limited resources we may not exe-

```
L1': for (i1=0; i1<N1; i1+=3) {
  b(i1-1, N2-1)  a(i1,0 )
                 b(i1,0)   a(i1+1, 0)
                           b(i1 +1, 0)    a(i1+2, 0)
  L2': for (i2=1; i2<N2; i2++) {
                 a(i1, i2)                 b(i1 +2, i2-1)
                 b(i1, i2)  a(i1+1, i2 )
                           b(i1 +1, i2)    a(i1+2, i2)
    }
  }
                                           b(i1-1, N2-1)
```

Figure 4. Rewritten loops

cute all points in a slice simultaneously. Therefore we cut the slices into *groups*, with each group containing S iterations of L_1 loop. Then we push down (i.e. delay the execution of) the next group until resources are available.

Example: We illustrate the above thoughts with the loop nest example in Fig. 2. Let us assume operation a and b have a latency of 1 and 2 cycles, respectively. Further assume that we have two functional units, and both are pipelined and can perform any of the operations. For any i_1, point $(i_1, 0)$ is allocated to the first slice, and point $(i_1, 1)$ to the second, etc. Fig. 3(a) shows the allocation with $N_1 = 6$ and $N_2 = 3$. Here each slice is modulo scheduled so that successive iteration points within this slice initiate at an interval of $T = 1$ cycle [4]. The schedule length l equals 3, and therefore there are $S = \lceil \frac{l}{T} \rceil = \lceil \frac{3}{1} \rceil = 3$ stages.

Although the resource constraints are respected within each modulo scheduled slice, resource conflicts (between slices) arise when successive slices are issued greedily. To remove the conflicts, we cut the slices into groups, with each group having $S = 3$ iterations of L_1 loop. There are two groups in this schedule. Each group, except the first one, is pushed down by $(N_2 - 1) * S * T$ cycles relative to its previous group. The delay is designed to ensure that repeating patterns definitely appear. This leads to the *final schedule* that maps each instance of an operation to its schedule time, as shown in Fig. 3(b). Note that not only dependence and resource constraints are respected, but the parallelism degree exploited in a modulo scheduled slice ($S = 3$) is still preserved, and the resources are fully used. Lastly, it is straightforward to rewrite the final schedule in a more compact form (Fig. 4).

How to Handle Resources?

Resource constraints are enforced at two levels: at the slice level when we modulo schedule the slices and at the inter-slice level when we push down slices appropriately.

How to Handle Dependences?

A major obstacle to software pipelining of loop nests is how to handle n-dimensional distance vectors. The above approach, however, easily solves this problem. A key observation is that if a dependence is respected before pushing down the groups, it will be also respected after that because pushing down can only increase the time distance between the source and sink operations of the dependence. Therefore we only need to consider the dependences necessary to obtain the schedule before the push down.

There are two kinds of legal dependences: one is across two slices, and the other one is within a slice, as shown in Fig.5, where each parallelogram represents a slice, and each dot an iteration point. Although not shown on the picture, each slice is software pipelined.

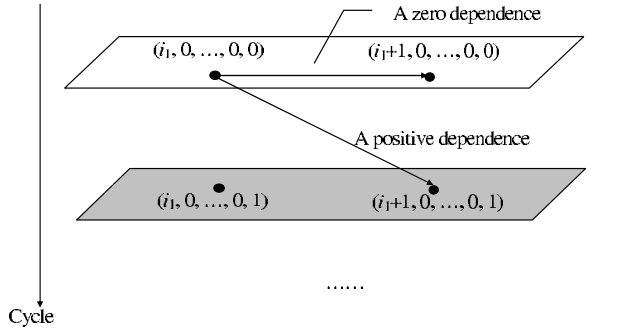


Figure 5. Dependences

Due to the way the iteration points are allocated, a dependence across two slices has a distance vector $\langle d_1, d_2, \cdots, d_n \rangle$, where $d_1 \geq 0$, and $\langle d_2, \cdots, d_n \rangle$ is a positive vector. Such a dependence is naturally resolved because the two slices are executed sequentially.

A dependence within a slice has a distance vector $\langle d_1, d_2, \cdots, d_n \rangle$, where $d_1 \geq 0$, and $\langle d_2, \cdots, d_n \rangle$ is a zero vector. Such a dependence has to be considered during software pipelining.

The two kinds of dependences are named *positive* and *zero dependences*, respectively. Note that a dependence from a slice to a previous slice is illegal. It is called a *negative dependence*. Negative dependences can be changed to be zero or positive dependences using loop skewing, as will be explained in Section 4.

In summary, we only need to consider zero dependences to construct the software-pipelined schedule.

4 Although the example in Fig. 3(a) is very simple, the principles apply to general cases as well. In general, a slice can simply be any possible modulo schedule.

4. Solution

In this section, we first classify dependences, and then formalize our approach into 3 steps: (1) loop selection, (2) dependence simplification and 1-D schedule construction, and (3) final schedule computation.

Let $\boldsymbol{d} = \langle d_1, d_2, \cdots, d_n \rangle$ be the distance vector of a dependence. We say that this dependence is *effective at loop level* $L_x (1 \leq x \leq n)$ iff $\langle d_1, d_2, \cdots, d_{x-1} \rangle = \mathbf{0}$ and $\langle d_x, d_{x+1}, \cdots, d_n \rangle \geq \mathbf{0}$, where $\mathbf{0}$ is the null vector with appropriate length. By *effective*, we mean that such a dependence must be considered when we software pipeline L_x. All effective dependences at L_x compose the *effective DDG* at L_x.

According to the definition, if a dependence is effective at L_x, we have $\langle d_x, d_{x+1}, \cdots, d_n \rangle \geq \mathbf{0}$ (And of course the first element $d_x \geq 0$). We *classify the dependence by the sign of the sub-distance-vector* $\langle d_{x+1}, \cdots, d_n \rangle$*, when* $x < n$. If this sub-vector is a zero, positive, or negative vector, the dependence is classified as a *zero*, *positive*, or *negative dependence at* L_x, respectively. When $x = n$, we classify it as a zero dependence at L_x for uniformity.

Example: Fig.5 illustrates a positive dependence at L_1 with a distance vector $\langle 1, 0, \cdots, 0, 1 \rangle$, and a zero dependence at L_1 with a distance vector $\langle 1, 0, \cdots, 0, 0 \rangle$.

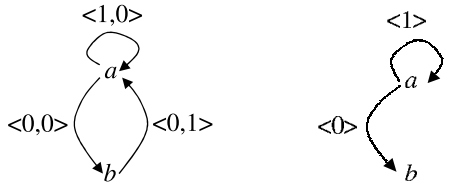
We must point out that the above classification is complete: *an effective dependence is in and only in one of the three classes*. Especially, the dependences are classified according to *the sign of the sub-distance-vector*, not that of the whole distance vector. For example, a dependence in a 3-deep loop nest with a distance vector of $\langle 1, -1, 2 \rangle$ is a negative dependence at L_1 because the sub-vector $\langle -1, 2 \rangle$ is negative, even though the whole distance vector is positive.

We classify only effective dependences since in the following sections, our discussion relates only to them. Although the dependence classification is dependent on the loop level, we will not mention the loop level when the context is clear.

As explained in Section 3.2.2, we assume that when we consider to software pipeline a loop level L_x, all effective dependences at this level are either zero or positive. Negative dependences cannot be handled directly. The loop nest must be transformed to make them zero or positive[5].

Moreover, only zero dependences need to be considered. Positive dependences can be ignored as they will be naturally honored in the final schedule. Lastly, only the dependence distance at L_x is useful for software pipelining. Thus we can reduce the effective DDG to have only zero dependences with 1-dimensional distance vectors. We refer to the resulting DDG as the *simplified DDG*. The definition is as follows: The *simplified DDG* at L_x is composed of all the zero dependences at L_x; the dependence arcs are annotated with the dependence distance at L_x.

Example: Fig.6(a) shows the effective DDG at L_1 for the loop nest depicted in Fig.2. There are two zero dependences in this DDG: $a \rightarrow a$ and $a \rightarrow b$. Associating the dependence distances at L_1 with the arcs, we get the simplified DDG shown in Fig.6(b).



(a) Effective DDG at L_1

(b) Simplified DDG at L_1

Figure 6. Dependence Simplification

4.1. Loop Selection

In this paper, our objective is to generate the most efficient software-pipelined schedule possible for a loop nest. Thus it is desirable to select the loop level with a higher initiation rate (higher parallelism), or a better data reuse potential (better cache effect), or both[6]. In this section, we address the essential problem of evaluating these two criteria.

4.1.1. Initiation Rate *Initiation rate*, which is the inverse of initiation interval, specifies the number of iteration points issued per cycle. Hence we choose the loop level L_x that has the maximum initiation rate, or minimum initiation interval.

The initiation interval T_{L_x} at loop level L_x can be estimated as:

$$T_{Lx} \geq max(RecMII_{L_x}, ResMII), \qquad (1)$$

5 It is always feasible to transform a negative dependence to a zero or positive dependence by loop skewing. However, after that, the iteration space becomes non-rectangular. Although we restrict to rectangular iteration spaces in this paper, the first two steps of SSP are still applicable to non-rectangular cases, without any change, since scheduling considers only DDG and hardware resources. It considers nothing about the shape of the iteration space. For the third step of SSP, in our SSP code generator for the IA-64 architecture in our experiments, we used predicate registers to dynamically form a non-rectangular iteration space in runtime, although the static code produced looks rectangular. The algorithms are neither presented here nor in the companion paper [27] due to the space limitation.

6 The specific decision is not made here, since that is implementation-dependent.

where *RecMII*$_{L_x}$ and *ResMII* are the minimum initiation intervals determined, respectively, by recurrences in the simplified DDG at L_x, and by the available hardware resources.

$$RecMII_{L_x} = \max_{\forall C} \frac{\delta(C)}{d(C)}, \quad (2)$$

where C is a cycle in the simplified DDG, $\delta(C)$ is the sum of the dependence latencies along cycle C, and $d(C)$ is the sum of the dependence distances along C [7].

For pipelined function units [4, 15],

$$ResMII = \max_{\forall \text{ resource type } r} \frac{\text{total operations that use } r}{\text{total resources of type } r} \quad (3)$$

We omit the details of the *ResMII* calculation for non-pipelined function units. Readers are referred to [15].

In addition to the initiation rate, we also look at the trip count of each loop level. In particular, the trip count should not be less than S, the number of stages in the schedule. Otherwise, this loop should not be chosen.

The reason is that the slices are cut in groups, where each group has S iterations of L_x loop. Then the trip count N_x is expected to be divisible by S. Otherwise, the last group will have fewer L_x iterations, resulting in a lower utilization of resources in that group. However, when $N_x > S$, it is always possible to apply loop peeling to avoid the situation.

Although S is unknown at loop selection time, it is generally small because the limited resources in a uniprocessor cannot support too many stages at the same time. As a guideline, a small estimated value can be set for S.

4.1.2. Data Reuse When we software pipeline a loop level, the data reuse potential can be measured by the average number of memory accesses per iteration point. The fewer the accesses, the greater the reuse potential. Without loss of generality, let us consider loop L_1.

In our approach, software pipelining results in S iterations of L_1 loop running in a group, which is composed of a series of slices. Select the first S number of successive slices in the first group. They include the following set of iteration points: $\{(i_1, 0, \cdots, 0, i_n) | \forall i_1 \text{ and } i_n \in [0, S)\}$, which is an $S * S$ square in the iteration space. This is a typical situation in our method, because L_1 iterations are executed in parallel, and the index of the innermost loop changes more frequently than the other indexes. Therefore we could estimate the memory accesses of the whole loop nest by those of the iteration points in this set. This set can be abstracted as a *localized vector space* $\alpha = span\{(1, 0, \cdots, 0), (0, \cdots, 0, 1)\}$. Now the problem is very similar to that discussed in Wolf and Lam [31]. Below we briefly describe the application of their method in this situation.

For a *uniformly generated set* in this localized space, let R_{ST} and R_{SS} be the self-temporal and self-spatial reuse vector space, respectively. And let *gT* and *gS* be the number of group temporal and group-spatial equivalent classes [8]. Then for this uniformly generated set, the number of memory accesses per iteration point is [31]:

$$\frac{gS + (gT - gS)/l}{l^e S^{\dim(R_{SS} \cap \alpha)}}, \quad (4)$$

where *l* is the cache line size, and

$$e = \begin{cases} 0 & \text{if } R_{ST} \cap \alpha = R_{SS} \cap \alpha, \\ 1 & \text{otherwise.} \end{cases}$$

The total memory accesses per iteration point are the sum of accesses for each uniformly generated set.

4.2. Dependence Simplification and 1-D Schedule Construction

Our method software pipelines only the selected loop level. Enclosing outer loops, if any, are left as they are. Therefore, without loss of generality, we consider L_1 as the selected level. We simplify dependences to be single-dimensional first, then schedule operations.

4.2.1. Dependence Simplification As mentioned already, given the effective DDG at L_1, we can simplify the dependences to obtain a simplified DDG, which consists of only zero dependences with 1-dimensional distance vectors. To make sure positive dependences are respected, during this simplification process, we also compute $\delta_{max}(o)$ for each operation o, where:

$$\delta_{max}(o) = \max_{\forall \text{ positive dependence } (o \to o', \delta, \boldsymbol{d})} \delta, \quad (5)$$

The value of $\delta_{max}(o)$ is equal to 0 if there is no positive dependence with o as the source operation. The value will be used for scheduling operations so that positive dependences will be naturally honored in the final schedule.

Example: For the loop nest in Fig.2, its effective and simplified DDG at L_1 are shown in Fig.6(a) and Fig.6(b). $\delta_{max}(a) = 0$ because a is not the source of any positive dependence, and $\delta_{max}(b) = 2$ because of the positive dependence $b \to a$ with a 2-cycle latency.

7 The reader will find in section 4.2.2 that our scheduling method has extra Sequential Constraints added to modulo scheduling. They affect only schedule length in the 1-D schedule, but not the initiation interval (In the worst case, we can always increase schedule length to satisfy these constraints). Thus they will not affect the $RecMII$ or $ResMII$ values.

8 These symbols are consistent with those in [31], and their values can be computed by using the method in [31].

4.2.2. 1-D Schedule Construction Based *solely* on the simplified DDG and the hardware resource constraints, we construct a 1-D schedule. Since the DDG is 1-dimensional, from the viewpoint of scheduling, L_1 is treated as a single loop as if it had no inner loops. Any modulo scheduling method can be applied to schedule L_1 to obtain a 1-D schedule.

Let T be the initiation interval of the generated schedule, and S be the number of stages of the schedule. We refer to the schedule as a *1-D schedule* for the loop level L_1. Let the schedule time for any operation instance $o(i_1)$ be $\sigma(o, i_1)$[9]. The 1-D schedule must satisfy the following properties:

1. Modulo property:

$$\sigma(o, i_1) + T = \sigma(o, i_1 + 1) \tag{6}$$

2. Dependence constraints:

$$\sigma(o_1, i_1) + \delta \leq \sigma(o_2, i_1 + k) \tag{7}$$

   for every dependence arc $(o_1 \rightarrow o_2, \delta, \langle k \rangle)$ in the simplified DDG.

3. Resource constraints: During the same cycle, no hardware resource is allocated to more than one operation.

4. Sequential Constraints:

$$S*T - \sigma(o, 0) \geq \delta_{max}(o) \text{ for any operation } o \text{ if } n > 1. \tag{8}$$

The first three constraints are exactly the same as those of the classical modulo scheduling [4, 15]. We have added the sequential constraints to enforce sequential order between successive iteration points in the same L_1 iteration. This ensures that all positive dependences are honored at runtime.

Example:. Fig.7 shows the 1-D schedule constructed for the example loop nest in Fig.2. The 1-D schedule is based on the simplified DDG in Fig.6(b). As mentioned earlier, we have assumed two homogeneous functional units, and an execution latency of 1 and 2 cycles for operations a and b. The schedule has an initiation interval of 1 cycle ($T = 1$) and has 3 stages ($S = 3$). Also, $\sigma(a, i_1) = 0 + i_1 * T$ and $\sigma(b, i_1) = 1 + i_1 * T$.

4.3. Final Schedule Computation

As explained in Section 3.2.2, we first allocate iteration points in the loop nest to slices: for any $i_1 \in [0, N_1)$, iteration point $(i_1, 0, \cdots, 0, 0)$ is allocated to the first slice, $(i_1, 0, \cdots, 0, 1)$ to the second slice, and so on. Then we software pipeline each slice by applying the 1-D schedule to it.

If the successive slices are greedily issued without considering resource constraints across the slices (That is, two adjacent slices are put together without any "hole"between them. This can be done because the slices have the same shape), then we obtain the schedule like that in Fig.3(a). However, note that, within each slice, the resource constraints are honored during the construction of the 1-D schedule. Now, to enforce resource constraints across slices, we cut the slices in groups, with each group having S number of L_1 iterations. Each group, except the first one, is delayed by a given number of cycles as shown in Fig.3(b).

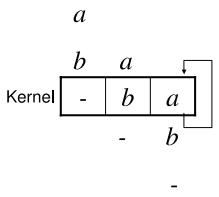


Figure 7. The 1-D schedule

Using the above procedure, a final schedule can be constructed for the loop nest. Such a schedule can be precisely defined by the following mapping function. For any operation o and iteration point $\boldsymbol{I}$=$(i_1, i_2, \ldots, i_n)$, schedule time $f(o, \boldsymbol{I})$ is given by

$$\begin{aligned} f(o, \boldsymbol{I}) \quad = \quad & \sigma(o, i_1) + \sum_{2 \leq x \leq n} (i_x * (\prod_{x < y \leq n+1} N_y) * S * T) + \\ & \left\lfloor \frac{i_1}{S} \right\rfloor * ((\prod_{2 \leq x \leq n+1} N_x) - 1) * S * T, \end{aligned} \tag{9}$$

where N_{n+1}=1.

Let us briefly explain how the above equation is derived. First, let us consider the greedy schedule before cutting and pushing down the groups. For this schedule, the schedule time of $o(\boldsymbol{I})$ is equal to that of $o(i_1, 0, \cdots, 0)$ plus the time elapsed between the schedule times of $o(i_1, 0, \ldots, 0)$ and $o(\boldsymbol{I})$. Since $o(i_1, 0, \cdots, 0)$ is in the first slice, the schedule time of $o(i_1, 0, \cdots, 0)$ is simply equal to $\sigma(o, i_1)$, the mapping function of the 1-D schedule. Note that this corresponds to the first term of the right-hand side (RHS) of Equation (9).

Next, between iterations $o(i_1, 0, \ldots, 0)$ and $o(i_1, i_2, \ldots, i_n)$, there are i_2*(N_3*N_4*..*N_n) + i_3*(N_4*N_5*..*N_n)+...+i_n number of iteration points. These points execute sequentially and each of them takes S*T cycles. Thus, the time elapsed between the schedule times of $o(i_1, 0, \ldots, 0)$ and $o(i_1, i_2, \ldots, i_n)$ is equal to

$$\sum_{2 \leq x \leq n} (i_x * (\prod_{x < y \leq n+1} N_y) * S * T). \tag{10}$$

Note that this corresponds to the second term of the RHS of Equation (9).

Next we discuss the effect of grouping and pushing down the slices. Iteration point $o(\boldsymbol{I})$ is located in group $\lfloor \frac{i_1}{S} \rfloor$. Each

9 $0 \leq \sigma(o, i_1) < S * T$, when $i_1 = 0$.

group in a slice is delayed by $\lfloor \frac{i_1}{S} \rfloor * w$ cycles, where w is the delay between two successive groups. For the example in Fig.3(b) with the 2-deep loop nest, we see that $w = (N_2 - 1) * S * T$. In general, for an n-deep loop nest, $w =$ (total iteration points in an L_1 iteration $- 1) * S * T$. Therefore the group where $o(\boldsymbol{I})$ is located is pushed down by

$$\left\lfloor \frac{i_1}{S} \right\rfloor * ((\prod_{2 \le x \le n+1} N_x) - 1) * S * T \qquad (11)$$

cycles. This is precisely the third term in Equation (9).

Example. To illustrate the mapping function for the final schedule, consider the 2-deep loop nest in Fig.2. From the 1-D schedule in Fig.7, we know that $S = 3$, $T = 1$, and $\sigma(a, i_1) = 0 + i_1 * T$. For any operation instance $a(i_1, i_2)$, we have the final schedule

$$f(a, (i_1, i_2)) = i_1 + i_2 * 3 + \left\lfloor \frac{i_1}{3} \right\rfloor * (N_2 - 1) * 3.$$

For instance, when $N_2 = 3$, we have $f(a, (4, 1)) = 13$, as shown in Fig.3(b).

5. Analysis

In this section, we establish the correctness and efficiency of the SSP approach, and its relationship with MS.

5.1. Correctness and Efficiency

Theorem 1 *The final schedule defined in Equation (9) respects all the dependences in the effective DDG and the resource constraints.*

Section 3.2.2 has described the intuition of the above theorem. The complete proof is documented in [29].

Next, we demonstrate the efficiency of the SSP approach over other innermost-loop-centric software pipelining methods from the viewpoint of computation time of the constructed schedule. In particular, we compare our approach with modulo scheduling of the innermost loop (MS), and modulo scheduling of the innermost loop and overlapping the filling and draining parts of the outer loops, referred to as extended modulo scheduling (xMS) in this paper [20] . Let us define the *computation time* as the schedule time of the last operation instance+1.

Theorem 2 *For an n-deep loop nest, suppose that MS, xMS, and SSP, find the same initiation interval T and stage number S. Furthermore, suppose that the SSP method chooses the outermost loop L_1, which has a trip count N_1. If N_1 is divisible by S, then the computation time of the final schedule of SSP is no bigger than that of the MS or xMS schedule.*

The complete proof is also documented in [29]. Intuitively, this theorem holds because the final schedule produced by SSP always issues one iteration points every T cycles, without any hole, as can be seen from the example in Fig 3(b).

The above theorem assumes that N_1 is divisible by S. If not, since $N_1 \ge S$ (according to the discussion in Section 4.1.1) and S is typically very small, we can always peel off some L_1 iterations to make it divisible. In this way, we can assure at least the same performance as that of MS or xMS.

Since we search globally for the loop with the smallest initiation interval T, which dominates the computation time of a (final) schedule [29], our computation time can be even better. Furthermore, since we use an accurate estimation model for data reuse to exploit data locality, the execution time of our schedule will be lower than that of the schedules generated by MS or xMS methods.

5.2. Relation with the Classical Modulo Scheduling of a Single Loop

If the loop nest is a single loop (n=1), our sequential constraints in the formula (8) are redundant. Other constraints are exactly the same as the those of the classical modulo scheduling. And the final schedule is $f(o, (i_1)) = \sigma(o, i_1)$. In this sense, classical MS is subsumed as a special case of our method.

The time complexity of SSP method is similar to MS. It is bounded by $O(u^3)$ or $O(u^4)$, where u is the number of operations [29]. The specific complexity depends on the loop selection criteria and modulo scheduling being used, and usually is limited to $O(u^3)$. $O(u^4)$ is an extreme case that happens when n (the number of loops) is close to u, which does not happen frequently in real world.

6. Experiments

We briefly introduce and analyze our experiments conducted on the IA-64 architecture. Detailed performance curves and cache misses, and their analysis are reported in the technical memo [29].

6.1. Experimental Setup

Our experimental platform is an IA-64 Itanium workstation with a 733MHZ processor, 2GB main memory, and 16KB/96KB/2MB L1/L2/L3 caches. The method was implemented as a stand-alone module. Our implementation addresses a number of implementation difficulties, including compile-time register renaming, filling/draining control, predicated code generation, rotating registers usage, and code size reduction. Interested reader is referred to our

companion paper [27] for details. The compilation time was negligible.

Huff's modulo scheduling method [16] was used to implement the 1-D scheduler. Using the generated schedule, the code generator produces two versions of the same program: one is performance-optimized, and the other is code-size optimized [27]. We refer to them as *SSP* and *CS-SSP*, respectively.

For comparisons, we chose important loop kernels from scientific applications: matrix multiply (MM), LU decomposition (LU), SOR relaxation (SOR), and modified 2-D hydrodynamics (HD) from the Livermore loops [21]. We compare the performance of the SSP method against modulo-scheduling method [16] and an optimized version of it where the prolog and epilog of consecutive iterations overlap [20]. We refer to them as *MS* and *xMS* respectively. We also applied three loop transformations to test each method in a different context for MM: loop interchange (6 different versions), loop tiling and unroll-and-jam.

To compare and analyze performance, we measured the actual execution time of the schedules from each method on the Itanium machine, and also measured the cache misses using the IA64 performance monitoring tool, Pfmon.

6.2. Performance Analysis

In all our experiments the SSP method always outperforms MS and xMS, even when loop interchange, loop tiling or unroll-and-jam is applied beforehand. Being able to choose a more profitable loop level clearly proved itself to be an advantage over MS and xMS.

Loop selection is important to exploit maximal parallelism. For MM-jik, MM-ijk and SOR, a recurrence cycle in the innermost loop prevents a better overlapping of iterations. However, by software-pipelining the outermost loop, SSP and CS-SSP achieves a higher speedup, e.g., 1.5 times faster than MS and xMS schedules on average for MM-ijk.

Loop selection is also important to reduce memory accesses and improve the real performance of software pipelined schedules. For MM-ikj, MM-jki, MM-kij, MM-kji,HD, and LU, the cache misses measurement shows that SSP schedules result in lower L2 cache misses, and lower or the same L3 cache misses, although there is no much difference in the parallelism exploited in each schedule. SSP was then able to get around limited data reuse opportunities of the innermost loop. As a consequence, the real performance of SSP and CS-SSP schedules are higher, e.g., 1 time faster than MS and xMS schedules on average for MM-ikj.

In low-level details, the advantage of SSP scheduling and code generation shows clearly in these tiled and register-tiled MM. Although the loops tested are perfect in high-level language, they become imperfect at low level. After tiling or register-tiling, the depth of the whole loop nest becomes deeper (from 3 to 5). And thus it becomes more important to offset the overhead of running the operations outside the innermost loop. In the tiled or register tiled loop, these operations are more frequently executed, since the inner loops have small trip counts. Thus effective scheduling of these operations are very important. Although not introduced in this paper, SSP considers scheduling these operations at the very beginning of building the 1-D DDG. The result is a compact schedule, with all hardware resources fully exploited. In comparison, xMS and MS mainly care about the efficiency of running the innermost loop operations and their software pipelined kernel includes only such operations. Other operations are inserted or executed as normal in the schedule, depending on the resources, dependences, and the validity concerns. Our experiments indicates that the bundle densities of SSP and CS-SSP are more than 10% higher than that of xMS [27]. This, in turn, leads to around 20% performance improvement.

Method	MS	xMS	SSP	CS-SSP
MM-ijk	1.0	1.0	2.5	2.6
MM-jik	1.0	1.0	3.9	4.0
MM-ikj	2.5	2.7	5.4	5.5
MM-jki	2.5	2.5	5.2	5.3
MM-kij	2.5	2.5	3.0	3.0
MM-kji	2.0	2.3	3.3	3.3
HD	1.6	1.6	2.0	2.0
SOR	1.2	1.2	2.5	2.5
LU	2.4	2.4	2.8	3.0
MM-jki with loop tiling	7.7	8.5	10.4	10.7
MM-jki with unroll-and-jam	11.6	11.9	14.4	12.7

Table 1. Average speedups

7. Related Work

Most software pipelining algorithms [1, 3, 4, 16, 24, 25] focus on the innermost loop, and do not consider cache effects. The most commonly used method, modulo scheduling [1, 4, 16, 24], is a special case of our method.

A common extension of modulo scheduling from single loops to loop nests, including *hierarchical reduction* [17], *OLP* [20], and *pipelining-dovetailing* [30], is to apply modulo scheduling hierarchically in order to exploit the parallelism between the draining and filling phases. The modulo reservation table and dependence graph need to be reconstructed before scheduling each level. In comparison, our method considers cache effects and performs scheduling

only once, independent of the depth of the loop nest. The draining and filling phases are naturally overlapped without any special treatment.

Next we address the question: *What is the difference between our method and the one that interchanges the selected loop with the innermost loop, and then software pipelines the new innermost loop with MS?* First, it may not always be possible to interchange the two loops. For example, if a dependence in a 3-deep loop nest has a distance vector of $\langle 1, 1, -1 \rangle$ and our method selects the outermost loop, it is not legal to interchange this loop with the innermost loop. Second, even if they are interchangeable, the resulting schedules have different runtime behavior due to different data reuse patterns. And for this interchanged loop nest, the best choice still has to be made by global searching. This is best explained by our experiments on matrix multiply *ijk*-order and, after interchange, the *kji*-order. MS does improve performance after interchange. However, our method performs even better. Third, in some situations, interchange may be a bad choice, as we discussed in Section 3.1. Lastly, loop interchange can be beneficial to SSP as well, as can be seen from the experiments on different versions of MM.

Loop Tiling [31] aims to maximize cache locality, instead of parallelism. Loop unrolling [6] duplicates the loop body of the innermost loop to increase instruction-level parallelism. Both methods are complementary to SSP.

Unroll-and-jam [9, 6] has been applied to improve the performance of software pipelined loops [8]. The outer loop is unrolled and but it is still the innermost loop that is software-pipelined. In other words, the $RecMII$ still strongly depends on the recurrences in the innermost loop, though reduced by the unroll factor. On the other hand, the $RecMII$ of an SSP schedule depends on the recurrences in the chosen loop level. Thus, by choosing an appropriate loop level in the loop nest, the SSP method can achieve a lower initiation interval that might not be attainable via unroll-and-jam followed by MS.

Unroll-and-squash first applies unroll-and-jam to a nested loop, and then "squash" the jammed innermost loop to generate software pipelined code [22]. SSP is different from unroll-and-squash in the following ways: (1) the unroll-and-squash method presented in [22] appears to be limited to 2-deep loop nest; (2) it does not overlap the epilog and prolog and requires optimizations such as xMS to achieve this; and (3) it decides on the unroll factor first, and then software pipelines the innermost loop. An in-depth comparison of SSP with unroll-and-jam and unroll-and-squash is left for future work.

Hyperplane scheduling [18] is generally used in the context of large array-like hardware structures (such as systolic arrays and SIMD arrays), and does not consider resource constraints. There has been an interesting approach recently that enforces resource constraints to hyperplane scheduling by projecting the n-D iteration space to an $(n-1)$-D virtual processor array, and then partitioning the virtual processors among the given set of physical processors [12]. This method targets parallel processor arrays, and does not consider low-level resources (like the function units within a single processor) or cache effects. A subsequent software pipelining phase may need to be applied to each physical processor in order to further exploit instruction-level parallelism from the iterations allocated to the same processor.

Other hyperplane-based methods [10, 11, 23, 14] formulate the scheduling of loop nests as linear (often integer linear) programming problems. Optimal solutions to integer programming have exponential time complexity in the worst case when using the Simplex algorithm or branch-and-bound methods [7]. Furthermore, they consider neither resource constraints nor cache effects.

Unimodular and non-unimodular transformations [7, 13] mainly care for coarse-grain parallelism or the communication cost between processors.

Fine-grain wavefront transformation [2] combines *loop quantization* and *perfect pipelining* [3] to explore coarse and fine-grain parallelism simultaneously. It is based on outer loop unrolling and repetitive pattern recognition. There are two difficult problems involved here, namely, determining the optimal unrolling degree and identifying functional equivalent nodes [3].

8. Conclusion

We have introduced a unique 3-step approach to software pipeline a loop nest at an arbitrary level. This approach reduces the challenging problem of n-dimensional software pipelining into a simpler problem of 1-dimensional software pipelining. Our method, referred to as Single-dimension Software Pipelining (SSP), provides the freedom to search for and schedule the most profitable level, where profitability can be measured in terms of parallelism exploited, data reuse potential, or any other criteria.

We demonstrated the correctness and efficiency of our method. The schedule generated by our method was shown to naturally achieve the shortest execution time compared to traditional innermost-loop-centric modulo scheduling methods due to better data reuse, instruction- and loop-level parallelism, and/or code generation.

Future work includes the study of different aspects of SSP such as register allocation, interaction with other loop nest transformations, and affordable hardware support for kernel-only code generation.

Acknowledgments

We are grateful to Prof. Bogong Su, Dr. Hongbo Yang, and the anonymous reviewers of the previous versions of this paper for their patient reviews and valuable advice.

References

[1] *Intel IA-64 Architecture Software Developer's Manual, Vol. 1: IA-64 Application Architecture*. Intel Corp., 2001.

[2] A. Aiken and A. Nicolau. Fine-grain parallelization and the wavefront method. *Languages and Compilers for Parallel Computing, MIT Press, Cambridge, MA*, pages 1–16, 1990.

[3] A. Aiken, A. Nicolau, and S. Novack. Resource-constrained software pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1248–1270, Dec. 1995.

[4] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[5] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conf. Record of the Tenth Annual ACM Symp. on Principles of Programming Languages*, pages 177–189, Austin, Texas, January 1983. ACM SIGACT and SIGPLAN.

[6] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence Based Approach*. Morgan Kaufman, San Francisco, 2002.

[7] U. Banerjee. *Loop transformations for restructuring compilers: the foundations*. Kluwer Academic, Boston, 1993.

[8] S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In *Proc. 29th Annual Hawaii Intl. Conf. on System Sciences*, pages 183–192, 1996.

[9] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. on Prog. Lang. and Systems*, 16(6):1768–1810, Nov. 1994.

[10] A. Darte and Y. Robert. Constructive methods for scheduling uniform loop nests. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):814–822, Aug. 1994.

[11] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhuser, Boston, 2000. 280 p.

[12] A. Darte, R. Schreiber, B. R. Rau, and F. Vivien. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Trans. Des. Autom. Electron. Syst.*, 7(1):159–172, 2002.

[13] P. Feautrier. Automatic parallelization in the polytope model. *Lecture Notes in Computer Science*, 1132:79–103, 1996.

[14] G. R. Gao, Q. Ning, and V. Van Dongen. Software pipelining for nested loops. ACAPS Tech Memo 53, School of Computer Science, McGill Univ., Montréal, Québec, May 1993.

[15] R. Govindarajan, E. R. Altman, and G. R. Gao. A framework for resource-constrained rate-optimal software pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1133–1149, November 1996.

[16] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the ACM SIGPLAN '93 Conf. on Prog. Lang. Design and Implementation*, pages 258–267, Albuquerque, New Mexico, June 23–25, 1993. *SIGPLAN Notices,* 28(6), June 1993.

[17] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, Georgia, June 22–24, 1988. *SIGPLAN Notices,* 23(7), July 1988.

[18] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[19] S.-M. Moon and K. Ebcioğlu. Parallelizing nonnumerical code with selective scheduling and software pipelining. *ACM Transactions on Programming Languages and Systems*, 19(6):853–898, Nov. 1997.

[20] K. Muthukumar and G. Doshi. Software pipelining of nested loops. *Lecture Notes in Computer Science*, 2027:165–??, 2001.

[21] T. Peters. Livermore loops coded in c. http://www.netlib.org/benchmark/livermorec.

[22] D. Petkov, R. Harr, and S. Amarasinghe. Efficient pipelining of nested loops: unroll-and-squash. In *16th Intl. Parallel and Distributed Processing Symposium (IPDPS '02)*, Fort Lauderdale, FL, Apr. 2002. IEEE.

[23] J. Ramanujam. Optimal software pipelining of nested loops. In *Proc. of the 8th Intl. Parallel Processing Symp.*, pages 335–342, Cancún, Mexico, April 1994. IEEE.

[24] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, San Jose, California, November 30–December2, 1994.

[25] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview and perspective. *Journal of Supercomputing*, 7:9–50, May 1993.

[26] H. Rong. *Software Pipelining of Nested Loops*. PhD thesis, Tsinghua University, Beijing, China, 2001.

[27] H. Rong, A. Douillet, R. Govindarajan, and G. R. Gao. Code generation for single-dimension software pipelining of multi-dimensional loops. In *Proc. of the 2004 Intl. Symp. on Code Generation and Optimization (CGO)*, March 2004.

[28] H. Rong and Z. Tang. Hardware controlled shifts and rotations supporting software pipelining of loop nests. *China Patent*, November 2000. #00133535.9.

[29] H. Rong, Z. Tang, A. Douillet, R. Govindarajan, and G. R. Gao. Single-dimension software pipelining for multi-dimensional loops. CAPSL Technical Memo 49, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, September 2003. In ftp://ftp.capsl.udel.edu/pub/doc/memos/memo049.ps.gz.

[30] J. Wang and G. R. Gao. Pipelining-dovetailing: A transformation to enhance software pipelining for nested loops. In *Proc. of the 6th Intl. Conf. on Compiler Construction, CC '96*, volume 1060 of *Lecture Notes in Computer Science*, pages 1–17, Linkoping, Sweden, April 1996.

[31] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proc. of the ACM SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, pages 30–44, Toronto, June 26–28, 1991. *SIGPLAN Notices,* 26(6), June 1991.

[32] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proc. of the 29th Annual Intl. Symp. on Microarchitecture (MICRO 29)*, pages 274–286, Paris, December 2–4, 1996.